Master of Science Thesis

Redesign of the Oz Compiler

Markus Bohlin     Lars Bruce
bola@sics.se      bruce@sics.se

Department of Computer Engineering, Mälardalen University

Distributed Systems Lab, Swedish Institute of Computer Science

Examiner: Björn Lispé
Supervisor at SICS: Per Brand

SICS Technical Report T2002:08
ISSN 1100-3154
ISRN:SICS-T-2002/08-SE

June 12, 2002

Abstract

This master of science thesis describes a new design and its implementation for an Oz compiler. The project is based on the existing Oz compiler. The new compiler is designed more modular, with separate software components that can be replaced and modified locally. A prototype has been implemented, but further development is necessary.

We give an overview of the language Oz, its features and the underlying calculus. The features of Oz regarding object orientation, functional programming, logic and constraint programming are also discussed.

The liveness analysis and register allocation problems in general and regarding Oz specific compilers are analyzed, together with current and future optimizations suitable for the Mozart platform.

The design of the new compiler and information about the old one is presented, and future work regarding the compiler, optimizations, and analysis phases is discussed.

Appendices describing the interfaces between the phases of the compiler is included, together with documentation regarding the internal code formats used.
Acknowledgements

We foremost thank the people at the Distributed System Lab at SICS, who have been very helpful and encouraging, always willing to share their knowledge of the Mozart system.

We wish to express our gratitude to our supervisor at SICS, Per Brand. His knowledge of the virtual machine and the foundations of Oz is impressive, and we have had great benefit from his expertise.

We thank our supervisor and examiner Björn Lispér at the Department of Computer Engineering, Mälardalen University, for his thorough reading and corrections of the language and contents of this thesis. His broad knowledge in the field of computer science is admirable.

Leif Kornstaedt, the original author of the Oz compiler has given us lots of help, via the numerous (sometimes stupid) questions he has answered and the insights he has given us into the compiler design and implementation, both when he was visiting SICS and when answering our mail.

Our closest supervisor Nils Franzén at SICS has been supporting and helping us a lot along the way with everything from the language Oz to how to use Emacs correctly.

We would also like to thank our colleagues at the Department of Computer Engineering in Västerås: Jouni Axelsson, Johan Erikson, Waldemar Kocjan, Andreas Sjögren, Baran Çürüklü, Mikael Sandberg, Jan Carlsson, Robert Olofsson, and Rikard Lindell for their support and encouragement during the past year.

Markus Bohlin and Lars Bruce, March 2001
Contents

List of Figures vii
List of Tables ix

1 Introduction 1

2 Project Description 3
  2.1 Project Description 3
  2.2 Clarifying Notes 4
  2.3 Achievements 4
  2.4 Known Bugs 5
  2.5 Comparison 5

3 The Language Oz 7
  3.1 Features in Oz 7
  3.2 Mozart and Oz Overview 8
  3.3 Logic Variables 8
  3.4 Unification 9
  3.5 Core Language 11
  3.6 Some Oz Examples 15
  3.7 Concurrency 16
  3.8 Functional Programming in Oz 18
  3.9 Logic Programming 20
  3.10 Constraint Programming 23
  3.11 Object Orientation is Syntactic Sugar in Oz 27
  3.12 Classes 27
  3.13 Objects 27
  3.14 Private and Protected Methods 28
  3.15 Inheritance 28
  3.16 Overhead 28
  3.17 Example 28
  3.18 The γ-calculus 30
4 The Mozart Virtual Machine 35
  4.1 Registers in MVM ................................. 35
  4.2 The MVM Variable System ......................... 36
  4.3 Instruction Set ................................ 38
  4.4 Unification in the MVM ............................ 40
  4.5 Builtins .................................. 45

I Compiler Techniques 46

5 Liveness Analysis 47
  5.1 Liveness Analysis ............................... 47
  5.2 Disjoint Paths .................................. 50
  5.3 Improving the Liveness Analysis .................. 52

6 Physical Register Allocation 55
  6.1 The Goal of Register Allocation ................. 55
  6.2 Local Register Allocation ........................ 56
  6.3 The Graph-Coloring Problem .................... 56
  6.4 Other Approaches to Register Allocation ........ 58

7 Optimizations 61
  7.1 Peephole Optimization ........................... 61
  7.2 Tail Call Optimization ........................... 62
  7.3 Inlining ..................................... 62
  7.4 Delayed initialization ........................... 64
  7.5 Future Oz-specific Optimizations ............... 66

II Design 68

8 Overview of the New Compiler 69
  8.1 Area of Interest ................................ 69
  8.2 Design Overview ................................ 69
  8.3 Phases ....................................... 69
  8.4 Abstract Instructions ............................ 72

9 The Compiler Front-End 76
  9.1 The Parser ................................... 76
  9.2 The Object Graph ............................... 77
  9.3 Static Analysis Phase ............................ 79

10 Transformation to Intermediate Code 80
  10.1 From Object Graph to Abstract Instructions .... 80
  10.2 Obstacles Encountered .......................... 81
## 11 Applied Register Allocation 82
- 11.1 Register Optimization Strategies 82
- 11.2 The Original Register Allocator 83
- 11.3 The New Register Allocator 83
- 11.4 Overview 83
- 11.5 Register Constraints 84
- 11.6 Temporary Registers 84
- 11.7 Deciding Which Register to Allocate 85
- 11.8 Merging 85
- 11.9 Handling Function Calls 86

## 12 Code Emission 89
- 12.1 From Abstract Instruction to Bytecode 89
- 12.2 New Code Structure 89
- 12.3 Register Handling 92

## 13 Assembling 93
- 13.1 Peephole Optimization 93
- 13.2 Absolute Addressing 94
- 13.3 Loading 95

## III Future Work 96

## 14 Future Work 97
- 14.1 Redesign 97
- 14.2 Compiler Efficiency 98
- 14.3 Optimizations 99
- 14.4 Bugs 99

## IV Appendices 100

### A Interface to the Back-end Phases 101
- A.1 General Notes 101
- A.2 Abstract Code Generation 101
- A.3 Liveness Analysis 102
- A.4 Register Allocation 103
- A.5 Bytecode Emission 105
- A.6 Peephole Optimization 106
- A.7 Absolute Code Generation 106
- A.8 Loading 106
B Syntax Tree Format 107
  B.1 Compilation ........................................... 107
  B.2 The Base Language .................................. 107
  B.3 Finite Domain Extensions and Combinators ......... 111
  B.4 Gump Extensions ..................................... 111

C The Object Graph 114
  C.1 Oz Core Language: Graph Representation ............ 114

D Abstract Instructions 124
  D.1 Summary of the Abstract Instructions ............... 124
  D.2 Acronyms ............................................. 125
  D.3 Phases ................................................ 125
  D.4 The Abstract Instructions in Detail ................. 126

E Mozart Virtual Machine Bytecode 140
  E.1 The Instructions ...................................... 140

F Mozart Virtual Machine Instructions 168
  F.1 The MVM Instruction Set ............................... 168
# List of Figures

3.1 The problem that occurs without global constraints. The regular constraint propagation cannot see that the problem is not satisfiable. .................................................. 25

4.1 Situation in the MVM where the variable A is bound to B, which is bound to C, bound to D. ......................... 37
4.2 The same situation as in 4.1, but with D bound to the value 'aValue'. .................................................. 37
4.3 The MVM variable system, showing the bindings of several variables referenced via different register sets. .......................... 38
4.4 An illustration of how the record \texttt{bar(a:X foo:sub(b:Y) - b:third)} is stored in the MVM variable system. The round boxes represent variables that must be present for the example to work — the '~' character represents the variable that references the \texttt{bar} record, if any. The "headlines" of the big boxes ("record", "var" and "atom") is a tag for the type of data that is stored in the data structure node. ..................... 41

5.1 The liveliness graph \{(1,2) (2,4) (1,4)\}. Variable 1 and 2 are live at the same time, as are variable 2 and 4 and variable 1 and 4. .................................................. 48
5.2 The Oz \texttt{if}-construction showing the different paths forking from it. .................................................. 50
5.3 Inconsistent liveliness for the Oz \texttt{if} construct. The problem is to decide which registers have been born in all paths joining at \texttt{fi}. .................................................. 52

6.1 The dependency graph \(G = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_4)\})\). Variable \(v_1\) and \(v_2\) are live at the same time, as are \(v_1\) and \(v_3\) and \(v_1\) and \(v_4\), and \(v_2\) and \(v_4\) ...................... 57
6.2 A graph coloring problem. ........................................ 58
6.3 Graph coloring problem showing the clique separator \(CS = \{v_3, v_6, v_7\}\). .................................................. 59
6.4 Subgraphs constructed from $S_1 = \{v_1, v_2, v_8, v_9\}$, $S_2 = \{v_4, v_5\}$ and $CS = \{v_3, v_6, v_7\}$ with 3-coloring. .......... 59
6.5 The complete 3-coloring for the graph. .......... 59

8.1 The design of the old back-end. ............ 71
8.2 The structure of the new back-end. ............ 72
8.3 Structure example in abstract instructions .......... 73
8.4 Example of continuation in the abstract instruction set, showing the abstract instructions \texttt{aTestBool} and \texttt{aShared}. .... 74
8.5 Example of continuation in the virtual instruction set, showing \texttt{vTestBool} and \texttt{vShared}. ............ 75

9.1 Object graph for the factorial function. ............ 79

11.1 Register allocation of call instructions. ............ 87

12.1 The structure of the \texttt{aTestBool} abstract instruction, showing the three different paths forking from it. ............ 90
12.2 The structure of the \texttt{testBool} bytecode instruction as it looks after translation from abstract instruction to bytecode. .... 91
List of Tables

3.1 Core Oz statement grammar. Capital characters represent variables, except $S$ which represent statements. ............. 11
3.2 Unification of two variables. ....................... 13
3.3 Syntax of the $\gamma$-calculus ......................... 31
3.4 Structural congruence of the $\gamma$-calculus .......... 32
3.5 Reduction in the $\gamma$-calculus ..................... 33
Chapter 1

Introduction

This thesis describes the effort done to re-design, re- implement and document a compiler for the language Oz, a concurrent multi-paradigm language with object-orientation, higher-order functions, constraints, and much more. The thesis is split into four parts, describing the language Oz, relevant compiler theory, the design and implementation of the compiler, and various appendices containing low-level documentation of the compiler.

In part one we describe mainly the language Oz. However, we start off by explaining the goals of the project, and we summarize what we have achieved. Some of the shortcomings of our compiler are also discussed. The underlying calculus, the $\gamma$-calculus, is also defined in part one. The different programming paradigms possible in Oz are introduced with descriptions and examples. The basic concepts of Oz programming is explained, and the virtual machine used in the current state-of-the-art implementation of Oz is discussed.

In part two, we continue with an overview of some compiler theory with relevance to the thesis project. We describe liveness analysis in a primitive form that we have used, and discuss some of the problems that emerge when doing liveness analysis. The important area of register allocation is discussed, and we describe the basic concepts of register allocation via graph coloring. Some different approaches to register allocation are also described briefly. Different optimization techniques related to a language like Oz are described.

Part three presents the design and implementation notes for the new compiler. An overview of the new design is presented and compared to the old design. The different phases of the compiler and the new data flow inside the compiler is introduced. An overview of the front-end of the compiler is given, and the intermediate code formats used in the new compiler are presented and discussed. The transformation stages in the compiler are also described. Practical liveness analysis and register allocation techniques of interest are discussed, and selected algorithms are presented. The code
CHAPTER 1. INTRODUCTION

emission and assembling phases of the compiler is described.

The appendix contains documentation describing the compiler workings in-depth. The different code formats used internally in the compiler and the interface between the phases of the compiler are specified. Also, the high-level bytecode of the Mozart virtual machine is presented, together with the low-level machine instructions for the virtual machine.
Chapter 2

Project Description

2.1 Project Description

The following text is a description of the original goals and the purpose of the masters’ thesis project. Some clarifying notes and a summary of the goals achieved are given. The goals at the end of the project was mostly the same as in the beginning.

2.1.1 Introduction

*Mozart* is a distributed software platform, that cleanly separates important design aspects during program development. The underlying programming language is *Oz*, a multi-paradigm programming language that uniformly integrates concurrency, higher-order functions, objects and constraints.

The current Mozart compiler is working very efficiently, but a new compiler design was required because;

- The original compiler was not designed in a modular way,
- Documentation of the compilers’ inner workings was virtually inexistent,
- The knowledge of the compiler were not to be found at SICS, were the compiler probably will be developed in the future.

2.1.2 Purpose

The purpose of the thesis work was primary to redesign the existing Mozart compiler and if there was time, to investigate possible extensions of analysis and optimizations.
2.1.3 Goals

- Understanding the existing compiler.
- Redesign of the compiler in an open and modular way. The goal was to make it easy to extend the statical analysis, change the register allocator, etcetera.
- Implementation of the new compiler. A large part the original implementation could be reused.

In general, the design and documentation should give a good description of how the compiler is working, and how to extend it.

The goal of the thesis project was to produce a functioning compiler with the new structure. Fast compilation and efficient code was desirable, but not required.

2.2 Clarifying Notes

The above requirements meant in practice that the back-end of the compiler had to be redesigned and re-implemented. The intermediate code formats had to be modified and adapted to the modularity of the new compiler, and the different stages had to be reconstructed in a more disjoint way, with interaction tightly controlled with above mentioned intermediate code formats.

2.3 Achievements

- Knowledge gained
  - General compiler design and implementation knowledge
  - Knowledge of the old and new Oz compiler
  - Experience with the programming language Oz
  - Analysis and optimization theory and techniques
  - Knowledge about general virtual machines
  - Prolog virtual machine theory
  - Oz Virtual Machine knowledge
- Design
  - The compiler has been redesigned in a more modular way
  - The compiler has been prepared for extensions, like a new register allocation phase, new analysis phases, separate optimization passes etcetera
• Implementation
  – The new compiler design has been implemented
  – Some bugs still exist
  – The code produced is not as efficient as the code produced by the original compiler

• Documentation
  – The compilers’ intermediate code formats has been specified and documented
  – Interfaces between the different phases has been documented
  – A masters’ thesis has been produced
  – Comments has been added in the source code

2.4 Known Bugs

At the time of writing this, our implementation is far from bug free. Since we have planned to make some major changes to the design of the back-end, see the chapter on future work (Chapter 14) for details, it would be a waste of time to put to much effort in in tracking down every last bug. One major bug that we need to fix though, and anyone that uses the compiler in its current state must be aware of, is that parameter passing during message application does not work at all and hence the object-oriented features of the Oz language can not be used at present time. This is probably a bug in the abstract code generation phase. At the moment its also not possible to generate debugging information with our compiler. This is not really a bug, we simply have not had time to implement it yet. These two issues are not affected by the intended design changes, which mainly have to do with register allocation, and should therefore be attended to as soon as possible.

2.5 Comparison

When comparing our compiler with the original one, there are two ways of regarding the efficiency of our implementation. One can compare how long it takes to compile a given program with the two compilers. This is in our view not so important, as long as the performance is not terribly much worse. Also our intention is to first implement a working compiler and then go back and try to optimize it, and since we plan to change rather much of the current implementation, we have not made any attempt to optimize it yet. At the moment, its therefore really no point in measuring the compilation time with our compiler, but we expect that it even when we are done will
be somewhat slower than the original implementation simply, if for no other reason, because it uses more passes.

A more important aspect is how efficient the generated code is. First one has to decide what type of efficiency that is most important, space- or time-efficiency. Sometimes these types does not conflict with each other and other times they do, inlining for example leads to faster execution times but larger programs. From our viewpoint time efficiency is more important than space efficiency, but not at any cost. This might also change for other implementations of the compiler, in the Ozlite project for example, the Mozart engine will run on embedded and hand-held systems where memory is sparse and hence the compiler needs to optimize the memory consumption rather than the execution time.

Because we have not yet implemented the optimization that we have planned and because we had to remove some of the optimizations from the original implementation we have not made any benchmarks on our compiler. From inspection of the generated code one can easily see that our implementation, as it is at present, generates code that is inferior to that of the original compiler. The size of the absolute bytecode for example, is an average of about 40 percent larger using our compiler. Hopefully this will be remedied by the intended optimizations.
Chapter 3

The Language Oz

This chapter provides a very compact introduction to the language Oz. The language of Oz itself must not be fully understood to understand the design and implementation of the compiler, but some terminology and concepts from this chapter are used later in the thesis. The uninterested reader can skip to the next chapter.

Oz is a multi-paradigm language based on the calculus described at page 30. The main feature of Oz is that it provides a wide range of programming abstractions for the programmer, allowing him/her to quickly develop complex applications robustly, and with minimal friction. Oz integrates several paradigms into one language, merging object-oriented, functional, logic, and constraint approaches seamlessly.

For an excellent tutorial and introduction to Oz, see [1]. An extensive reference to the base environment of Oz can be found in [2]. A thorough description of the multi-paradigm ideas behind Oz can be found in [3].

3.1 Features in Oz

Summarizing, Oz is capable of the following:

- Oz provides features of functional programming, with compositional syntax, first-class procedures (and functions), and lexical scoping. Every Oz entity is in fact first-class, including procedures, threads, classes, methods, and objects.

- Oz has features from the logic and constraint approaches, by providing logic variables, disjunctive constructs, and programmable search strategies.

- Oz allows for object-oriented application development, with stateful entities, abstract data types, classes, objects, and inheritance (both single and multiple).
Oz is a concurrent language, with built-in support for dynamic creation of any number of lightweight processes that can interact elegantly via data-flow synchronization.

Oz supports (in the Mozart system) network-transparent distribution of computations. Multiple Oz sites can establish connection, and then automatically behave like a single Oz computation, sharing variables, objects, classes, and procedures.

3.2 Mozart and Oz Overview

The Mozart system implements Oz 3, the latest version of the Oz family of multi-paradigm languages based on the concurrent constraint model described in Section 3.17. Oz 3 is nearly fully compatible upwards with Oz 2. The main differences are functors, which are a kind of software components described in detail in [4], and futures, described in [5], for improved data-flow behavior. Oz 2 is itself a successor to the original Oz 1 language, which was released in 1995.

The full Mozart package with compiler, documentation, development environment, debugger, profiler, and much more can be found at the official Mozart programming system web-site, http://www.mozart-oz.org. All notes regarding Oz in the following document are to the current version of Oz and the Mozart package.

3.3 Logic Variables

The variables of Oz are logic variables, which differ in a number of ways from variables in, for example, the language C. Variables are held in a store. Once they are introduced into the store they cannot be removed. The rules of lexical scoping do however apply. A new variable with the same name as an existing variable will, if added to the store, shadow the previous variable. Also, when implemented, the variable system will of course remove the variable from the store once it goes out of scope to save space. The variable itself is either bound or unbound, i.e., either it has a value assigned to it or not. Once a variable is bound to a value, this value cannot be changed or removed. Hence it is only possible to add information to the variable store, either a new variable or a value to an existing variable. It is not possible to remove any information. A store that works in this way is said to be monotonic.
3.4 Unification

One of the most basic concepts of a programming language like Oz is the atomic operation of unification. This process is the same as deciding if two terms are the same, and at the same binding variables that make the terms equal. In other words, unification is the atomic operation of equating two expressions. Unification with a bound variable is the same as unification with the term bound to the variable. Three cases can occur:

1. Unification of an unbound variable with a general term \( v \). This operation succeeds with the unbound variable bound to \( v \).

2. Unification of an atom with another atom. This operation succeeds if the two atoms are equivalent.

3. Unification of a record \( r/n \) with a general term \( v \). First, the type of \( v \) is checked. If \( v \) is not a record, the operation fails via an exception. If however \( v \) is an unbound variable, case 1 applies. If \( v/m \) is a record with arity \( m \), the operation also raises an exception if \( n \neq m \), that is if the arity \( n \) of \( r \) and \( m \) of \( v \) is not the same. Also, if the labels of the records \( r \) and \( v \) are not the same, the operation raises an exception.

If this does not apply, we have two records \( r/n \) and \( v/n \) with the same label and the same arity. Now, we recursively try to unify each subterm \( r_i \) and \( v_i \) of \( r \) and \( v \) respectively. If this succeeds for all \( i \in n \), then the unification for \( r \) and \( v \) succeeds. Otherwise an exception is raised.

3.4.1 Examples of Unification

This section gives some examples of unification of general structures, illustrating successful and unsuccessful unification, as well as the problem of a variable occurring in the structure to unify this variable with\(^1\).

**Successful Unification**

Assuming that \( X \) and \( Y \) are unbound, unifying

\[
f(g(X), Y) = f(Y, g(h))
\]

can since the label of the both structures\(^2\) are equal be reduced to unifying

\[
g(X) = Y \quad Y = g(h)
\]

---

\(^1\)Checking this is called doing an “occur-check”, and is often omitted.

\(^2\)A *functor* as described in the literature is almost the same as an Oz *label* of a record, which is the equivalent of a structure. The difference is that this kind of functor has information about the arity of the structure as well. The term “functor” has a completely different meaning in Oz, and therefore we shall only use the term “functor” in this more general section.
The first equation succeeds if and only if \( Y \) is bound to the term \( g(X) \). This reduces the problem to unifying the terms
\[
g(X) = g(h)
\]
since \( Y \) is bound to \( g(X) \). This problem can, since the functors are the same, again be reduced to unification of
\[
X = h
\]
which succeeds if \( X \) is bound to the atom \( h \). Thus, the complete unification succeeds, with \( X = h \) and \( Y = g(X) \):
\[
f(g(X), Y) = f(Y, g(h)), X = h, Y = g(X)
\]
**Unsuccessful Unification**  Again assuming that \( X \) and \( Y \) are unbound, the unification
\[
f(Y, g(Y)) = f(h, g(a))
\]
reduces to
\[
Y = h \quad g(Y) = g(a)
\]
since the functor \( f \) is equal. If \( Y \) is bound to \( h \), the first equation unifies, and we get
\[
g(h) = g(a)
\]
which reduces to
\[
h = a
\]
This equation is *not* unifiable, since the two atoms \( h \) and \( a \) are not the same.

**Infinite Structures in Unification**  The problem of unifying
\[
f(g(X), h(Y)) = f(Y, h(X))
\]
reduces to the two subproblems of unifying the sub-terms:
\[
g(X) = Y \quad h(Y) = h(X)
\]
since the functor \( f(g(X), h(Y)) \) has the same label as the functor \( f(Y, h(X)) \).

Under the condition that \( X \) and \( Y \) are unbound at the start of the unification, the first unification \( g(X) = Y \) can only succeed if \( Y \) is bound to \( g(X) \). Now we have
\[
h(g(X)) = h(X)
\]
which reduces to
\[
g(X) = X
\]
This presents a problem for many languages with logic variables, for example Prolog and Oz. This equation is false, but many systems binds the variable \( X \) to the infinite structure \( g(g(g(\ldots))) \).3

3This is represented in Oz as a self-referencing structure, so there is no risk of running out of memory in this case.
3.5 Core Language

The full Oz language can be regarded as syntactic sugar to a small subset of the full language. This subset is called Kernel Oz, and contains enough functionality to express every program in the language Oz. However, kernel Oz is very inefficient for some of the constructs in the full Oz language. Therefore, a superset of kernel Oz (which is still a subset of the full Oz language!) called the Core language is used in the compiler to express Oz statements. The core language has an extended optimized statement set compared to kernel Oz.

The core language is described below. In this introduction, we shall only describe the core language in detail. The reader is encouraged to explore the complete Oz language by reading the Oz tutorial found at the Mozart programming system web-site. (see [1])

Table 3.1 defines the abstract syntax of a statement $S$ in the core language. Statement sequences are reduced sequentially inside a thread. Values (records, numbers, etc.) are introduced explicitly and can be equated to variables.

<table>
<thead>
<tr>
<th>$S$ :=</th>
<th>$S_1 S_2$</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td></td>
<td>Empty statement</td>
</tr>
<tr>
<td>$X$ =&lt;number&gt;</td>
<td></td>
<td>Numeric unification</td>
</tr>
<tr>
<td>$X$ =&lt;literal&gt;</td>
<td></td>
<td>Literal unification</td>
</tr>
<tr>
<td>$X$ = $(f_1:E_1 \ldots f_n:E_n)$</td>
<td></td>
<td>Record unification</td>
</tr>
<tr>
<td>$X$ = $\langle H T \rangle$</td>
<td></td>
<td>List unification</td>
</tr>
<tr>
<td>$X = Y$</td>
<td></td>
<td>Variable unification</td>
</tr>
<tr>
<td>$X.f$</td>
<td></td>
<td>Feature selection</td>
</tr>
<tr>
<td>{NewName $X$}</td>
<td></td>
<td>Name creation</td>
</tr>
<tr>
<td>local $X_1 \ldots X_n$ in $S$ end</td>
<td>Variable declaration</td>
<td></td>
</tr>
<tr>
<td>proc ${ P \ X_1 \ldots X_n }$ $S$ end</td>
<td>Procedure definition</td>
<td></td>
</tr>
<tr>
<td>${ P \ X_1 \ldots X_n }$</td>
<td>Procedure application</td>
<td></td>
</tr>
<tr>
<td>{NewCell $V \ C$}</td>
<td>Cell creation</td>
<td></td>
</tr>
<tr>
<td>{Exchange $C N O$}</td>
<td>Cell exchange</td>
<td></td>
</tr>
<tr>
<td>if $B$ then $S_1$ else $S_2$ end</td>
<td>Selection</td>
<td></td>
</tr>
<tr>
<td>thread $S$ end</td>
<td>Thread creation</td>
<td></td>
</tr>
<tr>
<td>try $S_1$ catch $X$ then $S_2$ end</td>
<td>Exception handling</td>
<td></td>
</tr>
<tr>
<td>raise $X$ end</td>
<td>Exception raising</td>
<td></td>
</tr>
<tr>
<td>{?Value.byNeed? $X Y$}</td>
<td>Future creation</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Core Oz statement grammar. Capital characters represent variables, except $S$ which represent statements.
Chapter 3. The Language Oz

Composition \( S_1 S_2 \)
Composition of two statements \( S_1 \) and \( S_2 \) is expressed simply as \( S_1 S_2 \). This is the same as first executing \( S_1 \), and then \( S_2 \).

Empty Statement skip
The statement \texttt{skip} does nothing. The semantics of the composition \texttt{skip} \( S_2 \) is the same as the semantics for \( S_2 \).

Numeric Unification \( X = \texttt{<number>} \)
Unification of a variable \( X \) with a number \( n \) fails via an exception if \( X \) is bound to anything but \( n \). The unification succeeds if \( X \) is already bound to \( n \), but no effect occurs. If on the other hand \( X \) is unbound, the unification succeeds and \( X \) is bound to \( n \).

Literal Unification \( X = \texttt{<literal>} \)
Unification of \( X \) with a literal \( a \) behaves exactly as the numeric unification described above.

Record Unification \( X = l(f_1:E_1 \ldots f_n:E_n) \)
Unification of a variable \( X \) with a record \( r = l(f_1:E_1 \ldots f_n:E_n) \) succeeds if \( X \) is unbound, in which case \( X \) is bound to \( r \). Also, if \( X \) is bound to a record \( s = l(f_1:Z_1 \ldots f_n:Z_n) \) with the same label \( l \) and arity \( (\{f_1,f_2,\ldots,f_n\}) \) as \( r \), and if the unification of each subterm \( E_1 = Z_1, E_2 = Z_2, \ldots, E_n = Z_n \) succeeds. Otherwise, the unification fails via an exception.

List Unification \( X = \texttt{'}\texttt{'}(H \ T) \)
List unification is an optimization found in the core language. A list in Oz is actually a record with the label \texttt{'}\texttt{'} and two entries, the head \( H \) and the tail \( T \). The list unification is just a faster way to do the equivalent record unification, and has the same semantics.

Variable Unification \( X = Y \)
Variable unification of two variables \( X \) and \( Y \) is expressed as \( X = Y \). The variables must be declared. The effect of a variable unification depends on the state of the variables \( X \) and \( Y \), and is described in table 3.2.
Feature Selection $X.f$

Feature selection from a variable $X$ with the feature $f$ is expressed as $X.f$. If $X$ is a record a feature $f$ with corresponding entry $E$, then the expression represents the entry $E$. Otherwise, the statement fails via an expression.

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>unbound</td>
<td>unbound</td>
<td>$X$ is bound to $Y$, which is unbound. If either $X$ or $Y$ is bound, both variables will reference the same term.</td>
</tr>
<tr>
<td>unbound</td>
<td>bound to $T2$</td>
<td>$X$ is bound to $Y$, which in turn is bound to $T2$. Referencing $X$ (or $Y$) will result in $T2$.</td>
</tr>
<tr>
<td>bound to $T1$</td>
<td>unbound</td>
<td>$Y$ is bound to $X$, which in turn is bound to $T1$. Referencing $Y$ (or $X$) will result in $T1$.</td>
</tr>
<tr>
<td>bound to $T1$</td>
<td>bound to $T2$</td>
<td>if the terms $T1$ and $T2$ can be unified (see page 3.4 then the unification succeeds with the bindings the unification demands. Otherwise the unification fails with an exception raised.</td>
</tr>
</tbody>
</table>

Table 3.2: Unification of two variables.

Name Creation \{\texttt{NewName $X$}\}

The statement \{\texttt{NewName $X$}\} creates a new, unique entity $N$ called a name, and unifies $X$ with $N$ as described above. Names cannot be inspected, forged, or printed, and are guaranteed to be globally unique. If $X$ is unbound, the thread suspends until $X$ is bound.

Variable Declaration \texttt{local} $X_1$ ... $X_n$ \texttt{in} $S$ \texttt{end}

A variable declaration \texttt{local} $X_1$ ... $X_n$ \texttt{in} $S$ \texttt{end} declares the variables $X_1, X_2, \ldots, X_n$ in the scope of $S$. A new variable is created unbound, and can only be used directly in the scope it is declared for.

Procedure Definition \texttt{proc} \{\texttt{P} $X_1$ ... $X_n$\} $S$ \texttt{end}

Definition of a new procedure \texttt{proc} \{\texttt{P} $X_1$ ... $X_n$\} $S$ \texttt{end} unifies the procedure abstraction defined by the body $S$ to the variable $P$, which is used to apply the procedure. The unification fails with an exception raised if $P$ is bound to something else than the procedure abstraction in question.
Procedure Application \( \{ P \; X_1 \ldots \; X_n \} \)

The statement \( \{ P \; X_1 \ldots \; X_n \} \) applies the procedure abstraction stored in \( P \) to the arguments \( X_1, X_2, \ldots, X_n \). The procedure application fails with an exception raised if \( P \) is not bound to a proper procedure abstraction, or if the arity of the procedure application is not the same as the arity of the procedure abstraction bound by \( P \). If \( P \) is unbound, the thread suspends until \( P \) is bound.

Cell Creation \( \{ \text{NewCell} \; V \; C \} \)

\( \{ \text{NewCell} \; V \; C \} \) creates a new cell with the stored value of \( V \), and unifies the cell with \( C \). A cell is an entity that is stateful, meaning that the value stored in a cell can be changed. The unification raises an exception if \( C \) is already bound.

Cell Exchange \( \{ \text{Exchange} \; C \; N \; O \} \)

The statement \( \{ \text{Exchange} \; C \; N \; O \} \) sets the cell bound to \( C \) to the value \( N \), and unifies \( O \) to the current value of the cell. The unification raises an exception if \( O \) is already bound to something else than the value of the cell. If \( C \) is unbound, the thread suspends until \( C \) is bound.

Selection \( \text{if} \; B \; \text{then} \; S_1 \; \text{else} \; S_2 \; \text{end} \)

A simple form of conditional is provided with the statement \( \text{if} \; B \; \text{then} \; S_1 \; \text{else} \; S_2 \; \text{end} \). If the variable \( B \) is bound to \text{true} then \( S_1 \) is executed. If \( B \) is bound to \text{false}, then \( S_2 \) is executed. The statement fails with an exception if \( B \) is bound to a non-boolean value. If \( B \) is unbound, the thread suspends until \( B \) is bound, and the cases above applies.

Thread Creation \text{thread} \; S_1 \; \text{end} \)

A new concurrent thread is created via the statement \text{thread} \; S_1 \; \text{end}. The new thread runs the statement \( S_1 \), and is terminated afterwards.

Exception Handling \( \text{try} \; S_1 \; \text{catch} \; X \; \text{then} \; S_2 \; \text{end} \)

Exception handling is incorporated via the statement \( \text{try} \; S_1 \; \text{catch} \; X \; \text{then} \; S_2 \; \text{end} \). Execution of this statement is equivalent to running \( S_1 \) if no exception is raised. If however an exception is raised, \( X \) is unified with the exception and \( S_2 \) is executed. The variable \( X \) is implicitly declared in the scope of \( S_2 \).
**Exception Raising** raise X end

An exception is raised by executing `raise X end`, which raises the exception bound to X. An exception can be any type of term.

**Future Creation** `{Value.byNeed' X Y}`

The statement `{Value.byNeed' X Y}` unifies an entity called a `future` with the variable Y. When a thread blocks on Y, the unary procedure X is applied to a new variable Z in a new thread. When the application of X is fully reduced, Z is bound to Y. This rather tricky scheme is used to provide lazy evaluation.

### 3.6 Some Oz Examples

The following program demonstrates how to code the factorial function in Oz.

```
declare
fun {Fac N}
  if N<=0 then
    1
  else
    N*{Fac N-1}
  end
end
```

The `fun` statement is syntactic sugar for a procedure with a last argument, which usually is used as a return value. This program is semantically equivalent with the following:

```
declare
proc {Fac N Result1}
  if N<=0 then
    Result1 = 1
  else
    Result1 = N*{Fac N-1}
  end
end
```

This in turn is syntactic sugar for the following core language program:

```
declare Fac in
proc {Fac N Result1}
  local
```
IfArbiter1
UnnestApply1
in
UnnestApply1 = 0
IfArbiter1 = N <> UnnestApply1
if IfArbiter1 then
  Result1 = 1
else
  local
  UnnestApply2
  UnnestApply3
  UnnestApply4
  in
    UnnestApply4 = 1
    UnnestApply3 = N - UnnestApply4
    {Fac UnnestApply3 UnnestApply2}
    Result1 = N * UnnestApply2
  end
end
end
end

This demonstrates how an Oz program is translated into core language in the front-end of the compiler.

Here is another example of an Oz program, calculating the factorial of N with an accumulator to make the calculations more effective:

fun {Fact N Acc}
  if N=<0 then
    Acc
  else
    {Fact N-1 N*Acc}
  end
end

3.7 Concurrency

Threads are easy to handle in Oz, partially because of the build-in data-flow synchronization that is present. Many of the different constructs of the Oz core language share a property; the thread running them blocks if a required variable is unbound at the moment of execution. This property makes it easy to synchronize multi-threaded programs. A thread in Oz is created via the construct

    thread S end
CHAPTER 3. THE LANGUAGE OZ

Executing this statement will fork a thread that runs concurrently with the current thread. Threads in Oz share the processor time available in fair shares, which means that starvation cannot occur.

This example shows how the concurrency combined with data-flow synchronization works. The declare statement is a construct that declares variables in the scope of the source file. This is useful for small demonstration programs like those in this thesis.

\[
\text{declare X0 X1 X2 X3}
\]

\[
\text{thread}
\]
\[
\text{local}
\]
\[
\text{Y0 Y1 Y2 Y3}
\]
\[
\text{in}
\]
\[
\{\text{Browse [Y0 Y1 Y2 Y3]}}
\]
\[
\text{Y0 = X0 + 1}
\]
\[
\text{Y1 = X1 + Y0}
\]
\[
\text{Y2 = X2 + Y1}
\]
\[
\text{Y3 = X3 + Y2}
\]
\[
\{\text{Browse completed}\}
\]
\[
\end
\]
\[
\end
\]
\[
\{\text{Browse [X0 X1 X2 X3]}}
\]

The line \{Browse [Y0 Y1 Y2 Y3]\} starts the Browser, a Mozart tool that can be used to inspect variables in real-time. The term \[\text{[Y0 Y1 Y2 Y3]}\] is a shortcut for building a list of \'|\'-records, and is compiled as \[\text{'(Y0 '}(Y1 \text{'|}'(Y2 \text{'|}'(Y3 \text{nil)})))\]. Also, the '+' operator is translated into a call to a regular library function that calculates sums. This is however optimized in the compiler.

Running this program will make the Browser showing the variables unbound. The thread is blocked at the first unification instruction, because \text{X0} is unbound. But feeding the following lines into the compiler (which works on-line in parallel to the program) will have the effect of resuming and suspending the thread, one unification instruction at at time.

\[
\text{X0 = 0}
\]
\[
\text{X1 = 1}
\]
\[
\text{X2 = 2}
\]
\[
\text{X3 = 3}
\]

The Browser will show the variables getting bound one at a time. At last, the \text{'completed'} literal will be shown in the Browser when the thread is finished.
Oz is capable of handling tens or even hundreds of thousand threads at the same time. The only limit is the amount of available memory, and Oz creates threads about 60 times faster than Java JDK 1.2. The purpose of threads in Oz is mainly to make the structuring of programs easier. A single threaded program runs almost always faster than a multi-threaded program (except in some special circumstances — concurrent simulation, for example), but the structure can become much more complicated for a singly-threaded program than a multi-threaded one.

3.8 Functional Programming in Oz

Oz extensively supports the functional programming paradigm. In this chapter, we will briefly discuss some functional topics handled by Oz and the Mozart system.

3.8.1 Lazy Evaluation

Lazy evaluation is to delay the calculation of certain values until they are actually needed. Consider the following program:

```oz
local
  List
in
  fun {Natural1 N}
    N | {Natural1 N+1}
  end
  {Show {Natural1 100}.1}
end
```

The line beginning with `fun` is short-hand syntax for defining a procedure with a return argument. The procedure defined actually takes two arguments, with the last one being the implicit return argument. The line `N | {Natural1 N+1}` implicitly unifies the return argument with the list built by taking `N` as the first element, and the list returned from the call to `Natural1` as the rest. A list in Oz is actually a record `'|'(First Rest), where first is accessed by using the dot notation `'.1'`. The statement `N | {Natural1 N+1}` is then syntactic sugar for building a record with the first feature being `N`.

Running this program will force the thread into an infinite loop. This behavior of in this example calculating the "complete" infinite list, although only the first element is needed is called eager evaluation. The example described above can be modified to work by using lazy evaluation:

```oz
local
  List
```
fun lazy {Natural2 N}
    N | {Natural2 N+1}
end
{Show {Natural2 100}.1}
end

In this case, the program will run and stop with the correct answer of 100. This is because the values of the infinite list is not calculated until needed, and because only the first value of the list is needed, only this value is calculated. Lazy evaluation in Oz is implemented using futures, which is described in [5].

### 3.8.2 Higher-order Functions

Oz is capable of handling higher-order functions, that is, functions which can take other functions as arguments. New functions can be created on-the-fly during runtime, and the functions in Oz are stored in regular variables. Also, Oz is dynamically typed, making advanced polymorphic techniques possible.

A good example of the higher-order capabilities of Oz is the Quicksort routine written in Oz:

```oz
1 fun {Quicksort Lst}
2     case Lst of First|Rest then
3         {List.partition Rest
4             fun {$(Entry)} Entry<First end
5                 Left Right}
6         {Append {QuickSort Left} First|{QuickSort Right}}
7         else Lst end
8     end
```

Line (1) tells us that a function with one argument (and one implicit return argument) follows. At line (2), an effort is made to split the argument Lst into two parts, the first element First and the rest of the list Rest. If this is not successful, the function returns the argument given to it at line (7). If the split succeeded however, that is, if the argument was an non-empty list, a library call is made at line (3-5) to a higher-order procedure List.partition. This call takes four arguments. The first is the list to partition (Rest in this example), the second the boolean function used to split the elements of the list. The third and fourth argument of the procedure are bound to the left and the right partitions of the original list.

At line (4), an anonymous function is dynamically created and passed as a parameter to List.partition\(^1\). This anonymous function (with formal

\[^1\text{For the reader familiar with }\lambda\text{-calculus, this particular expression is equivalent with }\lambda\text{Entry}.(\ldots)\]
parameter \textit{Entry} simply tells whether the current entry is less than the pivot element, that is, the first element \textit{First} in the list.

At last at line (6), the \texttt{QuickSort} function is recursively applied to the two partitions, and the results are appended together with the pivot element sandwiched in between them. The complete list is then returned from \texttt{QuickSort}.

### 3.9 Logic Programming

This chapter intends to provide an introduction to the different logic and constraint programming techniques available to the Oz programmer. For more information, see [6].

In order to understand the semantics of the different logic and constraint programming constructs available in Oz, we will begin with a description of the underlying mechanics of the constraint and logic programming models.

\textbf{Compiling Logic and Constraint Programming Code} The logic and constraint programming mechanisms in Oz are completely abstracted from the Core Oz described before, except for the unification procedure which is fundamental to Oz. Therefore, as a compiler designer there is no need to understand and take measures for constraint and logic programming. The following description of the constraint models in Oz can be seen as a general introduction to these topics, and is not needed to understand the compiler design and implementation.

#### 3.9.1 Constraint Stores

Oz threads share a store where variable bindings are stored in the form of equalities. This store is said to be a \textit{constraint store}. An Oz \textit{computation store} consists of a constraint store, a procedure store where procedures reside, and a cell store, where cells and object states reside.

#### 3.9.2 Computation Spaces

An Oz \textit{computation space} consists of a computation store and a set of threads that are executing in the environment of the computation store. In Oz, the programmer has access to multiple nested computation spaces.

The general structure of the computation spaces are

- There is always a topmost computation store, where threads can communicate with the external world. A thread trying to add inconsistent bindings to the store will raise an exception in the thread. The inconsistent bindings will not be added and the store remains consistent at all times.
• A thread may create a local computation space, either explicitly or implicitly. The new computation space will be a child space, and the current one the parent space.

• A thread belongs to exactly one computation space. Also, a variable belongs to only one computation space.

• A thread in a child space sees and may access variables in the child space as well as in all ancestor spaces. The opposite is false, the thread can neither see nor access variables in child spaces to its own computation space.

• A space can be merged with a child space. If this situation occurs, the child space disappears, and all its bindings move to the current space.

• A thread may add bindings to visible variables (variables in the current space or ancestor spaces). These bindings will only be visible in the current space and its children.

3.9.3 Constraint Entailment and Disentailment

A condition C is entailed by the store σ if C is logically implied by σ (where σ is considered as a logic formula). This means that adding C to the store σ does not add any information. A condition C is disentailed by the store σ if ¬C is logically implied by σ. This means that C is inconsistent with the store σ.

Since σ can be considered as a logic formula containing all equalities of the store, we can also talk about a constraint store σ₀ being entailed or disentailed by σ₁. Also, a space S₀ is entailed (disentailed) by another space S₁ if the constraint store σ₀ of S₀ is entailed (disentailed) by σ₁, the constraint store of S₁. A space that is disentailed (normally by a parent space) is called a failed space.

3.9.4 Deterministic Logic Programming

Deterministic logic programming, i.e. sequential logic programming without search, is provided with the constructs if, case and cond. The if selection statement has already been described. case and cond is however a bit different.

The cond-construct A logical conditional is a statement

\[
\text{cond } X_1 \ldots X_n \text{ in } S_0 \text{ then } S_1 \text{ else } S_2 \text{ end}
\]

where \(X_i\) are newly introduced variables, and \(S_i\) are statements. The semantics of cond are
• The current thread $T_0$ in the current space ($SP_0$) is blocked.

• A space $SP_1$ is created with a single thread running $S0$ (with the newly created variables $X_i$).

• $T_0$ remains blocked until $SP_1$ is either entailed or disentailed. This condition may never occur, for example when some thread suspends or loops in $SP_1$.

• If $SP_1$ is disentailed, $T_0$ continues with $S_2$.

• If $SP_1$ is entailed, $SP_0$ and $SP_1$ are merged and $T_0$ continues with $S_1$.

The case-construct  The case statement has the form

```plaintext
case V of E_1 then S_1
  elseif E_2 then S_2
  ...
  else S end
```

where $E_i$ are patterns. A pattern is a term, where the variables are declare implicitly. If a variable should not be implicitly declared, the variable should be preceded with ‘!’ . A syntactic abbreviation for elseif is [].

The semantics of the case construct is as follows;

• The unification $V = E_i$ is tried for each $E_i$ in a top to bottom matter.

• If the unification succeeds for $E_i$ without binding any variable occurrence in $V$, $S_i$ is executed.

• If the unification succeeds for $E_i$ but binds some variables in $V$, the thread suspends.

• If the unification $V = E_i$ does not succeed, the next pattern $E_{i+1}$ is tried. If $E_i$ is the last pattern, the else statement $S$ is executed.

The else part may be omitted, in which case an exception is raised if all patterns fail to unify.

For example, to append two lists into a third, one could write:

```plaintext
declare
proc {Append L1 L2 L3}
  case L1 of nil then L2=L3
    [] X|M1 then L3=X|{Append M1 L2}
  end
end
```
This procedure works whenever \texttt{L1} is instantiated when the call to \texttt{Append} is made. If this is not the case, the \texttt{case}-statement will block, waiting for \texttt{L1} to be bound before continuing. If we would be interested in a more general append, like the one in Prolog, we could have to introduce search into our program. We do not explain the concepts of search in logic programming in this thesis, but instead explain the constraint programming model in detail.

### 3.10 Constraint Programming

Constraint programming can be seen as an extension of the logic programming model. The difference is that in a constraint programming model, the search is limited by sets of \textit{constraints}, making the search space much smaller.

#### 3.10.1 Domains

The constraint programming paradigm has been applied to several domains, of which the \textit{finite domain} has by far been most successful. Finite domain constraint programming (or CLP-FD, for Constraint Logic Programming – Finite Domain) is concerned with finding values in a finite discrete set for a set of variables. In CLP-FD, the variables has values from a finite set (often integers from e.g. 0 to some machine-dependent constant).

Other types of domains that constraint programming has been applied to includes real numbers and rational numbers. In this chapter we will only look at CLP-FD.

#### 3.10.2 Constraint Propagation

To effectively prune the search space, constraint propagation is used. The propagation process involves the \textit{constraint space}, where the constraints interact with each other. The interaction between constraints and variables in the constraint space can be seen as a cauldron where different chemicals (constraints) react with each other, limiting the variables involved.

To add a new constraint to the constraint space is often referred to \textit{injecting} a constraint into the space.

**Interval Propagation**

Interval propagation means that only the \textit{interval} of a variable is kept in memory, effectively two values, the beginning and the end of the interval. Interval propagation therefore means less memory consumption, but often implies that the propagation of constraints is not as effective as one could wish.
Domain Propagation

Domain propagation means that the discrete values the value can take are all kept in memory. This often means better propagation, but takes a lot more memory — imagine that the value of $x$ is known to be in the interval $[0,10000]$. By using domain propagation, 10000 different values has to be kept in memory for $x$ only, as opposed to 2 values for interval propagation.

**Example:** Suppose we have the following CLP-FD model, using domain propagation:

$$X \in \{0..1000\} \quad (3.1)$$
$$Y \in \{0..1000\} \quad (3.2)$$
$$2 \times X = Y \quad (3.3)$$
$$X + Y = 1000 \quad (3.4)$$

From equation (6.1), (6.2) and (6.3), the conclusion $X \in \{0..500\}$ and $Y \in \{0,2,4,\ldots,1000\}$ can be drawn. Now, from this conclusion and (6.4), we can conclude $X \in \{0,2,4,\ldots,500\}$ and $Y \in \{500,502,504,\ldots,1000\}$. The solutions to the problem can be found by injecting constraints binding $X$ and/or $Y$ (see “search”). For example, the injection of the constraint $X = 42$ gives the solution $X = 42,Y = 958$.

If interval propagation is used instead, the only conclusion we can make is $X \in \{0..500\}$ and $Y \in \{500..1000\}$. Thus we have to search twice the number of possible solutions using domain propagation, but the memory consumption is decreased from 500 integers to 4 integers.

3.10.3 Search

When the constraints imposed by the constraint model (the program) cannot interact to further prune the domains of the variables, search has to be the next step to get possible values for the variables. Search is implemented as injection of constraints binding variables to values into the constraint space.

3.10.4 Atomic Constraints

Atomic constraints are regular constraints imposed on a set of variables.

**Example:** $10 \times A + 2 \times B =: C$ Atomic constraints were the first constraint type developed for CLP. There exist standard techniques for constraint propagation of atomic constraints.
3.10.5 Reified Constraints

Reification of a constraint is to assign to a certain constraint a truth value. Now, the constraint does not have to be fulfilled, but if it is not, its negation must hold. This opens new possibilities of modeling relations between constraints. The following example states that either $10A + 2B = C$ or $A + B = C$ must hold.

$$(10A + 2B = :C) \lor (A + B = :C)$$

3.10.6 Global Constraints

Global constraints are specialized constraints handling all the variables in the model at the same time. The first global constraint was created to handle a situation like the one in figure 3.1. The variables $A$, $B$ and $C$ have the domains $[1,2]$, and the constraints

$$
\begin{align*}
A &\neq B \\
B &\neq C \\
C &\neq A
\end{align*}
$$

are imposed on the variables. Now, the constraint propagation is unable to see that the problem is unsatisfiable! Search has to be performed to actually test the values of the variables before unsatisfiability can be concluded.

![Figure 3.1](image.png)

Figure 3.1: The problem that occurs without global constraints. The regular constraint propagation cannot see that the problem is not satisfiable.

To improve this situation, the first global constraint was invented. The constraint declares that all variables must have disjunct values. In Oz, this is expressed as:

$${Fd\text{.distinct Root}}$$

where $\text{Root}$ holds the variables that must be distinct.

Now, the constraint propagation can determine unsatisfiability at once, making search unnecessary in this situation.
3.10.7 Constraint Programming Example

For an example of a constraint problem (and its formalization in Oz) take the classical Send More Money puzzle.

Problem Specification

Find distinct digits for the letters $D, E, M, N, O, R, S, Y$ such that the equation

\[
\begin{array}{cccc}
S & E & N & D \\
\hline
M & O & R & E \\
\end{array}
\]

\[
= \begin{array}{cccc}
M & O & N & E & Y \\
\end{array}
\]

is satisfied.

simplification A consequence of the problem formulation is that neither $S$ nor $M$ can be zero, because then $O$ would be the same as the one of \{S, M\} which was not zero. This violates the constraint that the digits were to be disjunctive.

Model

We model the problem by having a variable for every letter, where the variable stands for the digit associated with the letter. The constraints are obvious from the problem specification.

Solution

```
declare

proc {Money Root}
    S E N D M O R Y
  in
    Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)
    Root ::: 0#9
    {FD.distinct Root}
    S \=: 0
    M \=: 0
    1000*S + 100*E + 10*N + D
    + 1000*M + 100*O + 10*R + E
    =: 10000*M + 1000*O + 100*N + 10*E + Y

    {FD.distribute ff Root}
  end
```
When this program is called with the Oz line

\{ExploreAll Money\}

the unique solution $9567 + 1085 = 10652$ can be seen.

### 3.11 Object Orientation is Syntactic Sugar in Oz

Oz allows the programmer to take full advantage of the object-oriented paradigm. This support can be regarded as syntactic sugar though, because all object oriented features could be expanded into core language. For efficiency reasons, some features of object orientation are optimized with specialized instructions in the virtual machine.

This section gives a short introduction to object orientation in Oz. For more information see [1].

### 3.12 Classes

A class in Oz is a description of how objects instantiated from this class should behave. It is a stateless record with the following components:

**Methods.** If the method name is a variable (starting with an uppercase letter) then the corresponding method will be private to that class, i.e. it can only be called from within another method in the same class. The arguments may have default values. All methods are kept in a table. When a message is applied to an object, that message is used to index into the method table and the corresponding method is executed.

**Features.** A feature is an ordinary Oz variable. They are accessed by a feature-name, which is either an atom or an Oz name. If the feature is initialized to a variable in the class declaration, all objects of that class will share the same variable.

**Attributes.** An attribute defined in a class can only be manipulated in objects instantiated out of that class or descendants to it, i.e. they are private to that object. At object creation, all attributes are assigned a conceptual stateful cell. Ordinary cells are manipulated via the \texttt{Assign} and \texttt{Access} procedures, but for cells assigned to attributes the short hand operators $<- \text{ and } @$ are used for assigning and accessing cells.

### 3.13 Objects

As classes, objects are also essentially records. They contain a reference to the class of the object. The state of the object is held by a cell, which
is hidden by lexical scoping. When a message is applied to an object, the corresponding method is handed a reference to the cell and can thus modify the state of that particular object.

### 3.14 Private and Protected Methods

As mentioned before, methods labeled by variables are private to their class. In many object oriented programming languages, there is the option of defining protected methods, meaning that they are only visible inside their base class and in descendants of that class. To achieve this in Oz, we have to use a little trick. A method that is defined as private can be assigned to an attribute, which is protected, and descending classes then can use that attribute to execute the method.

### 3.15 Inheritance

When one class inherits from another, there may be conflicting labels in them, i.e. the same label of a method, feature or attribute might occur in more than one class. If single inheritance is used, the label in the superclass will simply be shadowed by its descendent. If a class on the other hand inherits from two or more classes (multiple inheritance), there might be conflicts that can not be resolved. For example, say that we have a class that inherits two classes and that the same label occurs in both of them (or a super class of them). In this case there is nothing to do exception end the compilation with an error.

### 3.16 Overhead

Applying a message to an object generates some overhead compared to ordinary procedure application. The class of the object has to be dereferenced, the method corresponding to the message has to be looked up in the method table and the objects itself must be passed as an argument to the method. The remedy to this is the method call. Inside a class C, a method m(...) may be called in the following manner: C, m(...). This is called a static method call, and generates no extra overhead. The reason that method calls can only be used inside a class is that they take the current object as an implicit argument.

### 3.17 Example

This example demonstrates a class structure describing a car model:
class Car
    feat
        engine
        transmission
    attr
        price % US dollars
        acceleration % 0-100 km/h in seconds
meth init(P)
    price <- P
end

meth buy()
    {System.showInfo "Withdraw "#@price#" dollars from your account."}
end
end

class Lamborghini from Car
    meth init()
        price <- high
end

meth buy()
    {System.showInfo "First choose a model!"}
end
end

class Engine
    feat
        cylinders
        power % kilowatts (kW)
        volume % cubic centimeters (cc)
        torque % newtonmeters (Nm)
end

meth init(C P V T A)
    self.cylinder = C
    self.power = P
    self.volume = V
    self.torque = T
end
end
CHAPTER 3. THE LANGUAGE OZ

```plaintext
class DiabloVT2000 from Lamborghini
meth init()
  self.engine          = {New Engine init(12 405 5992 620)}
  self.transmission    = {New Transmission init(5)}  % Object not shown here
  price <- 455000  % dollars
  acceleration <- 4  % seconds
end
meth buy()
  Car, buy()
  {System.showInfo "You lucky bastard."}
end
```

3.18 The $\gamma$-calculus

The $\gamma$-calculus upon which Oz is based is introduced in [7]. The material presented here is a short summary and an introduction to the calculus that is the basis for Oz. It has no impact on the design of the compiler and can therefore be skipped by the uninterested. Most of the material in the section is taken from [7].

3.18.1 Introduction

The $\gamma$-calculus is a formal computational calculus for higher-order concurrent programming, and is the basis of the Oz programming language. With $\gamma$-calculus, it is possible to express both the eager and the lazy $\lambda$-calculus, as well as higher-order constraints and object hierarchies. The primitives of the $\gamma$-calculus are logic variables and names (which has been previously described), procedural abstractions, and cells. Cells provide an expressions of state that is fully compatible with concurrency and constraints. The $\gamma$-calculus is closely related to the $\pi$-calculus ([8, 9, 10]) as described by Schulte in [7].

The $\gamma$-calculus can be extended in various ways, for example with primitives providing for constraint-based problem solving in the style of logic programming, and with primitives that allow for lazy evaluation and embedding of both the eager and the lazy $\lambda$-calculus. A such extended $\gamma$-calculus combines first-order constraints with higher-order programming. This extension is described in [7], where Smolka also extends the $\gamma$-calculus with general first-order constraints and search, and shows how to express communication, functions with embedded state and objects in $\gamma$-calculus.
3.18.2 Syntax of the $\gamma$-calculus

<table>
<thead>
<tr>
<th>Symbols</th>
<th>$x, y, z$ : variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a, b, c$ : names</td>
</tr>
<tr>
<td></td>
<td>$u, v, w$ ::= $x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expressions</th>
<th>$E, F, G$ ::= $\top$ null</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E \land F$ composition</td>
</tr>
<tr>
<td></td>
<td>$\exists u E$ declaration</td>
</tr>
<tr>
<td></td>
<td>$u = v$ equation</td>
</tr>
<tr>
<td></td>
<td>$a : \exists / E$ abstraction (\exists linear)</td>
</tr>
<tr>
<td></td>
<td>$u \exists$ application</td>
</tr>
<tr>
<td></td>
<td>if $u = v$ then $E$ else $F$ conditional</td>
</tr>
<tr>
<td></td>
<td>$a : u$ cell</td>
</tr>
</tbody>
</table>

Table 3.3: Syntax of the $\gamma$-calculus

The syntax of the $\gamma$-calculus is given in table 3.3. An infinite alphabet of variables and a infinite set of (unique) names must exist. Variables and names are called references. Variables are place-holders for names, which stand for themselves and thus are semantically different if they are syntactically different. There are no other values than names in the $\gamma$-calculus.

The expressions given in $\gamma$-calculus are relational like in logic programming or the $\pi$-calculus. Composition works like conjunction in logic programming and parallel composition in $\pi$-calculus. Declarations $\exists u E$ introduces a new reference $u$ with the scope $E$. Equations (or unification) are like equations in logic programming.

An abstraction $a : \exists E$ consists of a name $a$ for the abstraction, formal arguments $\exists$ (where $\exists$ means a possibly empty sequence of variables), and an abstraction body $E$. $\exists$ must consist of pairwise distinct variables.

An application $u \exists$ applies the abstraction referenced to by $u$ to the actual arguments $\exists$. A conditional if $u = v$ then $E$ else $F$ reduces to either $E$ or $F$ depending on whether $u$ and $v$ turn out to be equal. A cell $a : u$ has the name $a$ and the reference $u$; reduction with an application $avw$ will impose the equation $u = v$ and update the cell to hold $w$.

3.18.3 Semantics of the $\gamma$-calculus

The expressiveness of the above syntax are formalized by rules rewriting the expressions. This setup is also found in the $\lambda$-calculus (functional computation) and SLD-resolution (relational computation). For the $\gamma$-calculus the rules are applied modulo a structural congruence, and can only be applied
to specific positions.

The structural congruence of the $\gamma$-calculus is given in table 3.4.

<table>
<thead>
<tr>
<th>The structural congruence of the $\gamma$-calculus is the least congruence “$E \equiv F$” on the set of expressions satisfying the following laws:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Composition $E \land F$ of expressions is associative, commutative, and satisfies $E \land \top \equiv E$</td>
</tr>
<tr>
<td>2. Declaration $\exists u E$ of references allows for consistent renaming of the declared reference $u$ and satisfies</td>
</tr>
<tr>
<td>$\exists u E \land F \equiv \exists u (E \land F)$ if $u$ not free in $F$</td>
</tr>
<tr>
<td>$\exists u \exists v E \equiv \exists v \exists u E$</td>
</tr>
<tr>
<td>$\exists u \top \equiv \top$</td>
</tr>
<tr>
<td>3. Abstractions $a : \vec{x}/E$ allow for consistent renaming of the formal arguments $\vec{x}$</td>
</tr>
<tr>
<td>4. Equations $u = v$ are symmetric.</td>
</tr>
</tbody>
</table>

| Table 3.4: Structural congruence of the $\gamma$-calculus |

Reduction in the $\gamma$-calculus is specified in table 3.5 by a system of inference rules. Only the three structure rules have premises, the others are axioms.

The structure rules say that reduction is modulo structural congruence, and that reductions of subexpressions not appearing beneath abstractions and conditionals can be taken as reductions of the entire expression. A reduction $E \to F$ is possible if and only if it can be derived with the structure rules from exactly one instance of an axiom. The application rule comes with the side condition that the number $|\vec{x}|$ of actual arguments in the application equals the number $|\vec{z}|$ of formal arguments in the abstraction.

**Closures** When reducing a $\gamma$-expression using the application or the elimination rule, the closure of the expression changes, as can be seen in table 3.5 where the new closure is enclosed in brackets. The closure of an expression contains substitutions for certain references.

**Proposition 3.1** Let contexts be defined as $C ::= \bullet | C \land E | E \land C | \exists u C$. Then $E \to E'$ is a reduction in the $\gamma$-calculus if and only if there exists a context $C$ and an instance $G \to G'$ of an axiom in Table 3.5 such that $E \equiv C[G]$ and $C[G'] \equiv E'$. 


3.18.4 Example: Creating a New Name

The expression \( \exists a(x = a) \) has the meaning; “Create a new name and equate the variable \( x \) with it”. To see why, consider the expression

\[
\exists x \exists y(\exists a(x = a) \land \exists a(y = a) \land \text{if } x = y \text{ then } E \text{ else } F)
\]

and suppose that \( x \) and \( y \) are distinct variables that does not occur free in \( E \) and \( F \). Moreover, assume that \( a \) is a name not occurring free in \( E \) and \( F \). We will show that the expression reduces to \( F \).

First, using the first and second laws of structural congruence (table 3.4) we move the left declaration of \( a \) outside the expression.

\[
\equiv \exists a \exists x \exists y(x = a \land \exists a(y = a) \land \text{if } x = y \text{ then } E \text{ else } F)
\]

Exchanging the declarations of \( x \) and \( y \) with the second law, and eliminating \( x \) with the Elimination Rule (table 3.5), we get the following:

\[
\rightarrow \exists a \exists y(\exists a(y = a) \land \text{if } a = y \text{ then } E \text{ else } F)
\]
CHAPTER 3. THE LANGUAGE OZ

Next, we rename the inner name $a$ to $b$, where $b$ is assumed to be different and not to occur free in $E$ and $F$.

$$\equiv \exists a \exists y (\exists b(y = b) \land \text{if } a = y \text{ then } E \text{ else } F)$$

Now we can eliminate $y$ in the same way as for $x$:

$$\rightarrow \exists a \exists b (\text{if } a = b \text{ then } E \text{ else } F)$$

Since $a$ and $b$ are different we get:

$$\rightarrow \exists a \exists b F$$

using the appropriate Conditional Rule. Remaining now is to remove the declarations of $a$ and $b$, which can be done using the congruence laws:

$$\equiv \exists a \exists b (\top \land F)$$
$$\equiv (\exists a \exists b \top) \land F$$
$$\equiv \top \land F$$
$$\equiv F.$$
Chapter 4

The Mozart Virtual Machine

The Mozart Virtual Machine, MVM, is a modern register-based virtual machine, with high-level\(^1\) bytecode instructions and built-in garbage collection routines that works without any interference from the compiler. The virtual machine is built in such a way, that the compiler designers do not have to worry about many special details about the language, such as the Constraint Solving system and the lazy evaluation mechanism.

This chapter intends only to provide an introductory overview. In [11, 12], a virtual machine called LVM\(^2\) is described in detail. Ait-Kaci provides a tutorial on the Prolog virtual machine in [13], which conforms in many aspects with the LVM.

In the rest of the thesis, if not otherwise stated when talking about variables, the mentioned variables are regular Oz variables, as described in this chapter.

4.1 Registers in MVM

As explained, MVM is a register-based virtual machine, as opposed to a stack-based one. All data are accessed via registers, even procedures.

The different register types in Oz are:

**X-registers** which are pretty much the same as “normal” registers, except that there is more of them. These registers are volatile, and can be altered anywhere in the compiled program. Oz also utilizes these registers for parameter passing, where the current constraint is that the parameters for a procedure \(p/n\) with arity \(n\) are passed in \(x_0, x_1 \ldots x_{n-1}\).

Built in procedures work in a similar way; the parameters are passed in X-registers, but the compiler can choose which registers will be used. The built-in is therefore passed a list of X-registers used as parameters.

\(^1\)In comparison with regular assembly languages
\(^2\)As in “The L Virtual Machine”, where L is a subset of Oz (like the Core Oz)
As default, MVM has a store of 4096 X-registers. The X-registers are stored in regular stack memory, and the number of registers are static — the only way of changing the number of X-registers available is to recompile the MVM.

**Y-registers** are equivalent to stack variables, as used in imperative languages like C. They are allocated dynamically when needed. As said, the X-registers are volatile, and can be changed anywhere in the program, and to save needed information stored in a X-register, the Y-register set is used. The Y-register are local to the current declaration, where the number of needed registers are allocated explicitly. For example, a definition may allocate \( y_0, y_1 \) and \( y_2 \) as its available Y-registers.

The tail recursion optimization featured in the Oz compiler often reduces the need of Y-registers.

**G-registers** makes it possible to reach data declared in an outer scope, i.e. the code in a procedure can reach variables declared in outer declarations through the G-registers. The G-registers makes up the *closure* of the procedure, as described in Chapter 3.

### 4.2 The MVM Variable System

The variable system of MVM is centralized around the notion of variables as references. A variable does not contain a value itself, instead it is *bound* (or unified) to a value, as seen in Figure 4.3. In the following text, variable names are used to illustrate the concept instead of physical registers. Replacing the variable names with physical registers shows how the variable system works in real life. Also, it does not matter for the variable system which register is used for referencing a variable.

When creating a new variable, a new “value-space” is created, and a free register is bound to that location. The “value-space” itself is unbound. When binding an existing variables to another, like in \( A = B \), the references are simply followed until a value-node is found. That is, the chain

```
local A B C D in
  D = 'iAmAValue'
  C = D
  B = C
  A = B
end
```

has the effect of binding all variables to the value *'iAmAValue'* , whereas the following code

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

instead binds first A to the value of B (which is unbound), then binds B to
the value of C (unbound), then C to the unbound value of D. This results in
a situation like in figure 4.1.

![Figure 4.1: Situation in the MVM where the variable A is bound to B, which
is bound to C, bound to D.](image)

Now, binding D to a value, like in D = 'aValue', results in the situation
shown in figure 4.2.

![Figure 4.2: The same situation as in 4.1, but with D bound to the value
'aValue'.](image)

In figure 4.3, the MVM variable system is illustrated with the variables
depicted as physical registers instead.
Figure 4.3: The MVM variable system, showing the bindings of several variables referenced via different register sets.

### 4.3 Instruction Set

The code format of the MVM is called absolute bytecode. The instruction set consists of a little more than 150 instructions, which can be divided into the following categories:

- Defining procedures
- Moving data around
- Creating variables
- Allocation of new data structures
- Matching and unification of data structures
- Allocation of local environments
- Applying built-ins
- Applying procedures and calling methods
- Control flow
CHAPTER 4. THE MOZART VIRTUAL MACHINE

- Conditionals
- Debug instructions
- Miscellaneous

The complete instruction set can be found in Appendix F.

Since this code format is unnecessarily explicit for the peephole optimization-and absolute addressing-phase, a more generic code format called relative bytecode (see Appendix E) is used there. In absolute bytecode there are, for example, different move instruction for different types of source- and destination-registers (moveXX, moveXY, moveYX and so on), whereas there is only one move instruction in relative bytecode, which instead supplies the register types as parameters. Whenever there occur examples with bytecode in the rest of this thesis, it will be in the form of relative bytecode.

Many of the instruction categories have one general instruction that can be used for all legal types, or values, of the parameters, and several alternative instructions optimized for specific parameter types/values. The unify instruction, for example, has a number of optimized variants, unifyNumber, unifyLiteral and so on.

There are also optimizing instructions used to substitute two, or in some cases three, other instructions. They usually combine the creation and initialization of variables/registers, but there are also instructions that perform two register moves in one single instruction. These instructions are usually introduced in the peephole optimization phase.

To give a small example of relative bytecode, we transform the following Oz procedure:

```oz
proc {Foo Bar X}
  {Bar foo}
  X=foo
end
```

This gives the code in the following listing:

```oz
1   1bl(7)
2      definition(x(0) 30 pid('Foo' 2 pos('Oz' 2 0) nil 2)
3        <P: 1> nil)
4      allocateL1
5      move(x(1) y(0))
6      sendMessage(foo x(0) 0 cache)
7      getLiteral(foo y(0))
8      deAllocateL1
9      return
10     endDefinition(7)
11    1bl(30)
```
The instructions in line 1 and 11 are symbolic labels and have no counterpart in absolute bytecode. The definition instruction marks the beginning of a new procedure abstraction and supplies the necessary information in its argument (see the appendix for information about the arguments). In line 4, a local environment of size 1 is allocated, i.e., register y(0) is allocated. Since we are about to make a procedure call (to the procedure that is supplied as the first argument and held by the variable Bar), we must save all X-registers that are supposed to survive the call, in Y-registers. In this case there is only one, x(1), which holds the reference to the formal parameter x. The move instruction in line 5 accomplishes this saving. The call is made in line 6. sendMsg is an optimized call instruction that sends the literal foo to the procedure held by register x(0). Next, the literal foo is assigned to the variable X, which is now referenced by y(0), since we cannot be sure that x(1) wasn’t corrupted be the call to the procedure Bar. Finally the local environment is deallocated followed by a return statement and the instruction that marks the end of the procedure body (which will never be executed).

4.4 Unification in the MVM

The unification process is described in detail in page 9. The Mozart virtual machine has special instructions for unification of variables with atoms, with records, and with variables, which makes unification very fast. The unification operation can be separated in three different cases:

Case 1: Unification of a variable with an atom is a straightforward matter of checking the identity of the atom with the type and the value of the variable.

Case 2: Unification of a variable with a record is a little more complicated, as the record to unify with has to be built (at least in part) in memory before the unification can proceed. This process is described later in this chapter.

Case 3: Unification between variables is handled completely by the virtual machine. This operation works as described in Section 3.4.

The variable system of the MVM has already been described. What has not been told is how structures (which are called records in Oz) are stored in the variable system.
4.4.1 Record Representation in MVM

A Oz record is a structure on the form $l(f_1 : T_1, f_2 : T_2, \ldots, f_n : T_n)$ where $l$ is the label of the record\(^3\) which is an atom, and $f_i$ where $i \in \{1 \ldots n\}$ are the features of the record. The features of a record are used to access the fields of the record, where a field $T_i$ is a general term — a reference to a variable, an atom, or a structure. The features themselves are atoms.

If the features of a record is not specified, they are by convention enumerated as $1, 2, \ldots, n$, so that the specification of a record $\text{foo}(\text{bar}(Y) \ \text{a})$ implicitly is defined as $\text{foo}(1:X \ 2: \text{bar}(1:Y) \ 3: \text{a})$.

A record is stored\(^4\) as the atom $l$ representing the label of the record, the arity of the record, which in general is a hash table with the features of the record as keys. The case where all features are enumerated integers is optimized, and does not have a hash table at all. Instead, the integers directly references the entries in the entry table, and the arity is stored as the number $n$ of features. This representation is entirely consistent with the descriptions of structures in the literature.

The hash table indexes into the entry table, which contains the actual entries of the record. An entry is a normal variable, which can be bound or unbound. For a nested record, this works as a pointer to the sub-record.

For example, the record $\text{bar}(a:X \ \text{foo}(\text{sub}(b:Y) \ b:\text{third})$ is stored as depicted in Figure 4.4.

Figure 4.4: An illustration of how the record $\text{bar}(a:X \ \text{foo}(\text{sub}(b:Y) \ b:\text{third})$ is stored in the MVM variable system. The round boxes represent variables that must be present for the example to work — the ‘\_’ character represent the variable that references the $\text{bar}$ record, if any. The “headlines” of the big boxes (“record”, “var” and “atom”) is a tag for the type of data that is stored in the data structure node.

\(^3\)In the literature, this is often referred to as the functor of the structure

\(^4\)Actually, the data structure also contains the record tag to identify the type.
4.4.2 Building Records

The process of building records can be done with the instructions putRecord, setConstant and setValue. More narrow variants handling the cases of lists and tuples, which are special cases of general records, also exists.

In general, there exists specialized instructions for many of the data types in Oz. In this section, we only cover those instructions that is needed to understand the unification and record building in the virtual machine.

First, the record node is created with the putRecord instruction, which adds a record node to the variable store, stores the feature hash table of the record, and prepares it to be filled with the entries of the record. The entries of the record must then be initialized by the same number of initialization instructions as there are features in the hash table.

The initialization instructions setConstant and setValue sets the current entry in the record to respectively an atom or a value. A value in this case is a register which references either a variable or a value. After the execution of an initialization instruction, the next entry in the record is automatically pointed to be set by the next instruction.

The setValue instruction is used to build sub-records, by first building the sub-record with a variable as reference, and then using setValue to enter this reference into the record. The interested reader should consult [13] for more information.

For example, the code for the program

```
local
    X Y Z = bar(a:X b:second foo:sub(b:Y))
in
    {Show [X Y Z]}
end
```

produces the following bytecode for building the record:

```
1   putRecord(sub [b] x(0))
2   createVariable(x(2))
3   setValue(x(2))
4   putRecord(bar [a b foo] x(1))
5   createVariable(x(3))
6   setValue(x(3))
7   setConstant(second)
8   setValue(x(0))
```

The variable X was assigned to x(3), and x(2) and x(1) was assigned to Y and X respectively. First, the sub-term sub(b:Y) is constructed with a reference to it stored in x(0). Then the record bar(a:X b:second foo:x(0)) is constructed with a reference to the node stored in x(1).
4.4.3 Unification with a Record

Unification with a known record can be done with special instructions, where the record is unified with the unknown term in a step-by-step process. Another approach is to always build all records in memory before unification with the more general unify instruction, but this instruction is slow, and complete construction of the record to unify with must not be done in many cases.

To unify an unknown term with a known record, the known record is not built up in memory — instead it is tested against the unknown term in a similar process. This is done using the getRecord instruction, which checks if the label and set of features are the same. Then, the entries of the record can be tested with instructions like unifyValue, unifyNumber and unifyLiteral in the same way as when building the record.

For example, assume that the variable X is external in the previous example. In this case, the compiler cannot tell if X has been used before. Therefore, a more general unification has to be done. For the following program:

```
local
    X Y Z
proc {Test}
    Z = bar(a:X b:second foo:sub(b:Y))
end
in
    {Test}
end
```

the bytecode that is emitted for unifying the data structure becomes:

```
1   getRecord(bar [a b foo] g(0))
2   unifyValue(g(1))
3   unifyLiteral(second)
4   createVariable(x(0))
5   unifyValue(x(0))
6   getRecord(sub [b] x(0))
7   unifyValue(g(2))
```

The G-register g(0) was assigned to Z, and g(1) and g(2) was assigned to X and Y respectively. Notice that unification instructions are used because the variables are unknown.

4.4.4 General Variable Unification

Unification between two unknown terms, like variables, are done using the unify instruction. For example, the code
local
    X Y
    proc {Test}
        X = Y
    end
in
    {Test}
end

produces
1      move(g(0) x(0))
2      unify(x(0) g(1))

for the unification x = y, since both are unknown. The move-instruction is
emitted because unify does not handle unification between two G-registers.

4.4.5 Unification with Constants

Unification with constants can be performed effectively with instructions
like getNumber and getLiteral. In this example, the external variable X is
unified with the atom ’foo’.

local
    X Y
    proc {Test}
        X = foo
    end
in
    {Test}
end

This program produces
1      getLiteral(foo g(0))

for the unification.

If the compiler knows that a variable has not been used, it can instead
skip unification and simply writing the data reference to the register that
applies. The program

local
    X = foo
in
    {Show X}
end

This program now produces
1      putConstant(foo x(0))

for the “unification”, which is far more effective than using getLiteral.
4.5 Builtins

Many of the operations in Oz that are part of the language reside as “builtins” in the compiler. This means that they are not part of the MVM, but instead referenced through a specific builtin-call.

Examples of builtins are \texttt{Value.byNeed}, described in Chapter 3, \texttt{Array.is}, determining if a variable is bound to an array, \texttt{Atom.toString}, converting an atom to a string, \texttt{Cell.assign}, storing a value in a cell, and \texttt{Object.is}, determining if a variable is bound to an object.
Part I

Compiler Techniques
Chapter 5

Liveness Analysis

In this part of the thesis we will describe the topics that applies to our implementation in a more general perspective. We have chosen to cover liveness analysis, register allocation and different optimization techniques since they relate closely to our work with a new design of the back-end of the compiler.

5.1 Liveness Analysis

This chapter describes liveness analysis applied to languages like Oz and Prolog. Information about a more general liveness analysis can be found in [14, 15, 16, 17, 18, 19].

During register allocation, information about when variables are declared and when they are used for the last time is required. Acquiring this information is called doing Liveness Analysis.

Liveness analysis is often done on a set of virtual registers instead of upon the actual variables themselves. The virtual registers are just an enumeration of the different variables present in the program, and when producing the virtual registers, name clashes and scope issues are handled. This step is present so that the liveness analysis and register allocator can disregard from scoping, name clashes and name resolving.

The purpose of the liveness analysis is to determine whether distinct virtual registers are live at the same time. A virtual register (or variable for that matter) is alive in all instructions, even those that does not use it, between the first occurrence of that register until the last.¹

The information acquired in the liveness analysis phase is used by the register allocator when deciding which virtual register will be stored in which physical registers.

¹This definition is a little bit different from the definitions found in the literature regarding liveness analysis. However, this more simple approach is enough for the MVM today.
CHAPTER 5. LIVENESS ANALYSIS

Often when talking about liveness analysis and register allocation, graphs are used to illustrate the liveness by using graphs, where the nodes correspond to the virtual registers, and the arcs describe the relation of being alive at the same time.

In the following example where virtual registers 1, 2 and 4 are live at the same time, the relation “are live at the same time” can be described by the set of unordered virtual register pairs \{ (1, 2) (2, 4) (1, 4) \}, as shown in figure 5.1.

![Figure 5.1: The liveness graph \{ (1, 2) (2, 4) (1, 4) \}. Variable 1 and 2 are live at the same time, as are variable 2 and 4 and variable 1 and 4.](image)

5.1.1 Liveness in Practice

The idea for the liveness analysis can be described as follows. First make a forward pass through the code, noting the “birth” of each virtual register when it first occurs. We pass along the information gained to the next instruction, and end up with notes at each instruction telling which registers ever has been “born” before.

More formally, the birth-sets of the instructions can be recursively defined as

\[
B_1 = \text{usedRegs}(1) \\
B_{i+1} = \text{usedRegs}(i + 1) \cup B_i
\]

where \text{usedRegs}(i) computes the used registers in instruction\(^2 i\).

Next, we make a backward pass through the code, now noting the “death” of each register when “first” occurred (remember we make the pass backwards, therefore the “first” occurrence of a register really denote the last occurrence). As before, we pass along the information gained to the next instruction (that is, the previous instruction in the regular program), and end up with notes telling which registers ever will “die” after this instruction.

Formally, the second step of the algorithm can be recursively defined as

\[
D_n = \text{usedRegs}(1) \\
D_i = \text{usedRegs}(i) \cup D_{i+1}
\]

\(^2\)Actually, \(i\) denotes the number of the instruction, not the instruction itself. However, the authors feel that this notation was appropriate when describing the algorithms.
where \( n \) is the last instruction (actually, the number of the last instruction) of the program.

The third step of the liveness analysis is combining the information from the both passes into the real liveness information. If the procedure described maintain two sets \( B_i \) and \( D_i \) for each instruction \( i \), we then form the intersection

\[
\forall i : L_i \leftarrow B_i \cap D_i
\]

which describe the liveness \( L_i \) for each instruction.

### 5.1.2 Liveness Example

For example, consider the following abstract code, where virtual registers are referenced by integers:

```plaintext
aEquateConstant(constant: 'System' vreg:162)
aInlineDot(alwaysSucceeds:false feature:'System'
result:167 vreg:158)
aUnify(167 160)
aEquateConstant(constant: 'Application' vreg:163)
aInlineDot(alwaysSucceeds:false feature:'Application'
result:168 vreg:158)
aUnify(168 161)
```

**First pass:** Executing the first pass of the algorithm described above, the following information about the liveness state of the virtual register is acquired:

\[
\begin{align*}
B_1 &= \{162\} \\
B_2 &= \{158, 162, 167\} \\
B_3 &= \{158, 160, 162, 167\} \\
B_4 &= \{158, 160, 162, 163, 167\} \\
B_5 &= \{158, 160, 162, 163, 167, 168\} \\
B_6 &= \{158, 160, 161, 162, 163, 167, 168\}
\end{align*}
\]

Observe how the information \( B_i \) in a instruction \( i \) is the basis for \( B_{i+1} \).

**Second pass:** The second pass yields the following information:

\[
\begin{align*}
D_1 &= \{158, 160, 161, 162, 163, 167, 168\} \\
D_2 &= \{158, 160, 161, 163, 167, 168\} \\
D_3 &= \{158, 160, 161, 163, 167, 168\} \\
D_4 &= \{158, 161, 163, 168\}
\end{align*}
\]
\[ D_5 = \{158, 161, 168\} \]
\[ D_6 = \{161, 168\} \]

**Third pass:** Combining the acquired information results in the following liveness data:

\[ L_1 = \{162\} \]
\[ L_2 = \{158, 167\} \]
\[ L_3 = \{158, 160, 167\} \]
\[ L_4 = \{158, 163, 168\} \]
\[ L_5 = \{158, 161, 168\} \]
\[ L_6 = \{161, 168\} \]

### 5.2 Disjoint Paths

This all looks very appealing in theory, but one very frequently occurring situation is yet to be handled; the situation where the path of execution divides, and the situation where multiple paths joins.

As examples of constructs that have to be handled with special care, we take the `if` construct available in most languages. In Oz Abstract Instructions, this construct is available as the two instructions `aTestBool` and `aShared` (see Section 8.4 and Appendix D), where `aShared` has a special meaning as a “collector” of disjoint paths, as in the end of an `if`-construct (see figure 5.2)\(^3\)

![Diagram of Oz if-construct showing the different paths forking from it.](image)

---

\(^3\)The third path in the figure is the `error` path, taken if the boolean expression generates an error.
5.2.1 Forking Paths

This situation occurs at every if-related instruction. Naturally, liveness data must be collected for all paths, but secondly, the liveness data must be consistent when the paths rejoin!

5.2.2 Inconsistent Liveness

At later stages in the compiler when emitting code, the birth of a variable indicates that an instruction creating a new variable in the variable store has to be emitted. If this is not done, the resulting program will probably access uninitialized memory and crash.

If the liveness is inconsistent, there may be multiple creations of a variable (which in worst case results in loss of variables, and a bug-infested compiler), or (even worse) non-existing variable creations. Also, inconsistency results in absent death of variables, which leads to unused resources (registers).

The liveness becomes inconsistent after an aShared-node if care is not taken to ensure that the liveness at the node is consistent with all of the paths joining. Also, the liveness becomes inconsistent before an aTestBool or other selection instruction if some of the paths’ liveness are inconsistent with the liveness at the selection node. The first situation is described in 5.3.

Associating liveness with nodes is a bad idea, since one has to keep track of both the types of inconsistency described above. A better idea is to associate liveness with paths, describing which variables has died (after the last usage at the previous node) and which variables is born (in the next node). This scheme makes sure that the allocation is consistent in the second type of sense mentioned.

To ensure that the liveness data collected is consistent at a path splitting, the following strategy can be used:

- at a selection instruction, perform liveness analysis on the paths in any order, stopping at the shared node that mark the end of the construct.
- at the shared node, save the liveness mask (the born registers so far) in the node

See Chapter 11 for more information about consistency and register allocation.
Figure 5.3: Inconsistent liveness for the Oz if construct. The problem is to decide which registers have been born in all paths joining at \( \text{fi} \).

5.3 Improving the Liveness Analysis

The algorithm described above makes three passes over the source code, and thereby the liveness analysis is in linear \( (O(n)) \) time\(^4\), assuming that set operations are in constant \( (O(1)) \) time.

The number of passes can with little effort be reduced to two, by combining the information from the first and the second pass at the same time as the second pass is done. Memory usage also benefits from this modification, because there is not any need of storing both sets anymore.

5.3.1 Killset Analysis

In some cases, the full-blown liveness analysis is unnecessarily powerful – sometimes it is enough to locate the last usage of a variable only. Doing this is called doing a Killset Analysis, since the last occurrence of a variable is often referred to as its death – the variable is killed when it is no longer used.

Doing a killset analysis is almost the same as doing a liveness analysis, except that one does not care when a variable is used for the first time – this information can be extracted rather easily directly when doing register allocation. Instead, we concentrate on the death of the variables at each instruction – this information is called the kill set of the instruction.

What this means in practice is that the first pass of the liveness analysis as described above can be skipped. Thus, the killset analysis can effectively be performed in one single pass.

\(^{4}\)Actually, since the liveness analysis must at least collect the liveness data from the source code, the process of doing liveness analysis is in \( \Omega(n) \) time.
Algorithm 1 liveness, Calculate the liveness for a definition

if $I$ is aShared then
    if $I$ contains pre-calculated ($\hat{K}, \hat{M}$) then
        $(K, M) \leftarrow (\hat{K}, \hat{M})$
        $\leftarrow (K, M)$
    else
        calculate killset $K$ and mask $M$ as below
        store $K$ and $M$ in $I$
        $\leftarrow (K, M)$
    end if
else if $I$ is aTestBool then
    $(\hat{K}_T, \hat{M}_T) \leftarrow \text{killset}(\text{cont}_T(I))$
    $(\hat{K}_F, \hat{M}_F) \leftarrow \text{killset}(\text{cont}_F(I))$
    $(\hat{K}_E, \hat{M}_E) \leftarrow \text{killset}(\text{cont}_E(I))$
    $K_T \leftarrow \text{used}(I) \cup (R \setminus \hat{M}_T)$
    $K_F \leftarrow \text{used}(I) \cup (R \setminus \hat{M}_F)$
    $K_E \leftarrow \text{used}(I) \cup (R \setminus \hat{M}_E)$
    $K \leftarrow (K_T, K_F, K_E)$
    $M \leftarrow K_T \cap \hat{M}_T \cap K_F \cap \hat{M}_F \cap K_E \cap \hat{M}_E$
    $\leftarrow (K, M)$
else
    $(\hat{K}, \hat{M}) \leftarrow \text{killset}(\text{cont}(I))$
    $K \leftarrow \text{used}(I) \cup (R \setminus \hat{M})$
    $M \leftarrow \text{used}(I) \cap \hat{M}$
    $\leftarrow (K, M)$
end if
Algorithm 2 \texttt{kilset}, Calculate the killset for a definition

\begin{align*}
\text{Input:} & \quad I & \quad \text{a valid abstract instruction} \\
& \quad R & \quad \text{the complete register set}
\end{align*}

\textbf{Require:}

\begin{align*}
\cont(x) & \quad \text{returns } x\text{'s continuation} \\
\cont\{T,F,E\}(x) & \quad \text{returns } x\text{'s true- false- and error-} \\
& \quad \text{continuation} \\
\used(x) & \quad \text{returns the used registers in } x
\end{align*}

\textbf{Ensure:}

\begin{align*}
K & \quad \text{killset for } I \\
M & \quad \text{mask for } I
\end{align*}

\begin{algorithmic}
\If{$I = \texttt{aShared}$}
\If{$I$ contains pre-calculated ($\hat{K}, \hat{M}$)}
\State ($K, M$) $\leftarrow$ ($\hat{K}, \hat{M}$)
\State $\leftarrow$ ($K, M$)
\Else
\State calculate killset $K$ and mask $M$ as below
\State store $K$ and $M$ in $I$
\State $\leftarrow$ ($K, M$)
\EndIf
\ElseIf{$I = \texttt{aTestBool}$}
\State ($\hat{K}_T, \hat{M}_T$) $\leftarrow$ killset($\cont_T(I)$)
\State ($\hat{K}_F, \hat{M}_F$) $\leftarrow$ killset($\cont_F(I)$)
\State ($\hat{K}_E, \hat{M}_E$) $\leftarrow$ killset($\cont_E(I)$)
\State $K_T$ $\leftarrow$ $\used(I) \cup (R \setminus \hat{M}_T)$
\State $K_F$ $\leftarrow$ $\used(I) \cup (R \setminus \hat{M}_F)$
\State $K_E$ $\leftarrow$ $\used(I) \cup (R \setminus \hat{M}_E)$
\State $K$ $\leftarrow$ ($K_T, K_F, K_E$)
\State $M$ $\leftarrow$ $K_T \cap \hat{M}_T \cap K_F \cap \hat{M}_F \cap K_E \cap \hat{M}_E$
\State $\leftarrow$ ($K, M$)
\Else
\State ($\hat{K}, \hat{M}$) $\leftarrow$ killset($\cont(I)$)
\State $K$ $\leftarrow$ $\used(I) \cup (R \setminus \hat{M})$
\State $M$ $\leftarrow$ $\used(I) \cap \hat{M}$
\State $\leftarrow$ ($K, M$)
\EndIf
\end{algorithmic}
Chapter 6

Physical Register Allocation

Register allocation is the process of assigning registers to the variables used in a program. Often, it is needed for data to be present in one or several registers to execute the instructions necessary. Also, constraints on which registers are to be used often occur.

This chapter is closely connected to Chapter 11, which describes the design choices made for the (rather trivial) implementation of the Oz register allocator. Further information about the register allocation process applied to Oz can also be found in Chapter 4.

6.1 The Goal of Register Allocation

As stated in Chapter 11, the goal of register allocation and optimization in Oz is to (in order of decreasing importance):

- Minimize the number of move-instructions emitted.
- Minimize the number of Y-registers used.
- Minimize the number of X-registers used.

The goal of register allocation in Oz as stated above is a bit different from the goal of a register allocator directed e.g. at an assembly target platform. The registers are as stated above available in quite a large quantity in Oz, and on top of that are discarded at each procedure or builtin call, as opposed to a low-level hardware target platform, where register are rare and instead very fast. Often, the low-level hardware platform can also access memory directly, thus bypassing the registers. This is not possible in the Oz VM architecture.

The register allocation goal in a compiler targeted at a platform like this is often to gain speed by using registers instead of direct memory accesses. In
Oz, the main goal is also to gain speed, but instead by doing smart register allocations which require few move-instructions.

In [16] and [15], some register allocation techniques are described.

6.2 Local Register Allocation

The idea of local register allocation is to weight inner loops of a program more heavily than outer ones, and more heavily than code not contained in loops. Then, the allocation benefits can be determined via heuristics utilizing the mentioned weight properties or via profiling information. Liveness information is also taken into account.

This approach was one of the first techniques used to perform register allocation. It is easy to implement, often works remarkably well, and was pretty much the only used technique before global register allocation techniques, like the graph coloring technique, became feasible to implement in a compiler for production use.

6.3 The Graph-Coloring Problem

When solving difficult problems, it is often an advantage if an abstraction of the problem can be found, thereby being able to avoid messy details of the original problem and instead concentrate on solving the clean abstracted problem.

The register allocation problem can be generalized to the problem of Graph Coloring, see for example [18, 19, 17]. As described in Chapter 5, the liveness of the virtual register set can be seen as a dependency graph $G = (V, E)$, where $V$ is a set of vertices (or nodes) depicting the virtual register set, and $P$ a set of edges conforming to the dependencies between the vertices. An edge is an unordered pair $(s, t)$ where $s$ and $t$ are the vertices connected via the edge. An edge $(s, t)$ represent that virtual registers $s$ and $t$ are live at the same time.

Two vertices $u, v$ are said to be adjacent (or neighbors) in the graph if there is an edge $(u, v)$ or $(v, u)$ in the edge set.

An example of a dependency graph is

$$G = ( \{ v_1, v_2, v_3, v_4 \} , \{ (v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_4) \} )$$

which can be visualized by the graph in figure 6.1.
Figure 6.1: The dependency graph $G = \{(v_1, v_2, v_3, v_4), \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4)\}\}$. Variable $v_1$ and $v_2$ are live at the same time, as are $v_1$ and $v_3$, $v_1$ and $v_4$, and $v_2$ and $v_4$.

The problem now is to find an assignment of colors (representing physical registers) to the vertices such that no adjacent vertices has the same color, or more formally:

$$\forall u \forall v : u \in V \land v \in V \land (u, v) \in E \land u \neq v \rightarrow c(u) \neq c(v)$$

The graph above can be colored by the mapping

$$c = \{v_1/c_1, v_2/c_2, v_3/c_3, v_4/c_2\}$$

showing that three colors are enough to color the graph.

### 6.3.1 Graph Coloring Algorithms

The goal of register allocation via graph coloring is to minimize the number of spills that has to occur. A spill occur when the number of available registers are not enough, and some virtual register has to be stored somewhere else.

The general outline of a graph coloring algorithm is to try to color the graph using $k$ colors, where $k$ is the number of available registers. If this is successful, the register mapping is given from the algorithm. Otherwise, some node is chosen to be spilled, and instructions for storing and retrieving the node is introduced. Next, the inference graph is modified appropriately (the problem is relaxed) and the register allocation is retried.

Determining if a graph is colorable with $k$ colors is in general NP-complete, but the following heuristic often gives sufficiently good results in reasonable time.

Suppose that a node $n$ in a graph $G$ has fewer than $k$ adjacent nodes. Remove $n$ and its edges from the graph to form $G'$. A $k$-coloring of $G'$ can be extended to a $k$-coloring of $G$ by assigning $n$ a color not assigned to any of its neighbors.

In [18] and [19] the idea about register allocation via graph coloring is extended with heuristics for choosing the node to be spilled. Also, several modifications to the original graph coloring algorithm has been proposed.
6.4 Other Approaches to Register Allocation

Several other approaches to register allocation via graph coloring has been proposed, among others techniques that use the control tree of a procedure to guide spilling and graph-pruning decisions [15]. The control tree of a procedure describe the relation between basic blocks and the original flow graph.

6.4.1 Graph Coloring via Maximal Clique Separators

Another interesting approach developed and presented by Gupta, Soffa and Steele [20] uses maximal clique separators to perform graph coloring. The strategy is based on results due to Tarjan [21] regarding the colorability of a graph by dividing it into subgraphs.

A clique separator is a completely connected subgraph CS of G whose removal disconnects G into at least two subgraphs $S_i, i \in \{1..n\} \land n \geq 2$. These subgraphs may be further decomposable into smaller subgraphs using other clique separators.

If each subgraph $S_i$ is colored using at most $k$ colors, then the entire graph $G$ can be colored using $k$ colors by combining the colorings of the subgraphs.

For the graph in figure 6.2, the clique $CS = \{v_3, v_6, v_7\}$ is a separator as seen in figure 6.3, as its removal results in disconnected subgraphs $S_1 = \{v_1, v_2, v_8, v_9\}$ and $S_2 = \{v_4, v_5\}$. These subgraphs are used to construct the graphs seen in figure 6.4, which can be colored separately using at most 3 colors as seen in the graph. The graphs shown in the figure is constructed by including the members of the clique separator $CS$ in each of the disconnected subgraphs.

![Graph Coloring Problem](image)

Figure 6.2: A graph coloring problem.
Figure 6.3: Graph coloring problem showing the clique separator $CS = \{v_3, v_6, v_7\}$.

Figure 6.4: Subgraphs constructed from $S_1 = \{v_1, v_2, v_8, v_9\}$, $S_2 = \{v_4, v_5\}$ and $CS = \{v_3, v_6, v_7\}$ with 3-coloring.

The colored subgraphs can now be combined to construct a coloring for the original graph $G$. The process involves renaming of colors in some of the subgraphs ($S_2 \cup CS$ in this example) so that all subgraphs use the same colors for the members of the clique. In this example, the combination became possible by interchanging the use of colors $c_1$ and $c_2$ in the subgraph including $S_2 \cup CS$, as seen in 6.5.

Figure 6.5: The complete 3-coloring for the graph.

Subgraphs that cannot be further decomposable are called atoms. In the previous example, $S_2$ is an atom, but $S_1$ can be further decomposed by clique $\{v_2, v_9\}$.

In [20], Gupta, Soffa and Steele describes techniques for register allocation via maximal clique separators, derived from the algorithms proposed
by Tarjan in [21]. It is possible to perform the register allocation without constructing an explicit graph as in the examples.
Chapter 7

Optimizations

This chapter will describe some optimization techniques of interest for our thesis project. That is, optimizations that we have, or are planning to, change or introduce compared to the original implementation of Oz compiler.

7.1 Peephole Optimization

When emitting code, often redundant instructions and inefficient code are produced, with suboptimal constructs existing. For example, in Oz there is a difference when initializing a variable and unifying it. Often, code creating a new variable, only to unify it in the next statement is often produced.

Inefficient bytecode:

\[
\text{createVariable}(x(20)) \quad \% \text{ Create a new variable in } x(20) \\
\text{equateConstant}(42 \ x(20)) \quad \% \text{ Equate this variable with 42}
\]

Such a code sequence is semantically equivalent (provided that the program never branches to the second instruction) to initialization of \( x(20) \) directly.

Semantically equivalent bytecode:

\[
\text{putConstant}(42 \ x(20)) \quad \% \text{ Equate this variable with 42}
\]

This is a typical peephole optimization, and actually one which is done in the Oz compiler.

Peephole optimization is one of the most used optimizations, and can produce quite amazing results in certain circumstances. The idea behind peephole optimization is to scan the produced code in a sequential manner, and replace inefficient sequences of instructions with more efficient sequences — very suitable to optimize small sequences of inefficient as the ones mentioned above.
7.2 Tail Call Optimization

In Oz, as in many functional languages, all loops are implemented as recursive calls. At each call, a stack frame for variables that survive the call must be allocated, and a closure of external references must be held in memory. All these operations is time and memory consuming, which would make all loops in Oz very ineffective if tail call optimization was not present.

Tail call optimization is an optimization that tries to minimize overhead in calls that are performed at the end of a procedure. If a recursive call is the absolutely last instruction in a procedure, it is impossible for the procedure to return after the call to do something more. These calls are said to be tail calls, because they operate at the tail of a procedure. This property can for example be used to optimize recursive calls in a very efficient way.

A tail call can simply be replaced by a jump instruction to the head of the procedure to be called, provided that the arguments are built in the right way. This optimization is present in Oz and is almost an absolute necessity for a functional language to be efficient.

7.3 Inlining


7.3.1 Inline Expansion

This optimization is done in assembly language or in machine code. It is basically a macro where an assembly language mnemonics is expanded to a number of assembly, or machine code, instructions. Each mnemonic has a corresponding template that consists of the code, the number of arguments expected and a list of register identifiers that must be substituted for real registers. The built-ins of Oz are sort of mix between inline expansion and ordinary call instructions. Each built-in has this type of template and although the code is never expanded inline, there is no ordinary call either. The usual parameter passing technique is not used, for example.

7.3.2 Automatic Inlining

In the other type of inlining, automatic inlining, a call to a procedure is replaced by the body of that procedure. This is usually done at intermediate code level, which leads to some extra advantages. Not only does this optimization eliminate the overhead of the procedure call, but also since the procedure body now is local, other types of optimizations, and register allocation, may be more efficient. Take the following Oz program for example:
CHAPTER 7. OPTIMIZATIONS

```
declare
proc {Foo X}
    Y = X
end
X = 1 {Foo X}
Y = X
```

This will be compiled into the following absolute bytecode:

```
1  %% Assignment of Global Registers:
2  %% g(0) = Foo
3  %% g(1) = Y
4  %% g(2) = X
5  %% Code Size: 40 words
6    Skip
7  lbl(1) definition(x(0) 37 pid('Toplevel abstraction' 0
8      pos('' 1 0) [sited] 1) unit [g(0) g(1) g(2)])
9  lbl(7) definition(x(0) 19 pid('Foo' 1 pos('Oz' 2 0) nil
10      1) <P: 1> [g(1)]))
11     unify(x(0) g(0))
12    return
13   endDefinition(7)
14  lbl(19) unify(x(0) g(0))
15    getNumber(1 g(2))
16    move(g(2) x(0))
17    callProcedureRef(<P: 1> 2)
18    getNumber(1 g(1))
19    return
20  endDefinition(1)
21  lbl(37) tailCall(x(0) 0)
```

If the call to Foo were to be inlined, the unify instruction at line 11 could be optimized, as the unify that would have been in line 18 is, to a getNumber instruction. There are also some loop transformation optimizations that are only possible if the loop does not contain any procedure calls, that is another example of where inlining a call may allow other types of optimizations besides the removed procedure call overhead.

### 7.3.3 Deciding When to Inline

An interesting question when using automatic inlining, is to decide which calls to inline. This can be left up to the programmer to decide, partly or totally (although when the only procedures that are inlined are those that the programmer explicitly decide so, the term procedure integration is more
accurate, since it is not at all automatic). In Oz this might be achieved by adding a procedure flag that the programmer may use to mark those procedure that she wants to inline. If it is up to the compiler to decide when to inline, some sort of heuristic has to be used. If a call contains constant-valued parameters or resides inside loop, as in the two examples above, it is more likely that inlining it will be advantageous. Other factors are the size of the procedure body, the number of calls and so on.

The decision on when to inline is not as obvious as it might seem at first appearance, especially not with the increasing importance of caches in modern processors. Sometimes it may take less time to make a few calls to a procedure whose body resides in cache memory, than to inline those calls and be forced to fetch instructions from the main memory. Therefore the heuristic for when to inline should be based on the analysis of thorough benchmarks.

7.4 Delayed initialization

This optimization was previously done at the same time as register allocation, but since we wanted each phase of the compiler as simple and isolated as possible, we decided to move it to a separate phase. Since we have not yet implemented it, we can see, when comparing output from our compiler and the original one, that this optimization has a major impact on both the size and the execution speed of the generated code.

Even though the effect is powerful, the optimization is actually quite simple. The instruction that initializes a register (real or virtual register, depending on in which phase it is done) is moved down until the next use of that register. There are several ways that this might speed up execution time. For example:

- If there exists one or more execution paths of a procedure body where that register is never used, the initialization of it should be moved down, so that if traversing one of this paths, the initialization instruction will never be executed.

- No real register has to be allocated before it is absolutely necessary, and hence register conflicts might be avoided.

- If the first use (or in certain cases, a later use) of the register has the constraint that it must be a X-register and there are one or more call instructions between the initialization and first use, the initialization has to be made to a Y-register from which it is then moved to the demanded X-register before its use. Delaying the initialization removes that unnecessary move instruction.

To illustrate these points we transform the following Oz function into bytecode:
CHAPTER 7. OPTIMIZATIONS

```
proc {Foo A B} X in
    X = bar
    if A == foo then
        {System.show B}
        {System.show X}
    else
        {System.show B}
    end
end
```

This yield the following listing:

```
1  lbl(7)  definition(x(0) 64 pid('Foo' 2 pos('Oz' 2 0) nil 3)
2     <P: 1> [g(1)]
3   testLiteral(x(0) foo 47)
4   move(g(0) x(0))
5   inlineDot(x(0) show x(2) cache)
6   move(x(1) x(0))
7   call(x(2) 1)
8   move(g(0) x(0))
9   inlineDot(x(0) show x(1) cache)
10  putConstant(bar x(0))
11  tailCall(x(1) 1)
12  lbl(47) move(g(0) x(0))
13   inlineDot(x(0) show x(2) cache)
14   move(x(1) x(0))
15   tailCall(x(2) 1)
16  endDefinition(7)
```

Without this optimization, the instruction that initializes the register allocated to variable X, would have been located at line 3. This is now instead performed by the putConstant instruction at line 10. This delayed initialization exemplifies all of the above mention advantages.

- If A != foo, the execution will branch to label 47 at line 12 and the initialization instruction will not be executed at all.

- A conflicting register constraint is avoided. Variable X has to be allocated to register x(0) in the call at line 11, but so does parameter B at line 7. Without delayed initialization, X would have been allocated to another register, leading to an extra move instruction before the call.

- The call instruction at line 7 would force the variable to be initialized to a Y-register since any X-registers might be corrupted before that call\(^1\).

\(^1\)For this particular example, the problem in this point (like its solution) coincides with the previous point.
7.5 Future Oz-specific Optimizations

This section describes two optimizations that are possible because of Oz's parameter passing technique.

7.5.1 The Cheap Call Optimization

A lot of procedures (and its invocations of other procedures in turn) only uses a relatively small static number of registers. An analysis of which registers are used and, in turn, which registers are used in subsequent calls in the procedure body could in this case make it possible to get by a procedure call without allocating any Y-registers at all. Instead, allocation in a procedure body could be guided by which registers are destroyed in a call, and if needed save X-registers that are destroyed in unaffected X-registers instead.

7.5.2 The Call Convention Optimization

The goal of this data-flow analysis optimization is to minimize the number of data copying instructions, i.e., to reduce the number of move instructions. The usual convention of passing actual parameters during procedure calls, is to use registers $x(0), x(1) \ldots x(n - 1)$, where $n$ is the arity of the called procedure. Via data-flow analysis we would try to find procedures where it would be suitable to use other X-register to pass parameters. The basic requirement is that every call to this procedure must be known. This can be assured by only consider those procedures that is not defined on top-level, that is, they are nested inside another procedure definition. If this procedure is referenced outside of the definition scope, via a binding to a non-local variable, we must try to locate this variable and check whether it's used to call the procedure or if any further bindings to this variable take place.

If we can be sure that we know every location from where the procedure is called, we can analyze the code of the called procedure, and of all the calling procedures, to try to find some arrangement of the parameter registers that will lead to fewer move instructions. Lets take a simple example. We have a procedure $\text{Foo}$ that takes one argument, referenced by register $x(0)$ with the normal call convention. When we analyze the calls to this procedure, we find that the actual parameters that are used, are often allocated to register $x(1)$, which will make it necessary to introduce a move($x(1)$, $x(0)$) instruction before the calls. Instead we break the call convention, and let all calls to $\text{Foo}$, and also the definition of $\text{Foo}$ itself, used register $x(1)$ as parameter register. This removes the need of the move instructions. Of course, those moves should be avoided already during register allocation, but sometimes this can't be achieved.

The data-flow analysis needed for this optimization, would likely be quite
time consuming, so it would probably be wise to make this optimization optional.
Part II

Design
Chapter 8

Overview of the New compiler

8.1 Area of Interest
The Oz compiler is a very complicated piece of software, currently about 25,000 lines of Oz code. Thus, we had to restrict our task quite a bit.

8.2 Design Overview
The design of the new compiler is modeled with the underlying thought that the compiler should work in several passes, and instead of doing a lot in a few places, little work should be done in several phases of compilation. Thus, it would be simple to exchange different parts of the compiler, and modification of the compiler back-end could be restricted locally to modification in a module, instead of having to modify (and debug) a very large, interconnected piece of code.

8.3 Phases
In our design, we have striven to modularize the back-end so that every logically separable task is performed in a separate phase. Intermediate code generation, liveness analysis, register allocation and so on, are all performed in separate phases, each adding new information to the instructions and/or removing information that is no longer needed. This does not only simplify changes to the existing phases but, perhaps more importantly, makes it easier to add new phases to the back-end. For example, a pass which makes inlining of function calls will probably be added in the future. Such a pass could simply be inserted between the intermediate code generation phase and the liveness analysis, without any changes to those phases at all. The writer of the inlining phase would merely have to make sure that the new phase generates the correct output format, that is, the instruction format that the next phase, in this case the liveness analysis, expects. Our attempt at
achieving this desired modularization resulted in the design depicted in figure 8.2.

8.3.1 Phases comparison

In the process of analyzing and re-engineering the old compiler, we have taken several looks on the design of the old compiler as we see it — since there is very little documentation on the old compiler, and no documentation of the old compiler design, figure 8.1 is merely our view of the different “phases” of the back-end.

The compiler in figure 8.1 works mainly in a definition-by-definition way — a definition with its body is converted to Virtual Instructions, as the abstract instructions are called in the old compiler. At the same time, virtual registers are created. Then, the definition body is analyzed to find the liveness of the separate instructions it is built of. The definition body is then converted to byte code, during which register allocation and different optimizations also are done. At last, the definition (in bytecode format) is passed back, and the next definition is taken through the whole procedure.

Also, if a nested definition is encountered during compilation, the procedure is first done on the nested definition, and then compilation can continue.
Figure 8.1: The design of the old back-end.

The main difference between the old design in figure 8.1 and the new one in figure 8.2 is that in the new design, the compilation phases are performed, one at a time, on all of the program. That is, first the complete program is transformed to abstract instructions, and at the same time annotated with virtual registers. The core graph is scrapped at this time, since all we need is contained in the middle-code format the abstract instructions are. Then, all of the abstract instructions, which make up the complete program, is passed to the liveness analysis phase. Here, the instructions are annotated with liveness information, and after the pass, the previous version of the program is scrapped — once again, all we need is contained in the new annotated instructions.
8.4 Abstract Instructions

The abstract instructions are the name of the intermediate code format used in the compiler. They are described in detail in Appendix D, but we will here give an overview, which will sufficient for understanding the rest of this thesis.

8.4.1 Overview

To generalize, you can describe the abstract instructions as a high-level assembly language, with control flow information and several details abstracted away. There are no branches in this code format, neither explicitly nor implicitly via, for example, false branches in a conditional statement. Instead
there are direct references in the code. The program that is being compiled will be represented by a graph built up from abstract instruction (see figure 8.3 for an example of the structure of this code format). This makes it very easy to traverse the code by simply following the references and not having to bother about branches. These features of the intermediate code format make it very suitable for doing optimizations upon, such as register allocation, data-flow analysis, call-convention optimizations and so on.

Figure 8.3: Structure example in abstract instructions

The abstract instructions will have different features after the different phases of the back-end. The liveness analysis phase, for example, adds a feature named kill (which holds the kill-set of this instruction). This feature will then be removed in the register allocation phase, after which it is no longer needed.

Abstract instructions are based on the intermediate code format used in the original implementation of the Oz compiler, which was called virtual instructions. The biggest difference is the way that continuations are handled, which will be described in the next section. We have also tried to minimize the number of instructions by making some of them more general. There are for example only one call instruction\(^1\) while virtual instructions have several for different types of procedure calls.

8.4.2 Continuations in Abstract Instructions

When we talk about the continuation of an instruction, abstract or bytecode, we simply mean the instruction that will be executed after the current in-

\(^1\)There is only one call instruction for ordinary procedures, there is however also one instruction for calling built-ins.
instruction. Some instructions may have several continuations; meaning that there are several instructions that might be executed next, which one it will be, depend of the result of executing the current instruction.

The biggest difference between the virtual and the abstract instructions, which also is the reason to most changes in the implementation of the abstract generation pass, is the way the continuation of some instructions are represented. In a\textit{Match}, a\textit{TestBool} and a\textit{ExceptionHandler} there is no longer a direct reference to the next statement. Instead the continuation is added, starting with a \textit{aShared} instruction, at the end of all the paths of the body of the instruction. Figure 8.4 illustrates the structure of two nested \textit{aTestBool} instructions \footnote{Note that the two \textit{aTestBool} instructions connects to the same \textit{aShared} instruction.} and figure 8.5 shows their virtual counterpart.

![Diagram](image_url)

Figure 8.4: Example of continuation in the abstract instruction set, showing the abstract instructions \textit{aTestBool} and \textit{aShared}.

The \textit{aShared} instruction is needed to tie together all the paths. Without it, it would be impossible to know when traversing the code, when the path in the body of one instruction ends and the continuation starts. This information is needed several times in the back end, for example when making the register mapping consistent between all paths.
Figure 8.5: Example of continuation in the virtual instruction set, showing \texttt{vTestBool} and \texttt{vShared}.

With Virtual instruction this is handled with a continuation stack, where the top element is the continuation of the current instruction. Our way of handling continuation, which was recommended to us by Leif Kornstaedt (the developer of the original Oz compiler), makes it easier to traverse the code during different kinds of analysis or when, for instance, predicting which register is the most suitable to allocate for a certain variable. At the end of an execution path you simply skip the \texttt{aShared} instruction and continue to traverse the continuation. It is a little bit more work to implement though. You have to make sure that, for example, machine code for the continuation is only generated once, we solved this by counting the number of paths that ended in a specific \texttt{aShared} instruction and then saving that information in the instruction. Later on when traversing the code, its easy to keep a counter on how many times this \texttt{aShared} instruction has been visited and only at the last visit also traverse the continuation.
Chapter 9

The Compiler Front-End

Since our masters’ thesis project did not involve the front-end of the compiler, we will only give a brief overview of this part. For more information, including grammars for the two syntax formats used in this compiler phase, the reader is referred to [22] and appendices B and C.

9.1 The Parser

In the first phase of the front-end, the source code is translated into a syntax tree. The language used to build this tree is called †Syntax and is described in [22] and Appendix B. If the source code is syntactically incorrect, the compilation will end and a parser error will be reported.

Let us once again take the factorial function as an example: Example:

```
declare
fun {Fac N}
  if N<0 then
    1
  else
    N*{Fac N-1}
  end
end
```

If we translate this program into †Syntax, we will get the following syntax tree:

```
[fun(fVar(‘Fac’ pos(’’ 1 5)) [fVar(‘N’ pos(’’ 1 9))]
 fBoolCase(fOpApply(‘<’ [fVar(‘N’ pos(’’ 1 15))
     fInt(0 pos(’’ 1 18))] pos(’’ 1 16))
 fInt(1 pos(’’ 1 25))
 fOpApply(‘*’ [fVar(‘N’ pos(’’ 1 32))
```

76
fApply(fVar('Fac' pos(’’ 1 35))
  [fOpApply(?- [fVar('N' pos(’’ 1 39))
    fInt(1 pos(’’ 1 41))]
  pos(’’ 1 40)]) pos(’’ 1 34 ’’ 1 42))]) pos(’’ 1 33))
pos(’’ 1 12 ’’ 1 44)) nil pos(’’ 1 0 ’’ 1 48))]

The tree is built out of tuples describing the different language constructs with features containing the values of attributes of that particular language construct. We take the root of the tree, in this program the fFun tuple, and look closer at its features. This tuple has 4 features:

- First we have the variable that holds this function definition.
- Secondly there is a list with the formal arguments of the definition.
- The third feature is a sub tree describing the function body.
- Lastly there is the starting and ending coordinates of this construct. Coordinates consist of a filename (which is left out here since the filename is not known) and two integers denoting the line and column number. All constructs have at least a starting coordinate.

9.2 The Object Graph

In the next phase of the compiler, the Oz program is transformed from fSyntax into the object graph format, representing an extended core language (for efficiency reasons). Since the core language is a subset of the complete Oz language, the program cannot be directly transformed into the object graph. Instead, it must be transformed in steps to core language. This procedure is called unnesting the source code. This unnesting phase consist of the following steps:

- Expand abbreviations.
- Make implicit declarations explicit.
- Unnest expressions inside statements.
- Instantiate the graph nodes.

The graph of the unnested factorial function from the previous section is illustrated in figure 9.1. It is also possible the get a textual representation of an object graph from the OPI (Oz Programming Interface). The following is a listing of the factorial function:

```plaintext
declare Fac in
  proc {Fac N Result1}
```
local IfArbiter1 UnnestApply1 in
UnnestApply1 = 0
{`=<` N UnnestApply1 IfArbiter1}
if IfArbiter1 then
  Result1 = 1
else
  local UnnestApply2 UnnestApply3 UnnestApply4 in
  UnnestApply4 = 1
  {`-` N UnnestApply4 UnnestApply3}
  {Fac UnnestApply3 UnnestApply2}
  {`*` N UnnestApply2 Result1}
end
end
end
Figure 9.1: Object graph for the factorial function.

### 9.3 Static Analysis Phase

The last phase of the front-end is the static analysis phase, which usually adds two attributes, value and type. These attributes are needed for many of the optimizations that are applied later on, like for example constant propagation.
Chapter 10

Transformation to Intermediate Code

There are usually two reasons for an intermediate code format mentioned in compiler textbooks. The main reason is that if you let the front-end of the compiler generate intermediate code, you can use the same front end for all intended target machines and only write different back-ends (likewise you can also use the same back-end for different front-ends that generates the same intermediate code format from different source languages). This is not an issue in our case though. Our only source language is the Oz language and our only target machine is the Mozart Virtual Machine.

The second big reason for generating intermediate code has to do with modularization. If we were to translate source language directly to machine code, everything from syntactic and semantic analysis to liveness analysis and register allocation would have to be done in one single step, probably an impossible task for a language of Oz’s complexity. Instead the actions of both the front and back-end is divided into several independent passes. This also makes it much easier to change different parts of the compiler. Say for instance that someone would like to use a different register allocator. Since this is a single pass in our back end, an allocator with the same interface could, without any change to the rest of the compiler, simply replace the current one.

10.1 From Object Graph to Abstract Instructions

For all types of statements in the Oz (core) language there is a class that represents that statement in the object graph. Every pass of the front end adds, via inheritance, their own methods and attributes to the different classes. In this pass the method codeGen is called for every object in the graph with the effect of translating the object graph into abstract instructions, which is the name of the intermediate code format in the compiler (see Section 8.4
and Appendix D for a description of this code format).

In the original implementation, this translation was done in a per-procedure-definition manner with all the passes of the back end applied to one procedure before the next procedure was translated. Since we wanted a clean separation between the front and back-end, we decided to translate the whole program into intermediate code before it was handed to the back end. Also, some of the optimizations that we have planned, e.g. to change the call convention where suitable, depend on the ability to analyze the whole program at the intermediate code level.

Besides the mentioned modification, we made some changes that were needed due to the difference between abstract instructions and the previous intermediate code format, which is called virtual instructions. Abstract instructions are a bit more general; there is for example only one instruction for a test statement, which is the $\texttt{aTestBool}$ instruction, while the virtual instruction set contains several test instructions, e.g. $\texttt{vTestBool}$ and $\texttt{vTestNumber}$. $\texttt{aTestBool}$ is then, in a later pass, either translated to another intermediate instruction or directly to machine code, depending on which type of test it was. The purpose of this is to make the intermediate code as general and independent of the target machine as possible and thereby simplifying future changes to the Mozart Virtual Machine.

### 10.2 Obstacles Encountered

Although there are relatively small changes to the original implementation, we spent quite some time with this phase. We had to understand most of the implementation before we could start to make any changes to it; this was of course complicated by the fact that neither of us had any experience with Oz. The object graph uses a quite large class tree with several steps of inheritance. The fact that Oz is a dynamically typed language makes the effort of understanding source code even more difficult. When for example a method in an object is called, you have to find out the type of the object before you know which method is actually executed.

Unfortunately we were not able to compile the original compiler for debugging. Executing a program step by step is otherwise a god way to understand how it works and would certainly have shortened the time spent for us.
Chapter 11

Applied Register Allocation

The register allocation is a very important part of an optimizing compiler, if not the most important.

11.1 Register Optimization Strategies

When doing register optimization for Oz, a number of goals exist:

- Minimize the number of move-instructions emitted. When doing register allocation, the data stored in the different registers must be moved around to meet certain constrains induced by the target platform. For example, the calling convention of MVM is to pass parameters in $x(0), x(1), \ldots, x(n)$ where $n$ is the arity of the procedure called. Thus when compiling a statement involving a call machine instruction, data stored in these X-registers must be moved somewhere else, and the parameter data must be moved to the particular X-registers to be used for the call.

- Minimize the number of Y-registers used. An Y-register is a kind of “stack-variable” that has to be used to ensure survival of variables between procedure calls.

- Minimize the number of X-registers used. This is actually not at all as important as the goals described above, as by default 4096 different X-registers are available in MVM.

See Chapter 4 for more about the target platform. Since the number of external references from a procedure depends on the source code, the number of G-registers cannot be manipulated and henceforth is not optimizable.
11.2 The Original Register Allocator

The original register allocator is very effective in minimizing Y-register usage and move-code emission. The register optimizer takes advantage of the total control of the code generation at all levels to generate very effective code, streamlined to the current combination of instructions.

Unfortunately, due to the lack of structure and the fact that the original register optimizer is spread out into the source code of the compiler, we had to build a new one – the existing register allocator was very hard to modify and/or replace, and was considered expendable due to the clash with the requirements that the new compiler should be modular, flexible, and easy to modify and improve.

11.3 The New Register Allocator

We had several ideas for a new register allocator, of which some were good, and some were bad. As mentioned, the register allocator in the official Oz compiler works by predicting the future usage of the virtual register about to be allocated to a physical register. This is a reasonably good method.

We have analyzed several other “cheap” approaches, in which we could easily allocate pretty good sets of registers. One of these methods, referred as the Post-Allocation method, worked in a similar way as the original allocator. But instead of looking ahead, we chose to postpone the allocation until we knew more about the usage of the virtual register. In this way, we could achieve the same performance of allocation as the original compiler, in a more time-efficient way.

11.4 Overview

The allocation is performed on a per procedure definition basis, meaning that whenever an aDefinition or aDefinitionCopy abstract instruction is encountered, on top-level or nested inside another definition, a new register allocator object is created and used when allocation is performed on the body of that definition. There is an important point that needs to be emphasized here; the register that holds the procedure definition is allocated in the surrounding definition, or at top-level, and then patched into the corresponding aDefinition or aDefinitionCopy instruction. So those instructions are actually handled by two different allocator objects. One handles the procedure definition register and another handles the parameter- and global-registers of those instructions. This might seem odd at first, but is actually quite logical. The procedure is defined in another procedure, or at top-level, so the register that holds the definition is handled there, but the parameter-
and global-registers are associated with the procedure itself and are hence handled by another register allocator.

When a new procedure definition is encountered, the register that will hold the definition is first allocated with the current allocator. As described above, a new allocator is then created for the body of the definition. This allocator is first initialized with the parameter- and global-registers. The virtual registers that represent the formal parameters are allocated to the first N X-registers, where N is the arity of the procedure, and all non-local virtual registers are allocated to G-registers (it is the responsibility of the liveness analysis to find out which virtual registers that are non-local and to place those in the gregs-list of the instruction).

Now the instructions that constitute the body of the definition can be register allocated. Besides the two definition instructions, there are a few other instructions that need to be handled in a separate manner, but the general scheme is quite straightforward. First it's checked whether the current instruction puts any constraint on which type of register may be used. Then the allocator looks ahead to find any conflicting register constraints appearing in later instructions. Based on those constraints a specific register is chosen. Should this register already be occupied, an aReorganize instruction is inserted before this instruction. The register allocator uses the aReorganize abstract instruction to gather all moves and allocations that need to be inserted by the emitter before an instruction is translated into byte code. In this case the aReorganize only contains a single move to free the chosen register. The next step is to deallocate all registers that die after this instruction, which are all registers allocated to virtual registers that occur in the kill-set of this instruction. The last action performed on the instruction is to replace the virtual registers with the real register allocated to them and also removing the kill-set, which is no longer needed.

11.5 Register Constraints

Some instructions put constraints on which type of real registers that the virtual registers may be allocated to. The most common constraint is that it must be a X-register. There are only two exceptions; the aSetSelf instruction demands a G-register and the aGetVariable instruction either a X- or Y-register. All constraints can be found in Appendix E.

11.6 Temporary Registers

There sometimes arises the need of a temporary register, which is a X-register (it could also be a Y-register, but since we want to minimize the number of used Y-registers we try to avoid this) that is not really allocated to any virtual register but is merely used as a temporary reference to a variable.
Temporary registers are used, for example, to resolve a cyclic dependency among parameter register, as described in the section about arranging parameters. Often, a temporary register is necessary at one specific instruction only. Because temporary registers have such a short lifespan, they are not handled by the standard register allocation algorithm. Instead a free register is used as a temporary and returned to the register allocator immediately. This does not affect the register allocation procedure at all.

11.7 Deciding Which Register to Allocate

When a variable is first initialized, or rather virtual register, which is the objects that the register allocator handles, it must be decided to which register it shall be allocated to. The naïve approach is to choose the first free X-register. This will however most certainly lead to unnecessary move instructions later on. Instead we follow the continuation of the instruction until we encounter an aCall instruction, or the end of the procedure. Should there not exist any calls after this instruction, we are free to choose any register at all, and since we want to minimize the number of Y-registers used, we choose an X-register. If there is an aCall instruction we check whether the virtual register is used as an actual parameter in this call. If so, we would like to allocate it to the corresponding parameter register.

This is as far as the register allocator tries to optimize the allocation for the time being. However, there is some room for further optimization here. For example, if the virtual register is used as an actual parameter and it will survive the function call (or some other instruction in-between puts the constraint that it must be allocated to a X-register), it would be better to allocate it to both a X- and a Y-register. There are namely instructions that can initialize two registers at the same time, and hence a later move from the X-register to the Y-register in order to save it, can be avoided.

11.8 Merging

When doing allocation in disjoint paths, very frequently the register allocation becomes “inconsistent” between the different paths. This problem has to be remedied where the paths rejoin, e.g. at the end of an if-statement. The method used in our implementation is to do a kind of complicated “merging” between the disjoint allocations, and then emitting the necessary instructions to modify all register mappings into some kind of “best-fit” register mapping. The mapping map\textsubscript{M} for each virtual register \( v \) in path \( i \) is chosen according to a simple heuristic;

- If \( v \) is unused in all paths, then \( \text{map}_M(v) = \text{undef} \).
• If $\text{map}_i(v)$ is consistent ($p$) in all paths $i$ where it is used ($\forall i : \text{map}_i(v) \neq \text{undef} \Rightarrow \text{map}_i(v) = p$) then $\text{map}_M(v) = p$.

• If $\text{map}_i(v)$ is inconsistent, then the mapping $p_{\text{MOST}}$ with most occurrences is chosen for $\text{map}_M(v)$.

When $\text{map}_M$ has been generated, code for modifying the different mappings can be emitted.

11.9 Handling Function Calls

The call instructions, $\text{aCall}$ and $\text{aCallBuiltin}$, are two of those instructions that need to be handled in a special way. Here the register allocator must not just replace the virtual register with the real register that holds the definition of the called procedure, but also make sure that all X-registers that should survive the call are saved in Y-registers and that the actual parameters of the call are allocated to the right registers. In figure 11.1 a pseudo call instruction is register allocated. Virtual register 0, 1 and 2 are actual parameters to the call. There exist 4 virtual registers; all except number 2 shall survive the call. The process of saving surviving virtual registers to Y-registers and setting up parameters generates three new move instructions and changes the allocation table. The following two sections will describe these processes separately.
Pseudo code before allocation:
```
call(name: Foo args:[v(0) v(1) v(2)] kill-set:[v(2)])
```
Allocation table:

<table>
<thead>
<tr>
<th>Virtual register</th>
<th>X-register</th>
<th>Y-register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

Pseudo code after allocation:
```
move(x(2) y(2))
move(x(1) x(2))
move(y(1) x(1))
call(name: Foo arity: 3)
```
Allocation table:

<table>
<thead>
<tr>
<th>Virtual register</th>
<th>X-register</th>
<th>Y-register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 11.1: Register allocation of call instructions.

### 11.9.1 Saving Surviving Virtual Registers

All virtual registers that are alive after an `aCall` instruction must be allocated to Y- or G-registers and all X-register allocations must be removed. Should the virtual register before the call only be allocated to a X-register, a move instruction from that X-register to a new Y-register must be inserted and the X-register allocation removed.

In figure 11.1, the virtual registers 2 and 3 don’t have any Y-registers allocated to them. Register 2 won’t however survive the call, so only virtual register 3 needs to be saved, which generates the first new move instruction. As can be seen in the allocation table after the register allocation, all surviving virtual registers are now allocated to Y-registers and every X-register allocation is removed.

### 11.9.2 Arranging Parameters

This is where we make sure that the actual parameters are referenced by the right registers, which is registers x(0) to x(n), where n is the arity of the called function. If possible the register allocator should, already when
 CHAPTER 11. APPLIED REGISTER ALLOCATION

A virtual register is first allocated to a real register, try to satisfy this type of constraint. This is of course not always possible, which will lead to some extra move instructions which must be inserted before the call. We might also face the situation where a parameter register of an actual parameter is occupied by another actual parameter which has not yet been moved to its correct parameter register. There may also be cyclic dependencies, so that one actual parameter is allocated to \( x(i) \) but should be allocated to \( x(j) \), which is now occupied by the actual parameter that should be in register \( x(i) \). There might of course also be longer chains of this kind of cyclic dependences. However, if at least one of these actual parameters should survive the call, the solution is quite simple. Since that surviving parameter will already be allocated to a Y-register, its X-register can be overwritten with a move instruction without loosing all references to it, and hence the dependency chain is broken. Should there on the other hand exist a dependency chain with only register that will die after this call and where none of them are also allocated to a Y- or G-register, a temporary register must be used, and a unnecessary move instruction from one of the parameter register to this temporary register must be inserted.

In the example in figure 11.1, the second and third actual parameters (virtual registers 1 and 2), have to be moved to their correct parameter registers, which are \( x(1) \) and \( x(2) \) respectively. By first inserting a move from \( x(1) \) to \( x(2) \), we free register \( x(1) \) and are then able to insert a move from \( y(1) \) to \( x(1) \), without the use of any temporary registers.

11.9.3 Built-ins

The semantic difference of the builtin calls compared to ordinary calls, makes the register allocation of them quite different. The usual call convention is not used. Instead any X-register may serve as a parameter register (it must however be a X-register). We also know which of the parameter register that will be modified be the call procedure, and need only to save those parameters to Y-registers.
Chapter 12

Code Emission

12.1 From Abstract Instruction to Bytecode

This is the phase were the intermediate code is translated into relative bytecode. It is called relative because branches in the code do not use addresses, but symbolic labels. The labels can only be replaced by addresses after the peephole optimization (which is done in the loading phase).

The translation is very much a matter of simply replacing the current abstract instruction with the bytecode counterpart. This is done by a recursive procedure that translates the first abstract instruction to one or more bytecode instructions, and then calls itself with the continuation of that abstract instruction. There are a couple of things that need to be handled with extra care though. First, bytecode uses a different code structure, which leads to the introduction of the branch instruction. Second, to assure proper use of the registers, instructions to allocate and move between registers also need to be added. These two issues will be handled separately in the following sections.

12.2 New Code Structure

Most abstract instructions are equivalent to their bytecode counterparts. There is one big difference though. The abstract instructions utilize continuations, which are implemented as linked lists directly referencing the following instruction. See figure 12.1 where the features trueL, falseL and errorL are continuations to corresponding paths.
Figure 12.1: The structure of the \texttt{aTestBool} abstract instruction, showing the three different paths forking from it.

The bytecode on the other hand is a sequential list of instructions, where the continuation of one instruction is the next instruction in the list, or in case of a branch, the corresponding label of that branch. There is no explicit reference to the first instruction of the continuation.

Like in the previous phases of the back end, instructions with more than one possible continuation needs to be handled with extra care. Let us once again take the \texttt{aTestBool} instruction as an example (figures 8.4 and 12.1). The bytecode counterpart called \texttt{testBool}, has three features; the register that is to be tested, the label to branch to if that register has the value \texttt{false} and the label to branch to if the register did not contain a boolean value. If the register contains the value \texttt{true}, the instruction following \texttt{testBool} will be executed.
The translation starts with the true path. At the end of the instruction list that is generated from this path, we add a branch instruction and a new label. The branch will later be patched with a label marking the end of the whole testBool instruction. The label that is added to the end of the instruction list will be patched into the testBool instruction, where it marks where the execution will continue if the test turns out to be false. Finally the error path is translated in the same way. Figure 12.2 illustrates the result of transforming an atestBool instruction into bytecode.

All execution paths of a testBool instruction (except the last) will end with a branch instruction (remember that if this is a nested instruction there may be an arbitrary number of execution paths sharing the same continuation). This is handled by the aShared instruction, which keeps a counter of how many times it has been visited. At the first visit, a new label index is generated and saved in the aShared instructions data structure. For all but the last visit, a branch instruction to this label is returned. At the last visit (which also might be the first), the label instruction itself is generated, with the continuation of the aShared node translated to bytecode following the label.
12.3 Register Handling

Before a register can be assigned or read, it must be allocated. The previous phase, the register allocation phase, annotates the abstract instructions with information that the code emitter needs in order to allocate registers in the right place. It annotates aDefinition and aDefinitionCopy instructions with the size of the local environment of that definition. The local environment is in principle the number of Y-registers used. The emitter then appends a bytecode instruction that allocate the correct number of Y-registers.

X-registers on the other hand can be allocated anywhere in a definition. Some instructions, like the definition and inlineAt instructions, implicitly allocate X-registers. For other instructions the register allocator marks X-register used for the first time with a 'f', for example xf(5) instead of x(5). Therefore the emitter must, before it can replace any abstract instruction, check whether it contains any registers with that marking. If it does, a createVariable instruction is first generated for that register index.

The register allocator also annotates instructions with any moves between registers that might be needed before that instruction. These move instructions may be needed out of several reasons. Before a call, the parameters need to be in the right X-registers and all other X-registers needs to be saved to Y-registers. Other instructions also put constraints on the registers used, usually there is the constraint that G-registers can not be used. All constraints are resolved in the register allocator and the emitter simply emits the move instructions.
Chapter 13

Assembling

The assembling phase of the back end consists of three parts. First the peephole optimization processes the code. After that the symbolic labels in the code is replaced by absolute addresses, and finally the code is loaded. These phases can be executed by calling them separately or by calling the procedure \texttt{Assemble}, which then calls the different subphases and returns an executable program.

13.1 Peephole Optimization

Peephole optimization is carried out by inspecting one or a few instructions at a time, searching for segments of code that can be optimized. All optimization is done locally except when trying to optimize branches, in which case the branches are followed. This optimization is also carried out in this phase simply because it is much easier to perform on bytecode rather than on abstract instructions.

There are basically four different categories of optimizations:

\textbf{Unreachable-code elimination} removes code that cannot possibly be executed because there is no way no reach it. After for example a branch, all instructions until the next used label can be removed (if no branch is targeted there). To know which labels are actually used, a dictionary\footnote{A dictionary is a special kind of data structure, indexed by a key that is an atom. This is implemented using hash tables.} is kept, where the indices of the labels are saved, for all branch instructions.

\textbf{Instruction combining} replace single instructions or sequences of instructions by single ones that perform the same task faster. These optimizations constitute a large part of the Peephole procedure. When one single instruction is replaced, it is usually some sort of specialization,
like replacing a `call BI` (call built in) instruction with an `inlinePlus` instruction, which is a specialized add instruction, where possible. Replacing two consecutive `move` instructions into one `moveMove` is one example of combining instructions.

**Tail-call optimization.** This is a very important optimization, especially for the Oz language. The reason it is so important is that Oz implements loops via recursive calls. All calls at the end of a procedure can be replaced by tail calls, thus it is merely a task of replacing sequences like

```plaintext
call(Reg Arity)
deAllocate(N)
return

with

deAllocate(N)
tailCall(Reg Arity)
```

ordinary `call` can take any type of register as destination whereas a `tailCall` require a X-register. This is resolved by simply moving from the original register to the first X-register that is not a parameter (which is the X-register with the same index as the arity of the call procedure). This method must be modified if the “Call Convention” optimization is implemented, in which case it is not certain which registers are actually used for parameters.

**Branch optimization.** Everywhere we make a branch, either from an explicit branch instruction or from a match or one of the test instructions, we follow the branch to the corresponding label and inspect the first instruction. Should this instruction be another branch, we can continue to follow this branch until we reach a label that is not followed by a branch. Then we simply replace the first label index with the index of this new label and have thus removed one ore more branches. This might also lead to unreachable code, which can be eliminated later on. If we start this tracing with a branch (and not a match or test instruction) and end at a `deAllocate(X) | return` sequence, the branch instruction can be replaced by that sequence, which results in faster, although larger, code.

### 13.2 Absolute Addressing

This is where we replace the symbolic labels with absolute addresses. Starting from the top, we recurse through the code and for each instruction add
the size of that particular instruction to a code size counter. If we find an instruction that uses a label index that we have not yet seen, we replace that index with an, so far, unbound variable. This variable is then placed in a dictionary with the label index as key. Should the same label index be used later on, then this occurrence will also be replaced with the same unbound variable. When finally the label with that index is encountered, the variable will be bound to the value of the code size counter. The label itself is never added to the resulting code. The code size is needed when loading, so the final value of the counter is also return at the end.

13.3 Loading

This phase relies heavily on procedures found in the file CompilerSupport.oz, which is supplied alongside other system files when installing the Mozart compiler. The Load procedure itself merely allocates a chunk of memory, via the allocateCodeBlock procedure, and then stores each instruction in that chunk. This is done by first storing the op-code of the instruction and then storing whatever arguments the instruction uses, via different procedures calls (storeInt, storeXRegisterIndex and so on). All store procedures are supplied in the CompilerSupport.oz file. The only instruction that needs some work is the match instruction. It uses a list for the cases, this list is transformed to a hash-table before storing it.
Part III

Future Work
Chapter 14

Future Work

14.1 Redesign

It is often the case when one is working on a large project, that it is only at
the end of the project that you have fully understood every part of it. Very
often one would like to start all over again and do every thing right this time.
Certainly there are a number of things that we would have done differently
if we had known when we started what we know now. One advantage of the
current implementation is that the backend is more modular now, making it
easier to change parts of it without affecting the rest of the compiler.

14.1.1 Changes to the Register Allocation Phase

One such part that we plan to change is the register allocation phase. This
phase is extremely important. Bad allocations will not only use more regis-
ters than needed, but also since some instructions, like for example call
instructions, have restrictions on which register to use, it will also lead to
unnecessary move instructions. Already at the start of the project, we dis-
cussed using basic blocks for register allocation. Should we choose to do so,
we would have to make some major changes to the register allocator. With
this type of representation, you first try to solve constraints between basic
blocks and are then left with the more trivial task of allocating sequential
code inside the blocks. Something that would have an even larger effect on
this phase is if we would implement the “Cheap Call” and “Call Convention”
optimizations discussed previously. Where those optimizations are applied,
register constraints associated with call instructions may be relaxed or at
best completely removed, which of course will strongly affect the allocation
of the registers.
14.1.2 Calculate Liveness

Instead of ordinary liveness we decided to only calculate kill-sets, which was sufficient for our register allocator. However, there are now plans to change the garbage collection in the Mozart virtual machine — at present, it is only possible to garbage collect X-registers. The plans are to make all types of registers garbage collectable. This will need support from the compiler, because annotations of the code with liveness information and hence kill-sets will no longer be sufficient. Though the actual calculation of liveness instead of kill-sets does not demand any major change, the new instruction format that this leads to will of course effect all the following phases.

14.2 Compiler Efficiency

In almost every phase of the backend, we manipulate features and/or values of features in the records that represent the instructions. We do that via the different \texttt{adjoin} and \texttt{subtract} procedures available, which we found suitable. We later found out that they were quite time consuming, and could easily be replaced by more static techniques. An even bigger problem though, is that since records are stateless entities you can not change the features or values of them. Instead you make a new record a copy of the old but with the changed feature. If we consider the abstract instruction format, the previous instruction (that is, whose continuation is the changed instruction) must also be modified so that it references the new record. Since this reference is itself a stateless feature, we are forced to make a new record for that instruction too, and so on. This means that for every pass, we must construct a completely new instruction tree. Even if we want to change only one single feature in one record, we have to build a completely new tree.

The remedy to this is of course to make the records stateful. There is in fact an instruction that already has a feature bound to a stateful cell. The \texttt{info} feature of the \texttt{aShared} instruction is used to store information that may need to be changed several times in one single pass. Since this feature is bound to a cell, its value can be updated without creating a new record. We have planned to expand this solution and add a dictionary to each record. This dictionary will then contain all those features whose values may come to changes during the different phases. Thus updating, adding or removing a feature will not make it necessary to create a new record, meaning that the tree that is built in the code generation phase can be used during the whole backend.
14.3 Optimizations

There are a number of optimizations that we would like to implement and evaluate. However, the evaluation part should not be neglected. An optimization that looks promising on paper will not always give the expected result. Benchmarks should be used to verify the effect of the optimization.

As mentioned in Chapter 7, we removed Delayed Initialization from the register allocation phase. This is one optimization that definitely should be implemented again in a separate phase. The other optimizations mentioned in that chapter (inlining, “Cheep Call” and “Call Convention”) should also be considered, but are not as urgent. Of course there are a lot of optimizations in the literature that is worth to try. The fact that there now exists an explicit intermediate code format makes it a lot easier to experiment with implementing different optimizations and evaluate their usefulness.

14.4 Bugs

The compiler of course has to be cleaned from bugs, as there still exists some, mainly in the generation of instructions for object orientation as described in Section 2.4. Debug information is not present either, and this has to be corrected.
Part IV

Appendices
Appendix A

Interface to the Back-end Phases

This appendix describes the interface between the different phases in detail.

A.1 General Notes

*Emphasized* features in the following description denominates features not existing in all phases, used for passing data between intermediate phases.

A.2 Abstract Code Generation

**Requires**  Object Graph format.

**Notes**  The Object Graph format is described in Appendix C.

**Produces**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>aDefinition</td>
<td>vreg varname arity gregs code id</td>
</tr>
<tr>
<td></td>
<td>cont coord lowbd highbd</td>
</tr>
<tr>
<td>aDefinitionCopy</td>
<td>vreg varname arity gregs code id</td>
</tr>
<tr>
<td></td>
<td>cont coord lowbd highbd</td>
</tr>
<tr>
<td>aCall</td>
<td>vreg args info cont coord</td>
</tr>
<tr>
<td>aCallBuiltin</td>
<td>args name coord cont</td>
</tr>
<tr>
<td>aEquateConstant</td>
<td>vreg constant cont</td>
</tr>
<tr>
<td>aEquateRecord</td>
<td>vreg label arity args coord cont</td>
</tr>
<tr>
<td>aExHandler</td>
<td>exReg tryL catchL coord</td>
</tr>
<tr>
<td>aGetSelf</td>
<td>1 coord cont</td>
</tr>
<tr>
<td>aGetVariable</td>
<td>1 coord cont</td>
</tr>
<tr>
<td>Instruction</td>
<td>Features</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>aInlineAssign</td>
<td>attribute value coord cont</td>
</tr>
<tr>
<td>aInlineAt</td>
<td>attribute result coord cont</td>
</tr>
<tr>
<td>aInlineDot</td>
<td>vreg result feature coord cont</td>
</tr>
<tr>
<td>aLockEnd</td>
<td>coord cont</td>
</tr>
<tr>
<td>aLockThread</td>
<td>label lockReg coord cont</td>
</tr>
<tr>
<td>aMatch</td>
<td>vreg nomatch entries coord</td>
</tr>
<tr>
<td>aPopEx</td>
<td>coord cont</td>
</tr>
<tr>
<td>aSetSelf</td>
<td>1 coord cont</td>
</tr>
<tr>
<td>aShared</td>
<td>paths info coord cont</td>
</tr>
<tr>
<td>aTestBool</td>
<td>vreg trueL falseL errorL coord</td>
</tr>
<tr>
<td>aUnify</td>
<td>1 2 coord cont</td>
</tr>
</tbody>
</table>

Notes  The feature info of aShared contains phase-specific information - it’s only used internally in the different phases of the back-end. The abstract instructions are described in detail in Appendix D.

A.3 Liveness Analysis

Requires

<table>
<thead>
<tr>
<th>Abstract Instructions required by Liveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>aDefinition</td>
</tr>
<tr>
<td>aDefinitionCopy</td>
</tr>
<tr>
<td>aCall</td>
</tr>
<tr>
<td>aCallBuiltin</td>
</tr>
<tr>
<td>aEquateConstant</td>
</tr>
<tr>
<td>aEquateRecord</td>
</tr>
<tr>
<td>aExHandler</td>
</tr>
<tr>
<td>aGetSelf</td>
</tr>
<tr>
<td>aGetVariable</td>
</tr>
<tr>
<td>aInlineAssign</td>
</tr>
<tr>
<td>aInlineAt</td>
</tr>
<tr>
<td>aInlineDot</td>
</tr>
<tr>
<td>aLockEnd</td>
</tr>
<tr>
<td>aLockThread</td>
</tr>
<tr>
<td>aMatch</td>
</tr>
<tr>
<td>aPopEx</td>
</tr>
<tr>
<td>aReorganize</td>
</tr>
<tr>
<td>aSetSelf</td>
</tr>
<tr>
<td>aShared</td>
</tr>
<tr>
<td>aTestBool</td>
</tr>
<tr>
<td>aUnify</td>
</tr>
</tbody>
</table>
Notes

Produces

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>aDefinition</td>
<td>vreg gregs code cont lowbd highbd kill</td>
</tr>
<tr>
<td>aDefinitionCopy</td>
<td>vreg gregs code cont lowbd highbd kill</td>
</tr>
<tr>
<td>aCall</td>
<td>vreg args cont kill</td>
</tr>
<tr>
<td>aCallBuiltin</td>
<td>args cont kill</td>
</tr>
<tr>
<td>aEquateConstant</td>
<td>vreg cont kill</td>
</tr>
<tr>
<td>aEquateRecord</td>
<td>vreg args cont kill</td>
</tr>
<tr>
<td>aExHandler</td>
<td>exReg tryL catchL tryKill kill</td>
</tr>
<tr>
<td>aGetSelf</td>
<td>1 cont kill</td>
</tr>
<tr>
<td>aGetVariable</td>
<td>1 cont kill</td>
</tr>
<tr>
<td>aInlineAssign</td>
<td>value cont kill</td>
</tr>
<tr>
<td>aInlineAt</td>
<td>result cont kill</td>
</tr>
<tr>
<td>aInlineDot</td>
<td>vreg result cont kill</td>
</tr>
<tr>
<td>aLockEnd</td>
<td>cont kill</td>
</tr>
<tr>
<td>aLockThread</td>
<td>lockReg cont kill</td>
</tr>
<tr>
<td>aMatch</td>
<td>vreg nomatch entries kill</td>
</tr>
<tr>
<td>aPopEx</td>
<td>cont kill</td>
</tr>
<tr>
<td>aSetSelf</td>
<td>1 cont kill</td>
</tr>
<tr>
<td>aShared</td>
<td>info cont kill</td>
</tr>
<tr>
<td>aTestBool</td>
<td>vreg trueL falseL errorL trueKill falseKill kill</td>
</tr>
<tr>
<td>aUnify</td>
<td>1 2 cont kill</td>
</tr>
</tbody>
</table>

Notes

- For aTestBool, aMatch and aExHandler, the kill feature represents the kill set for the error path of the instruction.

- The instructions without register usage have kill sets, but this will of course indicate that no variables dies after this instruction.

The abstract instructions are described in detail in Appendix D.

A.4 Register Allocation
### Abstract Instructions required by Register Allocation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>aDefinition</td>
<td>vreg gregs code cont arity lowbd kill</td>
</tr>
<tr>
<td>aDefinitionCopy</td>
<td>vreg gregs code cont arity lowbd kill</td>
</tr>
<tr>
<td>aCall</td>
<td>vreg args cont kill</td>
</tr>
<tr>
<td>aCallBuiltin</td>
<td>name args cont kill</td>
</tr>
<tr>
<td>aEquateConstant</td>
<td>vreg cont kill</td>
</tr>
<tr>
<td>aEquateRecord</td>
<td>vreg args cont kill</td>
</tr>
<tr>
<td>aExHandler</td>
<td>exReg tryL catchL tryKill kill</td>
</tr>
<tr>
<td>aGetSelf</td>
<td>1 cont kill</td>
</tr>
<tr>
<td>aGetVariable</td>
<td>1 cont kill</td>
</tr>
<tr>
<td>aInlineAssign</td>
<td>value cont kill</td>
</tr>
<tr>
<td>aInlineAt</td>
<td>result cont kill</td>
</tr>
<tr>
<td>aInlineDot</td>
<td>vreg result cont kill</td>
</tr>
<tr>
<td>aLockEnd</td>
<td>cont kill</td>
</tr>
<tr>
<td>aLockThread</td>
<td>lockReg cont kill</td>
</tr>
<tr>
<td>aMatch</td>
<td>vreg nomatch entries kill</td>
</tr>
<tr>
<td>aPopEx</td>
<td>cont kill</td>
</tr>
<tr>
<td>aSetSelf</td>
<td>1 cont kill</td>
</tr>
<tr>
<td>aShared</td>
<td>info cont kill</td>
</tr>
<tr>
<td>aTestBool</td>
<td>vreg trueL falseL errorL trueKill falseKill kill</td>
</tr>
<tr>
<td>aUnify</td>
<td>1 2 cont kill</td>
</tr>
</tbody>
</table>

### Requires

**Notes**  The abstract instructions are described in detail in Appendix D.

### Produces

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>aDefinition</td>
<td>vreg gregs code cont arity coord id enusize max</td>
</tr>
<tr>
<td>aDefinitionCopy</td>
<td>vreg gregs code cont arity coord id lowbd highbd kill</td>
</tr>
<tr>
<td>aCall</td>
<td>vreg args cont coord moves</td>
</tr>
<tr>
<td>aCallBuiltin</td>
<td>name args cont coord gregs</td>
</tr>
<tr>
<td>aEquateConstant</td>
<td>vreg cont coord</td>
</tr>
<tr>
<td>aEquateRecord</td>
<td>vreg args cont coord</td>
</tr>
<tr>
<td>aExHandler</td>
<td>exReg tryL catchL coord</td>
</tr>
<tr>
<td>aGetSelf</td>
<td>1 cont coord</td>
</tr>
</tbody>
</table>
### A.5 Bytecode Emission

#### Requires

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>aDefinition</td>
<td>vreg gregs code cont arity coord id</td>
</tr>
<tr>
<td>aDefinitionCopy</td>
<td>vreg gregs code cont arity coord id</td>
</tr>
<tr>
<td>aCall</td>
<td>vreg args cont coord moves</td>
</tr>
<tr>
<td>aCallBuiltin</td>
<td>name args cont coord yregs</td>
</tr>
<tr>
<td>aEquateConstant</td>
<td>vreg cont coord</td>
</tr>
<tr>
<td>aEquateRecord</td>
<td>vreg args cont coord</td>
</tr>
<tr>
<td>aExceptionHandler</td>
<td>exReg tryL catchL coord</td>
</tr>
<tr>
<td>aGetSelf</td>
<td>1 cont coord</td>
</tr>
<tr>
<td>aGetVariable</td>
<td>1 cont coord</td>
</tr>
<tr>
<td>aInlineAssign</td>
<td>value cont coord</td>
</tr>
<tr>
<td>aInlineAt</td>
<td>result cont coord</td>
</tr>
<tr>
<td>aInlineDot</td>
<td>vreg result cont coord</td>
</tr>
<tr>
<td>aLockEnd</td>
<td>cont coord</td>
</tr>
<tr>
<td>aLockThread</td>
<td>lockReg cont coord</td>
</tr>
<tr>
<td>aMatch</td>
<td>vreg nomatch entries coord</td>
</tr>
<tr>
<td>aPopEx</td>
<td>cont coord</td>
</tr>
<tr>
<td>aReorganize</td>
<td>list tail cont coord kill</td>
</tr>
<tr>
<td>aSetSelf</td>
<td>1 cont coord</td>
</tr>
<tr>
<td>aShared</td>
<td>info cont coord</td>
</tr>
<tr>
<td>aTestBool</td>
<td>vreg trueL falseL errorL coord</td>
</tr>
<tr>
<td>aUnify</td>
<td>1 2 cont coord</td>
</tr>
</tbody>
</table>

**Notes** The abstract instructions are described in detail in Appendix D.
aShared  info  cont  coord
aTestBool vreg trueL  falseL  errorL  coord
aUnify  1 2  cont  coord

Notes The abstract instructions are described in detail in Appendix D.

Produces Relative bytecode.

Notes The relative bytecode is described in Appendix E, and in [23].

A.6 Peephole Optimization

Requires Relative bytecode.

Produces Relative bytecode.

Notes The relative bytecode is described in Appendix E, and in [23].

A.7 Absolute Code Generation

Requires Absolute bytecode.

Produces Absolute bytecode.

A.8 Loading

Requires Absolute bytecode.

Produces Program.
Appendix B

Syntax Tree Format

B.1 Compilation

B.1.1 Input

\[ \text{<input> ::= parseError} \]
\[ \quad \mid \text{<(compilation unit)>} \]

B.1.2 Compilation Units

\[ \text{<compilation unit> ::= <phrase>} \]
\[ \quad \mid <\text{directive}> \]
\[ \quad \mid \text{fDeclare(<phrase> <phrase> <coord>)} \]

\[ \text{<directive> ::= dirSwitch([<switch>])} \]
\[ \quad \mid \text{dirPushSwitches} \]
\[ \quad \mid \text{dirPopSwitches} \]
\[ \quad \mid \text{dirLocalSwitches} \]

\[ \text{<switch> ::= on(<switch name> <coord>)} \]
\[ \quad \mid \text{off(<switch name> <coord>)} \]

\[ \text{<switch name> ::= <atom>} \]

B.2 The Base Language

B.2.1 Phrases

At the syntactical level, statements can in general not be distinguished from expressions. Both are subsumed by <phrase>.

\[ \text{<phrase> ::= fStepPoint(<phrase> <atom> <coord>)} \]
\[ \quad \mid \text{fAnd(<phrase> <phrase>)} \]

107
APPENDIX B. SYNTAX TREE FORMAT

\[ \text{fEq}(\text{phrase} \ (\text{phrase}) \ (\text{coord})) \]
\[ \text{fAssign}(\text{phrase} \ (\text{phrase}) \ (\text{coord})) \]
\[ \text{fOrElse}(\text{phrase} \ (\text{phrase}) \ (\text{coord})) \]
\[ \text{fAndThen}(\text{phrase} \ (\text{phrase}) \ (\text{coord})) \]
\[ \text{fOpApply}(\text{atom} \ [[\text{phrase}]] \ (\text{coord})) \]
\[ \text{fOpApplyStatement}(\text{atom} \ [[\text{phrase}]] \ (\text{coord})) \]
\[ \text{fObjApply}(\text{phrase} \ (\text{phrase}) \ (\text{coord})) \]
\[ \text{fAt}(\text{phrase} \ (\text{coord})) \]
\[ (\text{atom literal}) \]
\[ (\text{variable}) \]
\[ (\text{wildcard}) \]
\[ \text{fSelf}(\text{coord}) \]
\[ \text{fDollar}(\text{coord}) \]
\[ (\text{int literal}) \]
\[ \text{fFloat}(\text{float} \ (\text{coord})) \]
\[ \text{fRecord}(\text{label} \ [(\text{record arg})]) \]
\[ \text{fOpenRecord}(\text{label} \ [(\text{record arg})]) \]
\[ \text{fApply}(\text{phrase} \ [[\text{phrase}]] \ (\text{coord})) \]
\[ \text{fProc}(\text{phrase} \ [[\text{phrase}]] \ (\text{phrase}) \]
\[ [(\text{proc flag})] \ (\text{coord})] \]
\[ \text{fFun}(\text{phrase} \ [[\text{phrase}]] \ (\text{phrase}) \]
\[ [(\text{proc flag})] \ (\text{coord})] \]
\[ \text{fFunctor}(\text{phrase} \ [[\text{functor descriptor}]] \]
\[ (\text{coord})] \]
\[ \text{fClass}(\text{phrase} \ [[\text{class descriptor}] \ [\text{meth}] \]
\[ (\text{coord})] \]
\[ \text{fLocal}(\text{phrase} \ (\text{phrase}) \ (\text{coord})) \]
\[ \text{fBoolCase}(\text{phrase} \ (\text{phrase}) \ (\text{opt else}) \ (\text{coord})) \]
\[ \text{fCase}(\text{phrase} \ [[\text{case clause}] \ (\text{opt else}) \]
\[ (\text{coord})] \]
\[ \text{fLockThen}(\text{phrase} \ (\text{phrase}) \ (\text{coord})) \]
\[ \text{fLock}(\text{phrase} \ (\text{coord})) \]
\[ \text{fThread}(\text{phrase} \ (\text{coord})) \]
\[ \text{fTry}(\text{phrase} \ (\text{catch}) \ (\text{finally}) \ (\text{coord})) \]
\[ \text{fRaise}(\text{phrase} \ (\text{coord})) \]
\[ \text{fSkip}(\text{coord}) \]

(\text{label}) ::= (\text{atom literal})
\[ | (\text{naked var}) \]

(\text{atom literal}) ::= \text{fAtom}(\text{literal} \ (\text{coord}))

(\text{naked var}) ::= \text{fVar}(\text{atom} \ (\text{coord}))

(\text{variable}) ::= (\text{naked var})
\[ | f\text{Escape}(\text{naked var} \ (\text{coord})) \]

\langle \text{wildcard} \rangle \ := \ f\text{Wildcard}(\langle \text{coord} \rangle)

\langle \text{int literal} \rangle \ := \ f\text{Int}(\langle \text{int} \ (\text{coord}) \rangle)

\langle \text{record arg} \rangle \ := \ \langle \text{phrase} \rangle
\quad | \ f\text{Colon}(\langle \text{feature} \ (\text{phrase}) \ (\text{coord}) \rangle)

For the moment, the only recognized flags are \text{instantiate, lazy, dynamic, and native.}

\langle \text{proc flag} \rangle \ := \ \langle \text{atom} \rangle

\textbf{B.2.2 Functors}

\langle \text{functor descriptor} \rangle \ := \ f\text{Require}(\langle [\text{import decl}] \ (\text{coord}) \rangle)
\quad | \ f\text{Prepare}(\langle \text{phrase} \ (\text{phrase}) \ (\text{coord}) \rangle)
\quad | \ f\text{Import}(\langle [\text{import decl}] \ (\text{coord}) \rangle)
\quad | \ f\text{Export}(\langle [\text{export decl}] \ (\text{coord}) \rangle)
\quad | \ f\text{Define}(\langle \text{phrase} \ (\text{phrase}) \ (\text{coord}) \rangle)

\langle \text{import decl} \rangle \ := \ f\text{ImportItem}(\langle \text{naked var} \ [\langle \text{aliased feature} \ (\text{opt import at}) \rangle \langle \text{opt import at} \rangle \ := \ f\text{NoImportAt}
\quad | \ f\text{ImportAt}(\langle \text{atom literal} \rangle)

\langle \text{export decl} \rangle \ := \ f\text{ExportItem}(\langle \text{export item} \rangle)

\langle \text{export item} \rangle \ := \ \langle \text{naked var} \rangle
\quad | \ f\text{Colon}(\langle \text{feature no var} \ (\text{naked var}) \rangle)

\textbf{B.2.3 Classes}

\langle \text{class descriptor} \rangle \ := \ f\text{From}(\langle [\text{phrase}] \ (\text{coord}) \rangle)
\quad | \ f\text{Prop}(\langle [\text{phrase}] \ (\text{coord}) \rangle)
\quad | \ f\text{Attr}(\langle [\text{attr or feat}] \ (\text{coord}) \rangle)
\quad | \ f\text{Feat}(\langle [\text{attr or feat}] \ (\text{coord}) \rangle)

\langle \text{attr or feat} \rangle \ := \ \langle \text{escaped feature} \rangle
\quad | \ \langle \text{escaped feature} \ #\langle \text{phrase} \rangle \rangle

\langle \text{meth} \rangle \ := \ f\text{Meth}(\langle \text{meth head} \ (\text{phrase}) \ (\text{coord}) \rangle)
\begin{itemize}
\item \textbf{Basic Syntax}
\item \textbf{Feature Syntax}
\item \textbf{Case Clause Syntax}
\item \textbf{Pattern Syntax}
\item \textbf{Catch Syntax}
\item \textbf{Finally Syntax}
\end{itemize}
\langle\text{opt\ else}\rangle \ ::= \ f\text{NoElse}\langle\langle\text{coord}\rangle\rangle
\ |
\langle\text{phrase}\rangle

B.2.6 Coordinates
Each triple consisting of an \langle\text{atom}\rangle and two \langle\text{int}\rangle s denotes a file name ('' if none known), a line number (starting at 1; required) and a column number (starting at 0; 1 if none known). If two triples are given, then they denote the starting and ending coordinates of a construct. A pos may be turned into a fineStep or a coarseStep, denoting a step point for debugging. unit is an unknown coordinate.
\langle\text{coord}\rangle \ ::= \ \text{pos}\langle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\rangle
\ |
\text{pos}\langle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\rangle
\ |
\text{fineStep}\langle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\rangle
\ |
\text{fineStep}\langle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\rangle
\ |
\text{coarseStep}\langle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\rangle
\ |
\text{coarseStep}\langle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\langle\text{atom}\rangle\langle\text{int}\rangle\langle\text{int}\rangle\rangle
\ |
\text{unit}

B.3 Finite Domain Extensions and Combinators
\langle\text{phrase}\rangle \ ::= \ \langle\text{fd\ expr}\rangle
\ |
\text{f\text{Fail}}\langle\langle\text{coord}\rangle\rangle
\ |
\text{f\text{Not}}\langle\langle\text{phrase}\rangle\langle\text{coord}\rangle\rangle
\ |
\text{f\text{Cond}}\langle\langle\text{clause}\rangle\rangle\langle\text{opt\ else}\rangle\langle\text{coord}\rangle\rangle
\ |
\text{f\text{Or}}\langle\langle\text{clause\ opt\ then}\rangle\rangle\langle\text{coord}\rangle\rangle
\ |
\text{f\text{Dis}}\langle\langle\text{clause\ opt\ then}\rangle\rangle\langle\text{coord}\rangle\rangle
\ |
\text{f\text{Choice}}\langle\langle\text{phrase}\rangle\rangle\langle\text{coord}\rangle\rangle

\langle\text{fd\ expr}\rangle \ ::= \ \text{f\text{DCompare}}\langle\langle\text{atom}\rangle\langle\text{phrase}\rangle\langle\text{phrase}\rangle\langle\text{coord}\rangle\rangle
\ |
\text{f\text{DIn}}\langle\langle\text{atom}\rangle\langle\text{phrase}\rangle\langle\text{phrase}\rangle\langle\text{coord}\rangle\rangle
\langle\text{clause}\rangle \ ::= \ \text{f\text{Clause}}\langle\langle\text{phrase}\rangle\langle\text{phrase}\rangle\langle\text{phrase}\rangle\rangle
\langle\text{clause\ opt\ then}\rangle \ ::= \ \text{f\text{Clause}}\langle\langle\text{phrase}\rangle\langle\text{phrase}\rangle\langle\text{opt\ then}\rangle\rangle
\langle\text{opt\ then}\rangle \ ::= \ \text{f\text{NoThen}}\langle\langle\text{coord}\rangle\rangle
\ |
\langle\text{phrase}\rangle

B.4 Gump Extensions
\langle\text{compilation\ unit}\rangle \ ::= \ \text{f\text{SynTopLevelProductionTemplates}}\langle\rangle
APPENDIX B. SYNTAX TREE FORMAT

([prod clause])

⟨phrase⟩ ::= fScanner(⟨naked var⟩ [⟨class descriptor⟩]
  [⟨meth⟩] [⟨scanner rule⟩] ⟨atom⟩ ⟨coord⟩)
  | fparser(⟨naked var⟩ [⟨class descriptor⟩]
    [⟨meth⟩] ⟨token clause⟩ [⟨parser descriptor⟩]
    ⟨int⟩ ⟨coord⟩)

⟨grammar symbol⟩ ::= ⟨atom literal⟩
  | ⟨naked var⟩

B.4.1 Scanners

⟨scanner rule⟩ ::= fMode(⟨naked var⟩ [⟨mode descriptor⟩])
  | ⟨lex clause⟩

⟨mode descriptor⟩ ::= fInheritedModes([⟨naked var⟩])
  | ⟨lex clause⟩

⟨lex clause⟩ ::= fLexicalAbbreviation(⟨grammar symbol⟩
  ⟨regex⟩)
  | fLexicalRule(⟨regex⟩ ⟨phrase⟩)

⟨regex⟩ ::= ⟨string⟩

B.4.2 Parsers

⟨token clause⟩ ::= fToken([⟨token decl⟩])

⟨token decl⟩ ::= ⟨atom literal⟩
  | ⟨atom literal⟩#⟨phrase⟩

⟨parser descriptor⟩ ::= ⟨prod clause⟩
  | ⟨syntax rule⟩

⟨prod clause⟩ ::= fProductionTemplate(⟨prod key⟩
  [⟨prod param⟩] [⟨syntax rule⟩]
  [⟨syn expr⟩] [⟨prod ret⟩])

⟨prod param⟩ ::= ⟨naked var⟩
  | ⟨wildcard⟩

⟨prod key⟩ ::= none#⟨string⟩
  | ⟨atom⟩#⟨string⟩

⟨prod ret⟩ ::= none
  | ⟨naked var⟩
\begin{verbatim}
\appendix
\section*{Syntax Tree Format}

\begin{verbatim}
\begin{align*}
\langle \text{syn rule} \rangle & \quad ::= \quad \text{fSyntaxRule}(\langle \text{grammar symbol} \rangle \ [\langle \text{syn formal} \rangle]) \\
\langle \text{syn formal} \rangle & \quad ::= \quad \langle \text{naked var} \rangle \\
& \quad \mid \quad \langle \text{wildcard} \rangle \\
& \quad \mid \quad \text{fDollar}(\langle \text{coord} \rangle)
\end{align*}
\end{verbatim}

\begin{verbatim}
\begin{align*}
\langle \text{syn expr} \rangle & \quad ::= \quad \text{fSynApplication}(\langle \text{grammar symbol} \rangle \ [\langle \text{phrase} \rangle]) \\
& \quad \mid \quad \text{fSynAction}(\langle \text{phrase} \rangle) \\
& \quad \mid \quad \text{fSynSequence}(\langle \langle \text{naked var} \rangle \ [\langle \text{syn expr} \rangle] \rangle) \\
& \quad \mid \quad \text{fSynAlternative}(\langle \langle \text{syn expr} \rangle \rangle) \\
& \quad \mid \quad \text{fSynAssignment}(\langle \text{variable} \rangle \ [\langle \text{syn expr} \rangle]) \\
& \quad \mid \quad \text{fSynTemplateInstantiation}(\langle \text{prod key} \rangle \ [\langle \text{syn expr} \rangle \ [\langle \text{coord} \rangle])
\end{align*}
\end{verbatim}
\end{verbatim}
\end{appendix}

\end{verbatim}
Appendix C

The Object Graph

C.1 Oz Core Language: Graph Representation

This appendix was constructed by the authors from developer notes, probably written by Leif Kornstaedt.

C.1.1 Abbreviations

\[
\begin{align*}
\langle V \rangle & ::= \text{VARIABLE} \\
\langle O \rangle & ::= \text{VARIABLEOccurrence} \\
\text{coordinates} \\
\langle C \rangle & ::= \ldots \quad (\text{same as in FSyntax}) \\
\text{terms} \\
\langle T \rangle & ::= \text{ValueNode} \\
& \quad | \text{Construction} \\
& \quad | \text{O} \\
\text{patterns} \\
\langle P \rangle & ::= \text{ValueNode} \\
& \quad | \text{RecordPattern} \\
& \quad | \text{EquationPattern} \\
& \quad | \text{PatternVariableOccurrence} \\
& \quad | \text{O} \\
\text{labels} \\
\langle L \rangle & ::= \text{ValueNode} \quad (\text{ValueNode must be an atom}) \\
& \quad | \text{O} \\
\text{features} \\
\langle F \rangle & ::= \text{ValueNode} \quad (\text{ValueNode must be a feature}) \\
& \quad | \text{O} \\
\text{artities} \\
\langle A \rangle & ::= \text{int}
\end{align*}
\]
| F+  
| attr/feat identifier and optional initializer  
\( (l) \) ::= F  
| F#O  

method formal  
\( (M) \) ::= MethodFormal  

statements  
\( (S) \) ::= Statement

### C.1.2 Class Descriptions

**class** Statement  
coord: C  
next: S singly linked list terminated by self reference  

**Note:** never instantiated

**class** TypeOf extends Statement  
arg: V  
res: O  

**Note:**  
res = ... typeof arg

**class** StepPoint extends Statement  
kind: atom  
statements: S+  

**Note:**  
statements

**class** Declaration extends Statement  
localVars: V+  
statements: S+  

**Note:**  
'local' localVars 'in' statements 'end'

**class** SkipNode extends Statement  
::  

**Note:**  
'skip' 'skip'
class Equation extends Statement
left:  O
right: T
Note:
left `=’ right

class Construction
label: L
args: (T|F#T)+ all featureless subtrees in front
Note:
label `( args [ ‘...’ ] ’)

class Definition extends Statement
designator: O
formalArgs: V*
statements: S+
isStateUsing: bool
procFlags: [atom]
toCopy: [(name|foreignPointer)] | unit
globalVars: V*
those of "local formalArgs in statements end"
Note:
’proc’ procFlags ’’ designator formalArgs ’’
statements ’end’

class ClauseBody extends Definition
:
Note:
’proc ’’ designator formalArgs ’’ statements ’end’

class Application extends Statement
designator: O
actualArgs: O*   
Note:
’’ designator actualArgs ’’


**class IfNode extends Statement**

*arbiter:* \( O \)

*consequent:* \( \text{IfClause} \)

*alternative:* \( \text{ElseNode | NoElse} \)

*global Vars:* \( V^* \)

*those of all clauses (not arbiter)*

**Note:**

'if' arbiter 'then' consequent alternative 'end'

---

**class IfClause**

*statements:* \( S^+ \)

*global Vars:* \( V^* \)

**Note:**

statements

---

**class PatternCase extends Statement**

*arbiter:* \( O \)

*clauses:* \( \text{PatternClause}^+ \)

*alternative:* \( \text{ElseNode | NoElse} \)

*global Vars:* \( V^* \)

*those of all clauses (not arbiter)*

**Note:**

'case' arbiter 'of' ( clause // '[]' ) alternative 'end'

---

**class PatternClause**

*local Vars:* \( V^* \)

*pattern:* \( \text{SideCondition | P} \)

*statements:* \( S^+ \)

*global Vars:* \( V^* \)

*those of statements and pattern*

**Note:**

[ local Vars 'in' ] pattern 'then' statements
### Class SideCondition

- **pattern**: P
- **localVars**: V*
- **statements**: S+
- **arbiter**: O

**Note:**
- pattern 'andthen' [ localVars 'in' ] statements
- arbiter

### Class RecordPattern

- **label**: L
- **args**: (P|F#P)+

**Note:**
- all featureless subpatterns in front
- isOpen: bool

**Note:**
- label '( args [ '...' ] )'

### Class EquationPattern

- **left**: PatternVariableOccurrence
- **right**: P
- **coord**: C

**Note:**
- left '=' right

### Class ElseNode

- **statements**: S+
- **globalVars**: V*

**Note:**
- 'else' statements

### Class NoElse

- **coord**: C

**Note:**
- empty
class TRYNode extends Statement
tryStatements: S+
exception: V
catchStatements: S+
globalVars: V*
those of tryStatements and catchStatements
Note:
'try' tryStatements 'catch' exception 'then'
catchStatements 'end'

class LockNode extends Statement
lockVar: O
statements: S+
Note:
'lock' lockVar 'then' statements 'end'

class ClassNode extends Statement
designator: O
parents: O*
properties: O*
attributes: I*
features: I*
methods: Method*
isVirtualToplevel: bool
Note:
'class' designator ['from' parents] ['prop'
properties] ['attr' attributes] ['feat' features] methods 'end'

class Method
label: L
formalArgs: M*
isOpen: bool
messageDesignator: V | unit
statements: S+
coord: C
globalVars: V*
only of "vars in label feats statements"
Note:
'meth' label '(' formalArgs ['...' ] ')’ ['=:
messageDesignator ] statements 'end'
```plaintext
class MethFormal
    feature: F
    arg: V
Note:
    feature '& arg

class MethFormalOptional extends MethFormal :
Note:
    feature '& arg '=""'

class MethFormalWithDefault extends MethFormal
default: O | unit
Note:
    feature '& arg '="" default

class ObjectLock extends Statement
    statements: S+
Note:
    'lock' statements 'end'

class GetSelf extends Statement
    destination: O
Note:
    destination '=' 'self'

class ExceptionNode extends Statement :
Note:
    'raise kernel(noElse ...) end'

class CondNode extends Statement
    clauses: Clause+
    alternative: ElseNode | NoElse
    global Vars: V*
Note:
    'cond' ( clause // ']' ) alternative 'end'
```
class ChoicesAndDisjunctions extends Statement
  clauses:  Clause+
  globalVars:  V*
Note:
  never instantiated

class OrNode extends ChoicesAndDisjunctions :
Note:
  'or' ( clause // '?' ) 'end'

class DisNode extends ChoicesAndDisjunctions :
Note:
  'dis' ( clause // ']' ) 'end'

class ChoiceNode extends ChoicesAndDisjunctions :
Note:
  'choice' ( clause // '?' ) 'end'

class Clause
  localVars:  V*
  guard:  S+
  kind:  'ask' | 'wait' | 'waitTop'
  statements:  S+
  globalVars:  V*  those of guard and statements
  guardGlobalVars:  V*  those of statements only
Note:
  [ localVars 'in' ] guard 'then' statements

class ValueNode
  value:  literal | number
  coord:  C
Note:
  value
### APPENDIX C. THE OBJECT GRAPH

<table>
<thead>
<tr>
<th>Class</th>
<th>Superclass</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td></td>
<td>coord: C, isTopLevel: bool, lastValue: unit</td>
</tr>
<tr>
<td>UserVariable</td>
<td>Variable</td>
<td>printName: atom, use: unused</td>
</tr>
<tr>
<td>RestrictedVariable</td>
<td>UserVariable</td>
<td>features: (atom</td>
</tr>
<tr>
<td>GeneratedVariable</td>
<td>Variable</td>
<td>origin: atom</td>
</tr>
<tr>
<td>VariableOccurrence</td>
<td></td>
<td>variable: V, value: O</td>
</tr>
<tr>
<td>RecordConstr</td>
<td></td>
<td>value: record, lastValue: unit</td>
</tr>
</tbody>
</table>

**Note:**
- `printName`
class PatternVariableOccurrence
extends VariableOccurrence
:
Note:
  variable

C.1.3 Remarks

Some core syntax expansions require shared references to the same code, such as when optimizing boolean conditionals with andthen/orelse in the guard, e.g.:

    if X orelse Y then S1 else S2 end

This is represented as

    proc {A} S1 end   % these are both instances
    proc {B} S2 end   % of <ClauseBody>
        if X then {A}
    else
        if Y then {A}
        else {B}
    end
end

The code generator will behave differently for ‘Definition’ and ‘ClauseBody’ and generate labels and branches instead of procedure definitions and applications.
Appendix D

Abstract Instructions

This appendix describes the different abstract instruction intermediate code format used within the redesigned Oz compiler. The abstract instructions are very much alike the original intermediate code format, the virtual instructions, but the abstract instructions has been improved in several ways and are treated differently within the compiler.

Much of the information in this appendix is taken from the original description of the masters’ thesis, see [22],

D.1 Summary of the Abstract Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>aDefinition</td>
<td>vreg varname arity gregs code id</td>
</tr>
<tr>
<td></td>
<td>cont coord lowbd highbd emusize maxx kill</td>
</tr>
<tr>
<td>aDefinitionCopy</td>
<td>vreg varname arity gregs code id</td>
</tr>
<tr>
<td></td>
<td>cont coord lowbd highbd emusize maxx kill</td>
</tr>
<tr>
<td>aCall</td>
<td>vreg args info cont coord moves</td>
</tr>
<tr>
<td></td>
<td>kill</td>
</tr>
<tr>
<td>aCallBuiltin</td>
<td>args name coord cont args yregs</td>
</tr>
<tr>
<td></td>
<td>kill</td>
</tr>
<tr>
<td>aEquateConstant</td>
<td>vreg constant cont kill</td>
</tr>
<tr>
<td>aEquateRecord</td>
<td>vreg label arity args coord cont</td>
</tr>
<tr>
<td></td>
<td>kill</td>
</tr>
<tr>
<td>aExHandler</td>
<td>exReg tryL catchL coord tryKill</td>
</tr>
<tr>
<td></td>
<td>kill</td>
</tr>
<tr>
<td>aGetSelf</td>
<td>1 coord cont kill</td>
</tr>
<tr>
<td>aGetVariable</td>
<td>1 coord cont kill</td>
</tr>
</tbody>
</table>
D.2 Acronyms

In the following description, some acronyms are used.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Integer</td>
</tr>
<tr>
<td>Entry</td>
<td>Hash Table Entry</td>
</tr>
</tbody>
</table>

D.3 Phases

<table>
<thead>
<tr>
<th>Phases Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
D.4 The Abstract Instructions in Detail

D.4.1 Procedure Definition

**aDefinition**

Defines a new procedure closure with procedure body in **code**, and puts the procedure in register **vreg**. The closure of the body is given in **regs**.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vreg</strong></td>
<td><strong>Reg</strong></td>
<td>12345</td>
<td>A register where the reference to the created procedure is stored. The register does not contain the reference to the procedure in its own body.</td>
</tr>
<tr>
<td><strong>varname</strong></td>
<td><strong>Atom</strong></td>
<td>12345</td>
<td>The name of the definition.</td>
</tr>
<tr>
<td><strong>id</strong></td>
<td><strong>PRef</strong></td>
<td>12345</td>
<td>Procedure Reference...</td>
</tr>
</tbody>
</table>
| **sited** | **Bool** | 12345 | ???
| **arity** | **Int** | 12345 | The arity of the procedure. A procedure without parameters has an arity of zero. |
| **regs** | **Reg** | 2345 | The closure of the body, given as a list of registers. If the given registers are \( R_0, R_1, \ldots, R_n \), the references to these registers becomes \( g(0), g(1), \ldots, g(n) \) in the procedure body. |
| **lowbnd** | **Int** | 2345 | The lowest used virtual register in this definition. |
| **highbnd** | **Int** | 2345 | The highest used virtual register in this definition. |
| **envsize** | **Int** | 45 | The size of the local environment, i.e. the number of Y-registers used in this definition. |
| **maxx** | **Int** | 45 | The X-register with the highest number is stored here. This number has to be provided to have Oz VM perform garbage collection correctly. |
### APPENDIX D. ABSTRACT INSTRUCTIONS

<table>
<thead>
<tr>
<th>code</th>
<th>AInstr</th>
<th>12345</th>
<th>Reference to the first instruction in the procedure body. The definition in considered to end where nil follows an instruction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the instruction following the complete definition.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
aDefinition(vreg:v(17) arity:2 gregs:nil code:aEquateConstant(...) coord:... cont:aCall(...))
```

defines a procedure closure in v(17), with arity 2, no external references, containing as its first instruction an aEquateConstant, and with an aCall following the definition.

### aDefinitionCopy

Defines a new procedure closure, and allows for some optimizations. The procedure body is in code, and the procedure in stored in register vreg. The closure of the body is given in gregs.

<table>
<thead>
<tr>
<th>ADefinitionCopy</th>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vreg</td>
<td>Reg</td>
<td></td>
<td>12345</td>
<td>A register where the reference to the created procedure is stored. The register does not contain the reference to the procedure in its own body.</td>
</tr>
<tr>
<td>varname</td>
<td>Atom</td>
<td></td>
<td>12345</td>
<td>The name of the definition.</td>
</tr>
<tr>
<td>id</td>
<td>PRef</td>
<td></td>
<td>12345</td>
<td>Procedure Reference...</td>
</tr>
<tr>
<td>sited</td>
<td>Bool</td>
<td></td>
<td>12345</td>
<td>???</td>
</tr>
<tr>
<td>arity</td>
<td>Int</td>
<td></td>
<td>12345</td>
<td>The arity of the procedure. A procedure without parameters has an arity of zero.</td>
</tr>
</tbody>
</table>
**APPENDIX D. ABSTRACT INSTRUCTIONS**

<table>
<thead>
<tr>
<th>gregs</th>
<th>Reg*</th>
<th>2345</th>
<th>The closure of the body, given as a list of registers. If the given registers are ( R_0, R_1, \ldots, R_n ), the references to these registers becomes ( g(0), g(1), \ldots, g(n) ) in the procedure body.</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowbnd</td>
<td>Int</td>
<td>2345</td>
<td>The lowest used virtual register in this definition.</td>
</tr>
<tr>
<td>highbnd</td>
<td>Int</td>
<td>2345</td>
<td>The highest used virtual register in this definition.</td>
</tr>
<tr>
<td>envsize</td>
<td>Int</td>
<td>45</td>
<td>The size of the local environment, i.e. the number of Y-registers used in this definition.</td>
</tr>
<tr>
<td>maxx</td>
<td>Int</td>
<td>45</td>
<td>The X-register with the highest number is stored here. This number has to be provided to have Oz VM perform garbage collection correctly.</td>
</tr>
<tr>
<td>code</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the first instruction in the procedure body. The definition in considered to end where \texttt{nil} follows an instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the instruction following the complete definition.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
aDefinitionCopy(vreg:v(17) arity:2 gregs:nil code:aEquateConstant(...) coord:... cont:aCall(...))
```

defines a procedure closure in \texttt{v(17)}, with arity 2, no external references, containing as its first instruction an \texttt{aEquateConstant}, and with an \texttt{aCall} following the definition.

### D.4.2 Procedure Application

**aCall**

Invokes a procedure pointed out by register \texttt{vreg}. In some circumstances, the instruction should be replaced by a
**aCall**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vreg</td>
<td>Reg</td>
<td>12345</td>
<td>The register that holds the procedure.</td>
</tr>
<tr>
<td>args</td>
<td>Reg*</td>
<td>12345</td>
<td>A list of registers holding the arguments.</td>
</tr>
<tr>
<td>info</td>
<td>CInfo</td>
<td>12345</td>
<td>Optional information, used to separate different call types.</td>
</tr>
<tr>
<td>moves</td>
<td>Move*</td>
<td>45</td>
<td>A list of moves that has to be emitted before the actual call instruction.</td>
</tr>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
aCall(vreg:unit args:[v(2) v(2)]
    info:constant ...)
```

**aCallBuiltin**

Invokes a built-in procedure.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>atom</td>
<td>12345</td>
<td>The name of the builtin, e.g. 'Value.byNeed' or 'Array.is'.</td>
</tr>
<tr>
<td>args</td>
<td>Reg*</td>
<td>123</td>
<td>A list of registers holding the arguments.</td>
</tr>
<tr>
<td>args</td>
<td>Tuple</td>
<td>45</td>
<td>A tuple with two lists of the in-arguments and the out-arguments.</td>
</tr>
<tr>
<td>yregs</td>
<td>Reg*</td>
<td>45</td>
<td>A list of register that are to be saved in Y-registers before the actual callBI instruction.</td>
</tr>
</tbody>
</table>
APPENDIX D. ABSTRACT INSTRUCTIONS

<table>
<thead>
<tr>
<th>cont</th>
<th>$AInstr$</th>
<th>12345</th>
<th>Reference to the next instruction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>$Coord$</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
aCallBuiltIn(name: 'Value.lessThan' args: [value(v(23) number(10))] ...)
```

D.4.3 Unification

Both equateConstant and equateRecord may operate in either read (unification) or write mode. The read/write mode is determined if we know the content of a register or not – e.g. if a register is non-local (either external or procedure parameter), unification is done (read-mode), but if it is the first occurrence of that particular register, we can optimize and create a data structure using write-mode instead.

It is always possible to transform a read-mode creation into a write-mode creation – first, we write the data structure (using the write-mode instructions) to a temporary register, and then we do a general unification (aUnify) with the temporary register and the target register.

**aEquateConstant**

Equate a register with a constant.

```
@EquateConstant

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vreg</td>
<td>$Reg$</td>
<td>12345</td>
<td>Target register.</td>
</tr>
<tr>
<td>constant</td>
<td>$Const$</td>
<td>12345</td>
<td>A number or literal to equate $vreg$ with.</td>
</tr>
<tr>
<td>cont</td>
<td>$AInstr$</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>$Coord$</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>$Int^*$</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>
```

Example:

```plaintext
aEquateConstant(vreg:v(123) constant:foo ...)
```
**aEquateRecord**

Equate a register with a record.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vreg</td>
<td>Reg</td>
<td>12345</td>
<td>Target register.</td>
</tr>
<tr>
<td>label</td>
<td>Literal</td>
<td>12345</td>
<td>The label of the record.</td>
</tr>
<tr>
<td>arity</td>
<td>Arity</td>
<td>12345</td>
<td><em>arity</em> is either a inter or a list of features.</td>
</tr>
<tr>
<td>args</td>
<td>RArg+</td>
<td>12345</td>
<td>The argument of the record.</td>
</tr>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
</tbody>
</table>

Example:

```
aEquateRecord(vreg:x(2) label:foo arity:[bar a] args:[g(0) y(1)] cont:nil coord:...)```

**aUnify**

Unifies the contents of the two registers.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reg</td>
<td>12345</td>
<td>Left hand side of the unification.</td>
</tr>
<tr>
<td>2</td>
<td>Reg</td>
<td>12345</td>
<td>Right hand side of the unification.</td>
</tr>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

Example:

```
aUnify(x(0) x(1) ...)```
D.4.4 Conditionals and Control Flow

aTestBool

Test on a boolean value — if the register contains a non-boolean value, the error-path is taken.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vreg</td>
<td>Reg</td>
<td>12345</td>
<td>A register containing the boolean to be tested.</td>
</tr>
<tr>
<td>trueL</td>
<td>AInstr</td>
<td>12345</td>
<td>First instruction in the true-path.</td>
</tr>
<tr>
<td>falseL</td>
<td>AInstr</td>
<td>12345</td>
<td>First instruction in the false-path.</td>
</tr>
<tr>
<td>errorL</td>
<td>AInstr</td>
<td>12345</td>
<td>First instruction in the error-path.</td>
</tr>
<tr>
<td>trueKill</td>
<td>Int*</td>
<td>34</td>
<td>Killset for the true-path.</td>
</tr>
<tr>
<td>falseKill</td>
<td>Int*</td>
<td>34</td>
<td>Killset for the false-path.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>Killset for the error-path.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
</tbody>
</table>

Example 1 (Phase 3):

```
aTestBool(vreg:v(120) trueL:aCall(...)
    falseL:aUnify(...) errorL:aMatch(...)
trueKill:[v(120)] falseKill:nil
kill:[v(120)]
...
```

Example 2 (Phase 5):

```
aTestBool(vreg:y(3) trueL:aCall(...)
    falseL:aUnify(...) errorL:aMatch(...)
...
```

aMatch

Case statement. hmm...

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vreg</td>
<td>Reg</td>
<td>12345</td>
<td>Register to be matched against.</td>
</tr>
</tbody>
</table>
APPENDIX D. ABSTRACT INSTRUCTIONS

| nomatch | $AInstr$ | 12345 | First instruction in path taken if none of the hash table entries match. |
| entries | $Entry^+$ | 12345 | A list of hash table entries. |
| kill | $Int^*$ | 34 | Killset for the nomatch-path |
| coord | $Coord$ | 12345 | Location of the instruction in the source code. |

Example:

```
  aMatch(vreg:x(0) nomatch:aEquateConstant(...)
     entries,,,,)
  ...)
```

aGetVariable

This instruction is used to get the different fields in a path from an aMatch-instruction.

<table>
<thead>
<tr>
<th>aGetVariable</th>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>$Reg$</td>
<td>12345</td>
<td>Target register.</td>
</tr>
<tr>
<td>cont</td>
<td>$AInstr$</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
<td></td>
</tr>
<tr>
<td>coord</td>
<td>$Coord$</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
<td></td>
</tr>
<tr>
<td>kill</td>
<td>$Int^*$</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
<td></td>
</tr>
</tbody>
</table>

Example:

```
aGetVariable(v(51) kill:[v(5)] ...)
```

D.4.5 Objects

aGetSelf

Places reference to self-object in register.

<table>
<thead>
<tr>
<th>aGetSelf</th>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Reg$</td>
<td>12345</td>
<td>The reference is stored in this register.</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX D. ABSTRACT INSTRUCTIONS

<table>
<thead>
<tr>
<th>cont</th>
<th>AInstr</th>
<th>12345</th>
<th>Reference to the next instruction.</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

Example:

```
aGetSelf(x(0) ...)
```

**aSetSelf**

Set content of the given register as `self`-pointer in the emulator.

<table>
<thead>
<tr>
<th>aSetSelf</th>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reg</td>
<td>12345</td>
<td></td>
<td>The reference is taken from this register.</td>
</tr>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td></td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td></td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td></td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

Example:

```
aSetSelf(g(2) ...)
```

### D.4.6 Exceptions

**aExHandler**

Defines an exception handling routine.

<table>
<thead>
<tr>
<th>aExHandler</th>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exReg</td>
<td>Reg</td>
<td>12345</td>
<td></td>
<td>The register to keep the exception in.</td>
</tr>
<tr>
<td>tryL</td>
<td>AInstr</td>
<td>12345</td>
<td></td>
<td>The try-body.</td>
</tr>
<tr>
<td>catchL</td>
<td>AInstr</td>
<td>12345</td>
<td></td>
<td>The catch body.</td>
</tr>
<tr>
<td>tryKill</td>
<td>Int*</td>
<td>12345</td>
<td></td>
<td>Killset for the try-path.</td>
</tr>
</tbody>
</table>
APPENDIX D. ABSTRACT INSTRUCTIONS

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
catchL:aEquateRecord(...) tryKill:[v(220)]
```

**aPopEx**

Removes the latest exception handler from the exception stack.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
aPopEx(cont: aLockEnd(...) ...)```

D.4.7 Threads

**aLockEnd**

Unlocks the current thread.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>
Example:

\[a\text{LockThread}(\text{cont}:a\text{Shared}(\ldots)\ldots)\]

\textit{aLockThread}

Locks the current thread.

\begin{center}
\begin{tabular}{|l|l|l|l|}
\hline
Attribute & Type & Phase & Description \\
\hline
\text{label} & \text{Label} & 12345 & Label… \\
\hline
\text{lockReg} & \text{Reg} & 12345 & Register used for the lock. \\
\hline
\text{cont} & \text{AInstr} & 12345 & Reference to the next instruction. \\
\hline
\text{coord} & \text{Coord} & 12345 & Location of the instruction in the source code. \\
\hline
\text{kill} & \text{Int*} & 34 & The registers that die after this instruction are enumerated here. \\
\hline
\end{tabular}
\end{center}

D.4.8 Misc

\textit{aShared}

A shared-node.

\begin{center}
\begin{tabular}{|l|l|l|l|}
\hline
Attribute & Type & Phase & Description \\
\hline
\text{paths} & \text{Int} & 12345 & The number of paths that join in this node. \\
\hline
\text{info} & — & 12345 & Node bound to a cell, used internally in the different phases. \\
\hline
\text{cont} & \text{AInstr} & 12345 & Reference to the next instruction. \\
\hline
\text{coord} & \text{Coord} & 12345 & Location of the instruction in the source code. \\
\hline
\text{kill} & \text{Int*} & 34 & The registers that die after this instruction are enumerated here. \\
\hline
\end{tabular}
\end{center}

Example:

\[a\text{Shared}(\text{paths}:3\ \text{info}:\text{<Cell>}\ \text{cont}:a\text{Call}(\ldots)\ldots)\]
aReorganize

Used to re-organize the mapping of virtual to physical registers. This instruction is first generated in the RA phase, and does not exist in earlier stages.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>Move*</td>
<td>12345</td>
<td>List of moves to be emitted.</td>
</tr>
<tr>
<td>tail</td>
<td>N/B</td>
<td>12345</td>
<td>The (unbound) tail to the list-of-moves.</td>
</tr>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
aReorganize(list:move(x(0) y(0)) | move(g(0) x(0)) | _
  tail:
    cont: aShared(...) ..)
```

aInlineAssign

Inline Assignment.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>Attr</td>
<td>12345</td>
<td>Attribute</td>
</tr>
<tr>
<td>value</td>
<td>Value</td>
<td>12345</td>
<td>Operand</td>
</tr>
<tr>
<td>cont</td>
<td>AInstr</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>Coord</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>Int*</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

The following code:

```plaintext
foo <- 10
```
where \texttt{aSetSelf(...)} has been executed, and \texttt{foo} is an attribute of the current object, could be translated to:

\begin{verbatim}
  aEquateConstant(vreg:xf(22) constant:10 cont:
  aInlineAssign(attribute:foo value:x(22) ...)
  ...
\end{verbatim}

\texttt{aInlineAt}

This instruction accesses an attribute in an object atomically. First, the current object must be specified with a call to \texttt{aSetSelf}.

The \texttt{aInlineAt} instruction is an optimization, the same semantics can be achieved by a built in call to \texttt{'Object.at'}.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>\textit{Attr}</td>
<td>12345</td>
<td>Attribute</td>
</tr>
<tr>
<td>result</td>
<td>\textit{Reg}</td>
<td>12345</td>
<td>Destination register.</td>
</tr>
<tr>
<td>cont</td>
<td>\textit{Instr}</td>
<td>12345</td>
<td>Reference to the next instruction.</td>
</tr>
<tr>
<td>coord</td>
<td>\textit{Coord}</td>
<td>12345</td>
<td>Location of the instruction in the source code.</td>
</tr>
<tr>
<td>kill</td>
<td>\textit{Int*}</td>
<td>34</td>
<td>The registers that die after this instruction are enumerated here.</td>
</tr>
</tbody>
</table>

The following code;

\begin{verbatim}
  B = @foo
\end{verbatim}

where \texttt{B} is unbound and stored in \texttt{x(3)}, \texttt{foo} is an attribute of the current object, and \texttt{aSetSelf(...)} has been executed, could be translated to;

\begin{verbatim}
  aInlineAt(attribute:foo result:x(3) ...)
\end{verbatim}

\texttt{aInlineDot}

This instruction accesses a feature of a record or object atomically.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vreg</td>
<td>\textit{Reg}</td>
<td>12345</td>
<td>Source register.</td>
</tr>
<tr>
<td>result</td>
<td>\textit{Reg}</td>
<td>12345</td>
<td>Destination register.</td>
</tr>
</tbody>
</table>
The following code;

\[
B = A.\text{foo}
\]

where \(A\) is a record in \(g(0)\) and \(B\) in \(x(20)\), unbound, could be translated to;

\[
\text{aInlineDot(vreg:g(0) result:xf(20) feature:foo ...)}
\]
Appendix E

Mozart Virtual Machine Bytecode

This appendix describes the byte code in detail. The appendix is taken from documentation written by Andreas Sundström, see [23]. The text has been edited and converted to \LaTeX, but is otherwise untouched.

E.1 The Instructions

Oz byte code consists of a sequence of instructions. Each instruction consists of a sequence of words, where the first word is an opcode (telling which instruction it is) and the following words are arguments to the instruction.

This chapter contains sections describing instructions. Such sections start with information about the instructions' parameters. The type of the parameter, information on how registers are used, and a small description are given. The parameter types are described in

Three terms are used to describe how registers are used by the instructions (\texttt{def}, \texttt{use} and \texttt{kill}). An assignment to a register defines (\texttt{def}) that register. Other occurrences of that register are \texttt{use}-occurrences. After executing certain instructions one can not say anything about the value a register contains. That instruction cancels that the register has been defined. The instruction is said to kill that register.

This document lacks all information on constraints on content of registers. For example, does the \texttt{setSelf} instruction expects the register to contain an object.

For some instructions an equivalent sequence of instruction is given. For other instructions a sequence of instruction that is not legal is given. These instructions are used as a kind of pseudo code to help illustrate the semantics of the instruction. An instruction in these code examples is given as the instructions name followed by the arguments within parenthesis. The code examples are relative byte code; instead of absolute addresses in the memory,
labels are used. \texttt{label(L1)} in front of an instruction states that the instruction has the address of label L1.

Furthermore a description of the instructions is given. Most important constraints on how to use the instructions, in a legal way, are given.

In the description of the instructions following notions are used.

The graph store

The part of the heap that stores Oz data.

The structure pointer

A register that enables the emulator to work with the graph store.

Mode

While working on data structures the emulator may be in one of two different modes. Read mode, in which data in the graph store is matched against or write mode in which new data is built in the graph store.

Abstraction entry

A dereferenced Oz procedure for faster access to the code area.

The instructions in this chapter are sometimes more general than the real instructions. For example the instruction \texttt{setValue} correspond to \texttt{setValueX}, \texttt{setValueY} and \texttt{setValueG} and the instruction \texttt{allocateL?} correspond to \texttt{allocateL1}, \texttt{allocateL2}, ... and \texttt{allocateL10}. The real instruction is given in

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Type} & \textbf{def/use/kill} & \textbf{Description} \\
\hline
XRegister & def & register in which to store the created abstraction \\
Label & & address to jump to after abstraction creation \\
PredId & & print name, arity, source coordinates of the abstraction \\
OptPRef & contains use & address of an AbstractionEntry or unit \\
GRegRef & & contents of abstractions' local G registers \\
\hline
\end{tabular}
\end{table}

\textbf{what it does} Adds a procedure node to the store and puts a reference in the register. The body of the procedure is one part of this procedure node.
The body is the instructions between the definition instruction and the endDefinition instruction.

constraints Instructions within a definition should follow this pattern:

```
1bl(L1)    definition(_ L2 _ _ [R1 ... Rn])
   profileProc  % optional
   ...
   endDefinition(L1)
   localVarname(_)
   ...
   localVarname(_)
   globalVarname(_)
   ...
   globalVarname(_)
1bl(L2)
```

The number of localVarname instructions should be either 0 or the size of the local environment. The number of globalVarname instructions should be either 0 or the size of the globals list (last argument of the instruction). If any localVarname is present, there may only be one allocateL in the procedure. In a linear run through the code, definitions must always be correctly nested like this. Neither of endDefinition, localVarname and globalVarname may ever be executed by the emulator.

When running the body of the procedure the registers X(0) to X(arity-1) (for arity see PredId) and G(0) to G(N-1) (where N is the size of the globals list) are def.

definitionCopy

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use/def</td>
<td>both source and destination register address</td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td>to jump to after abstraction creation</td>
</tr>
<tr>
<td>PredId</td>
<td></td>
<td>print name, arity, source coordinates of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abstraction</td>
</tr>
<tr>
<td>OptPRef</td>
<td></td>
<td>address of an AbstractionEntry or unit</td>
</tr>
<tr>
<td>GRegRef</td>
<td></td>
<td>contents of abstractions’ local G registers</td>
</tr>
</tbody>
</table>

what it does Does the same things as definition except that it also makes a copy of the code segment.
Calls to procedures that is statically known to be unique can be optimized using special call instructions (see for example \code{callGlobal}. Calls to procedures on top-level can be optimized this way.

In a functor we can not do these optimizations right away. The imported variables is not statically known to be unique. They might hold different value every time the functor is executed.

Let us take a look at an example:

\begin{verbatim}
declare
functor F
import
    I
export
    E
define
    proc \{E\} \{I\} end
end
\end{verbatim}

We would like to optimize the call to \code{I} in the procedure \code{E}. The same code in Oz core language would look like this:

\begin{verbatim}
declare F in
    local Body1 UnnestApply1 UnnestApply2 RecordArg1 in
    proc instantiate \{Body1 IMPORT1 Result1\}
        local I E Feature1 in
        Feature1 = '\I'
        I = IMPORT1.Feature1
    proc \{E\}
        \{I\}
    end
    Result1 = 'export'(e: E)
    end
    RecordArg1 = 'procedure/0' % typeof E
    UnnestApply1 = 'import'('I': info(type: nil))
    UnnestApply2 = 'export'(e: RecordArg1)
    \{Functor.new UnnestApply1 UnnestApply2 Body1 F\}
end
\end{verbatim}

The procedure flag \code{instantiate} tells the compiler that we want to optimize calls to procedures passed as arguments. The down side of doing this is that calls to \code{Body1} will be a lot more expensive. Since functors are executed seldom, we can live with this. The emulator code follows:
APPENDIX E. MOZART VIRTUAL MACHINE BYTECODE

%% Assignment of Global Registers:
%% g(0) = F
%% Code Size:
88 % words

skip

lbl(1)  definition(x(0) 85 pid(‘Toplevel abstraction’ 0
  pos(’’ 1 0) [sited] 4 unit [g(0)])

lbl(7)  definition(x(2) 59 pid(‘Body1’ 2 pos(’’ 1 0) nil 3)
  <P: 1> nil)
  putList(x(2))
  setProcedureRef(<Q: 2>)
  setConstant(nil)

lbl(19)  definitionCopy(x(2) 54 pid(‘Body1/body’ 0 pos(’’ 1 0)
  nil 3) unit [x(0) x(1)])
  move(g(0) x(0))
  inlineDot(x(0) ’I’ x(1) cache)

lbl(34)  definition(x(2) 45 pid(‘E’ 0 pos(’’ 1 0) nil 0)
  <Q: 2> [x(1)])
  callGlobal(g(0) 1)
  endDefinition(34)

lbl(45)  getRecord(‘export’ [e] g(1))
  unifyValue(x(2))
  return
  endDefinition(19)

lbl(54)  tailCall(x(2) 0)
  endDefinition(7)

lbl(59)  putRecord(info [type] x(1))
  setConstant(nil)
  putRecord(‘import’ [?’I’] x(0))
  setValue(x(1))
  putRecord(‘export’ [e] x(1))
  setConstant(‘procedure/0’)
  move(g(0) x(3))
  callConstant(<P/4 NewFunctor> 9)
  endDefinition(1)

lbl(85)  tailCall(x(0) 0)

The definitionCopy instructions takes a list of procedure references in
the register. This is a list of procedures that is defined inside Body1. The
instruction uses the information in the list to maintain a hash table. In
the hash table, references to where the new copies are stored can be looked
up using the original procedures references. By copying the code each time
Body1 is called, we can assure that the arguments are unique. The call to I
is compiled to callGlobal(g(0) 1). If the flag instantiate is omitted the
call will be compiled to \texttt{tailCall(g(0) 0)}:

\%\% Assignment of Global Registers:
\%\%  \texttt{g(0) = F}
\%\% Code Size:
68 \% words

\texttt{skip}\nlbl(1) \texttt{definition(x(0) 65 pid(’Toplevel abstraction’ 0}
\hspace{1em}  \texttt{pos(’’ 1 0) [sited] 4) unit [g(0)]})
\texttt{inlineDot(x(0) ’I’ x(0) cache)}
lbl(7) \texttt{definition(x(2) 39 pid(’Body1’ 2 pos(’’ 1 0) nil 3)}
\hspace{1em}  \texttt{<P: 1> nil})
\texttt{return [x(0)]})
\texttt{tailCall(g(0) 0)}
\texttt{endDefinition(19)}
lbl(30) \texttt{getRecord(’export’ [e] x(1))}
\texttt{return x(2)}
\texttt{return [x(0)]})
\texttt{putRecord(info [type] x(1))}
\texttt{setConstant(nil)}
\texttt{putRecord(’import’ [’I’] x(0))}
\texttt{setConstant(x(1))}
\texttt{putRecord(’export’ [e] x(1))}
\texttt{setConstant(’procedure/0’)}
\texttt{move(g(0) x(3))}
\texttt{callConstant(<P/4 NewFunctor> 9)}
\texttt{endDefinition(1)}
lbl(65) \texttt{tailCall(x(0) 0)}

\textbf{constraints} exactly as for \texttt{definition}().

\textbf{profileProc}

No parameters.

\textbf{what it does} Updates information needed by the Mozart profiler.

\textbf{constraints} Can only occur as described in \texttt{definition}().

\textbf{endDefinition}
what it does  Marks the end of the procedure body.

constraints  Can only occur as described in definition ()..

E.1.1  Moving data around

move

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td></td>
<td>address of the corresponding ‘definition’ instruction</td>
</tr>
</tbody>
</table>

what it does  Copies the content of the first register to the second register.

moveMove

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>use</td>
<td>source</td>
</tr>
<tr>
<td>Register</td>
<td>def</td>
<td>destination, may not be a G register</td>
</tr>
</tbody>
</table>

equivalent to

move(Arg1 Arg2)
move(Arg3 Arg4)

special case  If the second and the third argument is the same, the third argument can be considered def (just as the second argument is).

E.1.2  Creating variables

createVariable

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>def</td>
<td>destination, may not be a G register</td>
</tr>
</tbody>
</table>
**what it does**  Adds a variable node to the store and puts a reference to it in the register.

createVariableMove

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>def</td>
<td>destination, may not be a G register</td>
</tr>
<tr>
<td>Register</td>
<td>def</td>
<td>co-destination</td>
</tr>
</tbody>
</table>

**equivalent to**

createVariable(Arg1)
move(Arg1 x(Arg2))

**E.1.3  Allocation of new data structures**

putConstant

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>def/</td>
<td>source</td>
</tr>
<tr>
<td></td>
<td>use/kill</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>def</td>
<td>destination, may not be a G register</td>
</tr>
</tbody>
</table>

**what it does**  Puts a reference to the Oz constant value in the register.

putRecord

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>def/</td>
<td>record label, may not be '}' if Arg2 == 2</td>
</tr>
<tr>
<td></td>
<td>use/kill</td>
<td></td>
</tr>
<tr>
<td>RecArity</td>
<td>def</td>
<td>record arity, may not be 2 if Arg1 == '}'</td>
</tr>
<tr>
<td>Register</td>
<td>def</td>
<td>destination, may not be a G register</td>
</tr>
</tbody>
</table>

**what it does**  Adds a record node to the store and puts a reference to it in the register. The structure pointer (SP) is set to the first field of the record such that the following instructions can initialize the fields of the record.

**constraints**  After a putRecord, the values of exactly a number of subtrees equal to the width of the record must be initialized by the setConstant, setValue, setVariable or setVoid instructions.
putList

<table>
<thead>
<tr>
<th>putList</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Register</td>
</tr>
</tbody>
</table>

used instead of

putRecord(‘?’ 2 Arg1)

**what it does** Adds a list node to the store and puts a reference to it in the register. The structure pointer (SP) is set to the first field of the list, such that the following instructions can initialize the fields of the list.

**constraints** After a putList, the values of exactly two subtrees must be initialized by the setConstant, setValue, setVariable or setVoid instructions.

setConstant

<table>
<thead>
<tr>
<th>setConstant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Constant</td>
</tr>
</tbody>
</table>

**what it does** Writes the node representation of the Oz constant value in the field where SP is pointing and increments SP.

**constraints** Can only occur as described in putList or putRecord.

setValue

<table>
<thead>
<tr>
<th>setValue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Register</td>
</tr>
</tbody>
</table>

**what it does** Writes the content of the register in the field where SP is pointing and increments SP.

**constraints** Can only occur as described in putList or putRecord.

setVariable
### setVariable

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>def</td>
<td>register to hold variable</td>
</tr>
</tbody>
</table>

**equivalent to**

```plaintext
createVariable(Arg1)
setValue(Arg1)
```

**constraints** Can only occur as described in `putList` or `putRecord`.

### setVoid

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>def</td>
<td>number of arguments to set to '<em>'</em></td>
</tr>
</tbody>
</table>

**what it does** Adds `number` of variable nodes to the field where SP is pointing and following fields. The SP is incremented `number` steps.

**constraints** Can only occur as described in `putList` or `putRecord`. Initializes as many subtrees as stated in the first argument.

### setProcedureRef

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRef</td>
<td>def</td>
<td>source</td>
</tr>
</tbody>
</table>

**what it does** Writes the node representation of the procedure reference (foreign pointer) value in the field where SP is pointing and increments SP.

**constraints** Can only occur as described in `putList` or `putRecord`.

### E.1.4 Matching and unification of data structures

#### getNumber

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>def</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>use</td>
<td></td>
</tr>
</tbody>
</table>
what it does  Unify the register with the number.

getLiteral

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>use</td>
<td></td>
</tr>
</tbody>
</table>

what it does  Unify the register with the literal.

getRecord

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>use</td>
<td></td>
</tr>
<tr>
<td>RecArity</td>
<td>use</td>
<td>record arity, may not be 3 or 4 if Arg2 == 2</td>
</tr>
<tr>
<td>Register</td>
<td>use</td>
<td>value to unify with</td>
</tr>
</tbody>
</table>

what it does  If the register contains a variable then a record node is added to the graph store. The variable is bound to that record and the mode is set to write. SP is set to the first field of the record.

If the register contains a record with the right label and arity the mode is set to read and the SP is set to the first field of the record. Otherwise fail exception is raised.

customs  After a getRecord, the values of exactly a number of subtrees equal to the width of the record must be unified by the unifyNumber, unifyLiteral, unifyValue, unifyVariable, unifyValVar or unifyVoid instructions.

getList

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>use</td>
<td></td>
</tr>
</tbody>
</table>

used instead of

getRecord("|", 2 Arg1)

what it does  If the register contains a variable then a list node is added to the graph store. The variable is bound to that list and the mode is set to
write. SP is set to the first field of the list.
If the register contains a list the mode is set to read and the SP is set to the first field of the list. Otherwise fail exception is raised.

**constraints** After a getList, the values of exactly two subtrees must be unified by the unifyNumber, unifyLiteral, unifyValue, unifyVariable, unifyValVar
or unifyVoid instructions.

**getListValVar**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>use</td>
<td></td>
</tr>
<tr>
<td>XRegister</td>
<td>def</td>
<td></td>
</tr>
</tbody>
</table>

equivalent to

gList(x(Arg1))
unifyValue(Arg2)
unifyVariable(x(Arg3))

**unifyNumber**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>use</td>
<td>value to unify with</td>
</tr>
</tbody>
</table>

**what it does** In read mode, if SP points to a variable bind it to the number, otherwise, fail if what SP points on differs from the number. In write mode, the number is written to the field where SP is pointing. In both modes SP is incremented.

**constraints** Can only occur as described in getList or getRecord.

**unifyValue**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>use</td>
<td>value to unify with</td>
</tr>
</tbody>
</table>
**what it does**  In read mode the content of the register is unified with whatever SP is pointing to. In write mode the content of the register is written to the field where SP is pointing. In both modes SP is incremented.

**constraints**  Can only occur as described in `getList` or `getRecord`.

**unifyVariable**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>def</td>
<td>register to hold variable</td>
</tr>
</tbody>
</table>

**createVariable(Arg1)**

**createVariable(Arg1)**

**constraints**  Can only occur as described in `getList` or `getRecord`.

**unifyValVar**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>use</td>
<td>value to unify with</td>
</tr>
<tr>
<td>Register</td>
<td>def</td>
<td>register to hold variable</td>
</tr>
</tbody>
</table>

**equivalent to**

**unifyValue(Arg1)**

**unifyVariable(Arg2)**

**constraints**  Can only occur as described in `getList` or `getRecord`. Unifies two subtrees.

**unifyVoid**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td></td>
<td>number of arguments to unify with ' underscores'</td>
</tr>
</tbody>
</table>

**what it does**  In write mode, add `number` of variable nodes to the field where SP is pointing and following fields. The SP is incremented `number` steps. In read mode, just incremented SP `number` steps.
constraints Can only occur as described in getList or getRecord. Unifies as many subtrees as stated in the first argument.

unify

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>use</td>
<td>first operand</td>
</tr>
<tr>
<td>Register</td>
<td>use</td>
<td>second operand</td>
</tr>
</tbody>
</table>

what it does Unifies the contents of the two register.

E.1.5 Allocation of local environments

allocateL

<table>
<thead>
<tr>
<th>allocateL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>def/use/kill</td>
</tr>
<tr>
<td>Count</td>
<td>size of local environment to allocate, must be &gt; 0</td>
</tr>
</tbody>
</table>

what it does Allocates memory for Y-registers.

constraints The register $y(i)$ may only be referenced after an allocateL($N$) has been executed in the same procedure. Furthermore, $i$ must be positive and less than $N$. A procedure needs not have an environment. All environments must be explicitly deallocated in the same procedure. An allocateL instruction may not be executed if an environment is already allocated.

allocateL?

The instructions allocateL1, allocateL2, ... allocateL10, with no parameters, are equivalent to allocateL with argument 1, 2, ... 10 respectively.

deAllocateL

<table>
<thead>
<tr>
<th>deAllocateL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>def/use/kill</td>
</tr>
<tr>
<td>Count</td>
<td>size of local environment to allocate, must be &gt; 0</td>
</tr>
</tbody>
</table>

what it does Frees memory used for local environment.
constraints  See AllocateL.

deAllocateL?
The instructions deAllocateL1, deAllocateL2, ... deAllocateL10, with no parameters, are a faster form of deAllocateL. The local environment that is de-allocated must have the size 1, 2 ... 10 respectively.

E.1.6 Applying builtins

The inline instructions that accesses features or attributes takes a cache as an argument. If such an instructions is executed two times in a row with the same object (or record) the access will be faster the second time (due to the cache).

callBI

callBI

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BI-name</td>
<td>def/use/kill</td>
<td>name of the builtin procedures</td>
</tr>
<tr>
<td>Location</td>
<td>BI-depentent</td>
<td>parameters to the builtin procedure</td>
</tr>
</tbody>
</table>

what it does  Calls the builtin. Location contains information about which registers is used for the in and out parameters.

callBI

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>the data structure</td>
</tr>
<tr>
<td>Feature</td>
<td></td>
<td>the feature</td>
</tr>
<tr>
<td>XRegister</td>
<td>def</td>
<td>the return value</td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

constraints  The location information must match the specification of the builtin procedure. The specification of the builtin contains information on use, def and kill.

inlineDot

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>the data structure</td>
</tr>
<tr>
<td>Feature</td>
<td></td>
<td>the feature</td>
</tr>
<tr>
<td>XRegister</td>
<td>def</td>
<td>the return value</td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

what it does  Feature selection on records or features.

inlineAt

| inlineAt |
|----------|----------|
| Type     | def/use/kill | Description |
| XRegister | use         | the data structure |
| Feature   |             | the feature   |
| XRegister | def         | the return value |
| Cache     |             |              |
**what it does**  Attribute lookup.

**inlineAssign**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>use</td>
<td>attribute</td>
</tr>
<tr>
<td>XRegister</td>
<td>def</td>
<td>destination register</td>
</tr>
</tbody>
</table>

**what it does**  Sets an attribute.

**inlinePlus1**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>argument register</td>
</tr>
<tr>
<td>XRegister</td>
<td>def</td>
<td>result register</td>
</tr>
</tbody>
</table>

**what it does**  Adds the content of a register with one and stores the result in an other register.

**inlineMinus1**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>argument register</td>
</tr>
<tr>
<td>XRegister</td>
<td>def</td>
<td>result register</td>
</tr>
</tbody>
</table>

**what it does**  Subtracts one from the content of a register and stores the result in an other register.

**inlinePlus**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>first argument register</td>
</tr>
</tbody>
</table>
what it does  Adds the content of two registers and stores the result in an
other register.

inlineMinus

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>first argument register</td>
</tr>
<tr>
<td>XRegister</td>
<td>use</td>
<td>second argument register</td>
</tr>
<tr>
<td>XRegister</td>
<td>def</td>
<td>result register</td>
</tr>
</tbody>
</table>

what it does  Performs a subtraction between the content of two registers
and stores the result in an other register.

E.1.7  Applying procedures and calling methods

In all of the following instructions \( X(0) \) to \( X(\text{arity}-1) \) are use. These reg-
isters hold the parameters to the procedure.

call

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>use</td>
<td>procedure</td>
</tr>
<tr>
<td>Arity</td>
<td>use</td>
<td>expected arity</td>
</tr>
</tbody>
</table>

what it does  Waits until the content of the register is determined. It
raises exception if it is not a procedure or the arity does not match the
expected number of parameters. Stores the address of the next instruction
(on a task stack) and starts executing the procedure body.

tailCall

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>use</td>
<td>procedure</td>
</tr>
<tr>
<td>Arity</td>
<td>use</td>
<td>procedure arity</td>
</tr>
</tbody>
</table>

what it does  This is a tail call optimization. Logically it is equivalent to:
call(Arg1 Arg2)
return

callConstant

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td></td>
<td>procedure</td>
</tr>
<tr>
<td>Arity/Tail</td>
<td></td>
<td>2 * arity + (isTailCall? 1: 0)</td>
</tr>
</tbody>
</table>

**what it does** This instruction takes an Oz procedure as argument instead of a register. It can be used when it is statically known which procedure is called. The instruction is replaced by a `fastCall` (see .)

callProcedureRef

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRef</td>
<td></td>
<td>address of an AbstractionEntry</td>
</tr>
<tr>
<td>Arity/ Tail</td>
<td></td>
<td>2 * arity + (isTailCall? 1: 0)</td>
</tr>
</tbody>
</table>

**what it does** This instruction takes a reference to an abstraction entry as argument instead of a register. A procedure definition is in the implementation of the emulator referred to as an abstraction. An instance of a procedure is called abstraction entry. The instruction can be used when it is statically known which procedure is called and it is defined on top-level. The instruction is replaced by a `fastCall` (see .)

callGlobal

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRegister</td>
<td>use</td>
<td>procedure</td>
</tr>
<tr>
<td>Arity/ Tail</td>
<td></td>
<td>2 * arity + (isTailCall? 1: 0)</td>
</tr>
</tbody>
</table>

**what it does** Waits until the content of the register is determined then performs the call. The instruction is replaced by a `fastCall` (see .)

callMethod

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
</table>
what it does  Static method calls (i.e. `c, m(...)`) are compiled into this instruction. Waits until the content of the register is determined. It raises exception if it is not a class. If it exists a fast version of the method, the call is replaced with `fastCall` (see).

sendMsg

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>def</td>
<td>method label</td>
</tr>
<tr>
<td>Register</td>
<td>use</td>
<td>object</td>
</tr>
<tr>
<td>RecArity</td>
<td>use</td>
<td>expected method arity (may be 0)</td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

what it does  Calls which are statically known to take a record as its only argument may be compiled to this instruction. Waits until the content of the register is determined. Raises exception if it is not an object or a procedure. If it was an object a call is performed immediately. If it was a procedure the message must be created before the call. If a `sendMsg` instructions is executed two times in a row with the same object the access of the method will be faster the second time (due to the cache).

tailSendMsg

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>def</td>
<td>method label</td>
</tr>
<tr>
<td>Register</td>
<td>use</td>
<td>object</td>
</tr>
<tr>
<td>RecArity</td>
<td>use</td>
<td>method arity (may be 0)</td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

what it does  This is a tail call optimization. Logically it is equivalent to:

```
sendMsg(Arg1 Arg2 Arg3 Arg4)
return
```

E.1.8  Control flow

branch
\textit{branch} & *def/use/kill* & *Description*  
Label & & destination  

**what it does** Jumps to the instruction the label points to.

**return**

No parameters.

**what it does** Code of a procedure is terminated with this instruction. The emulator continues executing the code after the call. See also \texttt{lockThread} ()

**exHandler**

\begin{center}
\begin{tabular}{|c|c|l|}
\hline
\textit{exHandler} & *def/use/kill* & *Description*  
Label & & address to jump to after exception handler creation  
\hline
\end{tabular}
\end{center}

**what it does** Installs an exception handler and than jumps to the label (try code). It also stores a copy of the current local environment.

If an exception is raised the exception is stored in x(0) and the code following the instruction \texttt{exHandler} (catch code) is executed.

**constraints** Exception handling should be constructed using this pattern:

\begin{verbatim}
  exHandler(L1)
  ...
  % 'catch' code; exception is in x(0).
  ...
  ...
  branch(L2)  % (typically)
  lbl(L1) ...
  % 'try' code
  ...
  popEx
  lbl(L2)
\end{verbatim}

The catch-code may only use Y-registers initialized before the exHandler instruction. The try-code may not modify any Y-registers used in the handler/catch?? code. The try-code may not deAllocate Y-registers either.
popEx

No parameters.

**what it does**  Removes the topmost exception handler.

**constraints**  Can only occur as described in `exHandler()`.

### E.1.9 Conditionals

#### testBI

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bi-name</td>
<td>name of the builtin procedure</td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td>Bl-dependent</td>
<td>parameters to the builtin procedure</td>
</tr>
<tr>
<td>Label</td>
<td>destination if test is false</td>
<td></td>
</tr>
</tbody>
</table>

**what it does**  Calls a builtin procedure. If the function does not return `true` the emulator jumps to the label.

**constraints**  The location information must match the specification of the builtin procedure. The test flag in the specification of the builtin must be `true` (i.e. it must be a boolean function).

#### testLT

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>first operand</td>
</tr>
<tr>
<td>XRegister</td>
<td>use</td>
<td>second operand</td>
</tr>
<tr>
<td>XRegister</td>
<td>def</td>
<td>result</td>
</tr>
<tr>
<td>Label</td>
<td>def</td>
<td>destination if test is false</td>
</tr>
</tbody>
</table>

**what it does**  Jumps to the label if not the first argument is less then the second argument.

#### testLE

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>first operand</td>
</tr>
<tr>
<td>XRegister</td>
<td>use</td>
<td>second operand</td>
</tr>
</tbody>
</table>
what it does  Jumps to the label if not the first argument is less than or equal to the second argument.

testLiteral

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>value to test</td>
</tr>
<tr>
<td>Literal</td>
<td>use</td>
<td>literal to test against</td>
</tr>
<tr>
<td>Label</td>
<td>use</td>
<td>destination if test is false</td>
</tr>
</tbody>
</table>

what it does  Jumps to the label if the contents of the register is not equal to the literal.

testNumber

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>value to test</td>
</tr>
<tr>
<td>Number</td>
<td>use</td>
<td>number to test against</td>
</tr>
<tr>
<td>Label</td>
<td>use</td>
<td>destination if test is false</td>
</tr>
</tbody>
</table>

what it does  Jumps to the label if the contents of the register is not equal to the number.

testList

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>value to test</td>
</tr>
<tr>
<td>Label</td>
<td>use</td>
<td>destination if test is false</td>
</tr>
</tbody>
</table>

what it does  Jumps to the label if the register does not contain a list.

constraints  After a testList, up to two arguments may be matched by the getVariable, getVarVar or getVoid instructions.

testRecord
### testRecord

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>value to test</td>
</tr>
<tr>
<td>Literal</td>
<td>use</td>
<td>label of the record to test against</td>
</tr>
<tr>
<td>RecArity</td>
<td>use</td>
<td>Arity of the record to test against</td>
</tr>
<tr>
<td>Label</td>
<td>use</td>
<td>destination if test is false</td>
</tr>
</tbody>
</table>

**what it does** Jumps to the label if the register does not contain a record with the right record label and arity.

**constraints** After a testRecord, up to the width of the record number of arguments may be matched by the `getVariable`, `getVarVar` or `getVoid` instructions.

### testBool

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>value to test</td>
</tr>
<tr>
<td>Label</td>
<td>use</td>
<td>destination if test is false</td>
</tr>
<tr>
<td>Label</td>
<td>use</td>
<td>destination if test is neither true nor false</td>
</tr>
</tbody>
</table>

**what it does** Jumps to the first label if the register contains `false`. Jumps to the second label if the register does not contain a boolean.

### match

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRegister</td>
<td>use</td>
<td>value to test</td>
</tr>
<tr>
<td>TableRef</td>
<td>use</td>
<td>patterns and destinations</td>
</tr>
</tbody>
</table>

**what it does** Oz case statements are often compiled into a `match` instruction. This instruction contains a hash table that maps primitive values and records to labels. The hash table also contains a label which to jump to if no other match is found.

If the register contains a variable the instruction will suspend; otherwise it will jump to the label that matches the content of the register. If it matches a record, SP is set to the first field of that record.

**constraints** At labels which to jump to if a record is matched, up to n (width of the corresponding record pattern) record arguments may be
matched by the `getVariable`, `getVarVar` or `getVoid` instructions.

**getVariable**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>def</td>
<td>variable to match against</td>
</tr>
</tbody>
</table>

**what it does** Stores whatever SP is pointing to, in the register and increments SP.

**constraints** Can only occur as described in `match`, `testList` or `testRecord`.

**getVarVar**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>def</td>
<td>first variable to match against</td>
</tr>
<tr>
<td>Register</td>
<td>def</td>
<td>second variable to match against</td>
</tr>
</tbody>
</table>

**equivalent to**

`getVarVariable(Arg1)`
`getVarVariable(Arg2)`

**constraints** Can only occur as described in `match`, `testList` or `testRecord`. Matches two elements in the pattern.

**getVoid**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td></td>
<td>number of arguments for which to skip matching</td>
</tr>
</tbody>
</table>

**what it does** Increments SP `number` steps.

**constraints** Can only occur as described in `match`, `testList` or `testRecord`. Matches as many subtrees as stated in the first argument.
E.1.10 Debug instructions

Instructions with extra information needed by the Oz debugger.

**localVarname**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Varname</td>
<td></td>
<td>print name of the next Y register or &quot;&quot;</td>
</tr>
</tbody>
</table>

**what it does** Maps a Y-register to variable names.

**constraints** Can only occur as described in definition. Slots of the local environments named via localVarname (by a name other than ") may only be written to once (and possibly be cleared using clearY).

**globalVarname**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Varname</td>
<td></td>
<td>print name of the next G register or &quot;&quot;</td>
</tr>
</tbody>
</table>

**what it does** Maps a G-register to variable names.

**constraints** Can only occur as described in definition.

**debugEntry**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td></td>
<td>file name</td>
</tr>
<tr>
<td>Number</td>
<td></td>
<td>line number</td>
</tr>
<tr>
<td>Number</td>
<td></td>
<td>column number</td>
</tr>
<tr>
<td>Literal</td>
<td></td>
<td>comment</td>
</tr>
</tbody>
</table>

**what it does** The Mozart debugger makes it possible to execute a program step by step. In the beginning and end of each such step, the execution of a thread can be stopped. This instruction marks the start of such a step. It also contains information about what source code corresponds to this emulator code.

**debugExit**
### debugExit

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td></td>
<td>file name</td>
</tr>
<tr>
<td>Number</td>
<td></td>
<td>line number</td>
</tr>
<tr>
<td>Number</td>
<td></td>
<td>column number</td>
</tr>
<tr>
<td>Literal</td>
<td></td>
<td>comment</td>
</tr>
</tbody>
</table>

**what it does**  Marks the end of a debugging step (see debugEntry).

### E.1.11 Miscellaneous

#### skip

No parameters.

**what it does**  Nothing.

### lockThread

#### lockThread

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>def</td>
<td>where to continue after the critical section</td>
</tr>
<tr>
<td>XRegister</td>
<td>use</td>
<td>register pointing to lock</td>
</tr>
</tbody>
</table>

**what it does**  Suspends if the content of the register is a variable, otherwise, if it is something else than an Oz lock, an error is raised. If current thread has got the lock the thread keeps running. If no thread has got the lock, current thread gets it and keeps running. If some other thread has got the lock the thread suspends.

The return statement is overloaded to mark the end of a critical section. When the thread is in a critical section and reaches the return statement the emulator will jump to the label argument (of the lockThread instruction).

**constraints**  Locking threads should be done using this pattern:

```plaintext
lockThread(L1 _)  
...  % 'lock' body instructions
return  % release the lock and jump to L1
lbl(L1)  ...  % where to continue after releasing the lock
```

### getSelf

#### getSelf
**what it does**  The register is set to current object.

**setSelf**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRegister</td>
<td>use</td>
<td>source register</td>
</tr>
</tbody>
</table>

**what it does**  Current object is set to the content of the register. This instruction is used when a procedure inside a method needs access to the object.

**clearY**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YRegister</td>
<td>kill</td>
<td>register which value to drop</td>
</tr>
</tbody>
</table>

**what it does**  After this instruction the register has dropped its reference. That makes it possible for memory to be reclaimed.

### E.1.12 Hidden instructions

The emulator may replace call instructions with following instructions. The following instructions must have the same size as the other call instructions. Therefore they have a dummy argument.

**fastCall**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRef</td>
<td>def</td>
<td>address of an AbstractionEntry</td>
</tr>
<tr>
<td>Dummy</td>
<td>use</td>
<td></td>
</tr>
</tbody>
</table>

**what it does**  This instruction takes a reference to an abstraction entry as argument.

**fastTailCall**
**fastTailCall**

<table>
<thead>
<tr>
<th>Type</th>
<th>def/use/kill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRef</td>
<td></td>
<td>address of an AbstractionEntry</td>
</tr>
<tr>
<td>Dummy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**what it does** This is a tail call optimization. Logically it is equivalent to:

```plaintext
defCall(Arg1 Arg2)
return
```

**E.1.13 Parameter types**

The grammar for parameter types that follows are represented as it is in relative byte code. It has tuples and other Oz data types. Absolute byte code uses no tuples. For example is Location and CallMethodInfo is in absolute byte code represented with objects of the C++ classes OZ_Location and CallMethodInfo. This format is used in the grammar because it is easy to read and it contains the same information as absolute byte code.

The notation `[Type]` in the grammar means an Oz list with zero or more elements of type *Type*. 
Appendix F

Mozart Virtual Machine Instructions

F.1 The MVM Instruction Set

This appendix contains a list of the real machine instruction set as used by the MVM. The main difference between these instructions and those described in Appendix E is that the real instructions contains information about the type of the registers passed as arguments.

F.1.1 Defining procedures

definition
definitionCopy
endDefinition

F.1.2 Moving data around

moveXX
moveXY
moveYX
moveYY
moveGX
moveGY
moveMoveXXYY
moveMoveXXYY
moveMoveXXYX
moveMoveXXYY

F.1.3 Creating variables

createVariableX
createVariableY
createVariableMoveX
createVariableMoveY

F.1.4 Allocation of new data structures
putConstantX
putConstantY
putRecordX
putRecordY
putRecordG
putListX
putListY
setConstant
setVariableX
setVariableY
setValueX
setValueY
setValueG
setProcedureRef
setVoid

F.1.5 Matching and unification of data structures
getNumberX
getNumberY
getNumberG
getLiteralX
getLiteralY
getLiteralG
getRecordX
getRecordY
getRecordG
getListX
getListY
getListG
ggetListValVarX
unifyNumber
unifyValueX
unifyValueY
unifyValueG
unifyVariableX
unifyVariableY
unifyValVarXX
unifyValVarXY
unifyValVarYY
unifyValVarGX
unifyValVarGY
unifyLiteral
unifyVoid
unifyXX
unifyXY
unifyXG

F.1.6 Allocation of local environments
allocateL
allocateL
allocateL1
allocateL2
allocateL3
allocateL4
allocateL5
allocateL6
allocateL7
allocateL8
allocateL9
allocateL10
deAllocateL
deAllocateL1
deAllocateL2
deAllocateL3
deAllocateL4
deAllocateL5
deAllocateL6
deAllocateL7
deAllocateL8
deAllocateL9
deAllocateL10

F.1.7 Applying built-ins
callBI
inlineDot
inlineAt
inlineAssign
inlinePlus1
inlineMinus1
inlinePlus
inlineMinus

F.1.8 Applying procedures and calling methods

callX
callY
callG
tailCallX
tailCallG
callConstant
callProcedureRef
callGlobal
callMethod
sendMsgX
sendMsgY
sendMsgG
tailSendMsgX
tailSendMsgY
tailSendMsgG

F.1.9 Control flow

branch
return
exHandler
popEx

F.1.10 Conditionals

testBI
testLT
testLE
testLiteralX
testLiteralY
testLiteralG
testNumberX
testNumberY
testNumberG
testRecordX
testRecordY
testRecordG
testListX
testListY
testListG
testBoolX
testBoolY
testBoolG
matchX
matchY
matchG
getVariableX
getVariableY
getVarVarXX
getVarVarXY
getVarVarYX
getVarVarYY
getVoid

F.1.11 Debug instructions

localVarname
globalVarname
debugEntry
debugExit

F.1.12 Miscellaneous

skip
lockThread
getSelf
setSelfG
getReturnX
getReturnY
getReturnG
funReturnX
funReturnY
funReturnG
clearY
profileProc
Index

$, 15
Exchange, 14
NewCell, 14
NewName, 13
Value.byNeed, 15
case, 22
cond, 21
fun, 15, 18
if, 14
lazy, 18
raise, 15
skip, 12
thread, 14, 16
try, 14
γ-calculus, 30
  abstraction, 31
  application, 31
  closures, 32
  congruence, 32
  example, 32
  expressions, 31
  extended, 30
  reduction, 32
  semantics, 31
  syntax, 31

Abstract Instructions
aCall, 128
aCallBuiltin, 129
acronyms, 125
aDefinition, 126
aDefinitionCopy, 127
aEquateConstant, 130
aEquateRecord, 131
aExHandler, 134
aGetSelf, 133
aGetVariable, 133
aInlineAssign, 137
aInlineAt, 138
aInlineDot, 138
aLockEnd, 135
aLockThread, 136
aMatch, 132
aPopEx, 135
aReorganize, 137
aSetSelf, 134
aShared, 50, 74, 90, 91, 136
aTestBool, 50, 74, 90, 132
aUnify, 131
summary, 124
Unification, 130

Acronyms
  Abstract Instructions, 125
aDefinition, 126
anonymous procedures, 19
application
  procedure, 14
attributes, 27

Browser, 17

Byte Code
  allocateL, 153
  allocateL?, 153
  branch, 158
  call, 156
  callBI, 154
  callConstant, 157
  callGlobal, 157
  callMethod, 157
  callProcedureRef, 157
clearY, 166
createVariable, 146
INDEX

createVariableMove, 147
deAllocate, 153
deAllocateL, 154
debugEntry, 164
debugExit, 164
definition, 141
definitionCopy, 142
definitionInstance, 145
dexHandler, 159
descCall, 166
descTailCall, 166
descList, 150
descListValVar, 151
descLiteral, 150
descNumber, 149
descRecord, 150
descSelf, 165
descVariable, 163
descVarVar, 163
descVoid, 163
describeVariable, 164
inlineAssign, 155
inlineAt, 154
inlineDot, 154
inlineMinus, 156
inlineMinus1, 155
inlinePlus, 155
inlinePlus1, 155
localVariable, 164
lockThread, 165
match, 162
move, 146
moveMove, 146
popEx, 160
profileProc, 145
putConstant, 147
putList, 148
putRecord, 147
return, 159
sendMessage, 158
setConstant, 148
setProcedureRef, 149
setSelf, 166
setValue, 148

setVariable, 148
setVoid, 149
skip, 165
tailCall, 156
tailSendMsg, 158
testBI, 160
testBool, 90, 91, 162
testLE, 160
testList, 161
testLiteral, 161
testLT, 160
testNumber, 161
testRecord, 161
unify, 153
unifyNumber, 151
unifyValue, 151
unifyValVar, 152
unifyVariable, 152
unifyVoid, 152
cell
  creation, 14
  exchange, 14
classes, 27
closure
  G-registers, 36
compiler efficiency, 5, 98
Compiler Phases, 69
  relevant to Abstract Instructions, 125
composition, 12
computation space, 20
concurrency, 16
constraint
  disentailment, 21
  entailment, 21
constraint programming, 23
  compiling, 20
  constraint propagation, 23
  domain, 24
  interval, 23
  domains, 23
  example, 26
  propagation, 23
INDEX

domain, 24
interval, 23
search, 24
constraint propagation, 23
domain, 24
interval, 23
constraint store, 20
constraints
atomic, 24
global, 25
reified, 25
continuation, 73, 89
core language, 11
statements, 11
translation to, 15
data-flow synchronization, 17
declaration
procedure, 13
variable, 13
determinism, 21
disentailment, 21
domains, 23
emitting code, 61, 89
entailment, 21
exception
handling, 14
raising, 15
feature selection, 13
features
in classes, 27
Features in Oz, 7
fSyntax, 76
functional programming, 18
functions, 18
higher-order, 19
future, 15
futures, 19
Graph Coloring, 56
higher-order functions, 19
inheritance, 28
instruction set, 38
kill, 52
kill set, 52
kill set algorithm, 54
known bugs, 5
Lamborghini
Diablo VT, 28
Language
core
translation to, 15
lazy evaluation, 18
lists, 17
liveness, 56
future, 98
kill set algorithm, 54
liveness algorithm, 53
liveness algorithm, 53
Liveness Analysis, 47
logic programming, 20
compiling, 20
logic variables, 8
methods, 27
private and protected, 28
Mozart, 3, 8
virtual machine, 35
multi-paradigm, 7
names, 13
object graph, 77, 79, 80
Object Orientation
in Oz, 27
object orientation
overhead, 28
objects, 27
optimization
branches, 94
Call Convention, 66, 97, 99
Cheap Call, 66, 97, 99
delayed initialization, 64, 99
inlining, 62, 99
peephole, 61, 93
INDEX

registers, 82
  tail-call, 94

Oz
  anonymous procedures, 19
  application
    procedure, 14
  Browser, 17
  cell
    creation, 14
    exchange, 14
  composition, 12
  computation space, 20
  concurrency, 16
  constraint store, 20
  core language, 11
  data-flow synchronization, 17
  declaration
    procedure, 13
    variable, 13
  examples, 15
  exception
    handling, 14
    raising, 15
  feature selection, 13
  features, 7
  foundation, 30
  functional programming, 18
  functions, 15, 18
    higher-order, 19
  futures, 15, 19
  higher-order functions, 19
  keywords
    NewName, 13
    skip, 12
  lazy evaluation, 18
  list unification, 12
  lists, 17
  names, 13
  procedure
    application, 14
    declaration, 13
  record unification, 12
  selection, 14
  space, 20
  store, 20
  threads, 14, 16
  unification, 9, 12
  variable
    declaration, 13
    variable unification, 12
  versions, 8
  paths, 50, 74, 75, 85, 91
  procedure
    application, 14
    declaration, 13
  Prolog, 23
  propagation, 23
    domain, 24
    interval, 23
  records, 41
  Register allocation
    Graph Coloring, 56
  registers
    G-registers, 36
    register, 55
    X-registers, 35
    Y-registers, 36
  search, 24
  selection, 14
  Send-More-Money puzzle, 26
  space, 20
  store, 20
  structures, 41
  syntactic sugar, 27
  threads, 14, 16
    blocking, 17
  Unification
    Abstract Instructions, 130
  unification, 9, 12
    list, 12
    record, 12
    variable, 12
  variable
declaration, 13
versions
of Oz, 8
Bibliography


