Michael Mehl

The Oz Virtual Machine
Records, Transients, and Deep Guards

Michael Mehl

The Oz Virtual Machine
Records, Transients, and Deep Guards

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät
der Universität des Saarlandes

Saarbrücken
1999

Das Promotionskolloquium fand am 18. Mai 1999 statt.

Dekan:	Prof. Wolfgang Paul
Gutachter:	Prof. Gert Smolka
	Prof. Peter van Roy

To my family
Barbara, Lukas, and Julius

Abstract

In this thesis we describe the design and implementation of a virtual machine LVM for the execution of Oz programs. Oz is a concurrent, dynamically typed, functional language with logic variables, futures, by-need synchronization, records, feature constraints, and deep guard conditionals. The LVM supports light-weight threads, first-class procedures, exception handling, transients as generalization of logic variables, futures, and constraint variables, records and open records, and multiple computation spaces to implement the deep guard conditional. We discuss the modular, open, and extensible design of the LVM. Techniques for the efficient implementation of the store on standard hardware are shown. The LVM subsumes well-known virtual machines for functional, logic, and imperative languages.

Zusammenfassung

In dieser Arbeit beschreiben wir das Design und die Implementierung einer virtuellen Machine LVM für die Ausführung von Oz Programmen. Oz ist eine nebenläufige, dynamisch getypte, funktionale Sprache mit logischen Variablen, Futures, by-need Synchronization, Records, Feature Constraints, und einer bedingten Anweisung mit tiefen Wächtern. Die LVM unterstützt leichtgewichtige Threads, Prozeduren als Datenstrukturen erster Ordnung, Ausnahmebehandlung, Transients als Verallgemeinerung von logischen Variablen, Futures und Constraint-Variablen, Records und offene Records, sowie multiple Berechnungsräume zur Implementierung der bedingten Anweisung mit tiefen Wächtern. Wir diskutieren ein modulares, offenes und erweiterbares Design der LVM und zeigen Techniken zur effizienten Implementierung des Speichers auf aktuell verfügbarer Hardware. Die LVM subsummiert virtuelle Maschinen für funktionale, logische und imperative Sprachen.

Extended Abstract

In this thesis we describe the design and implementation of a virtual machine LVM for the execution of Oz programs. Oz is a concurrent, dynamically typed, functional language. For didactical reasons we restrict the language to a subset, called L.

The focus of this work is on non-standard extensions of functional languages. These extensions include logic variables to represent unknown values and futures as read-only views on variables. These kinds of unknown values are generalized to transients.

Beside synchronization on determination of transients the language L supports by-need synchronization which provides for lazy programming. For the representation of data structures L supports trees and their partial description with records, feature constraints, and width constraints. L allows for multiple computation spaces, which are the foundation for implementing search engines. Computation spaces are introduced for the implementation of the deep guard conditional operator which allows to decide entailment and disentailment.

We define the semantics of the language informally as a graph rewriting engine on the language graph. The language graph defines a representation of the data structures of the language. The language is defined as a small set of rewriting operations on the language graph.

We show how the model of the language graph can be extended to explain multiple computation spaces. The extended graph model allows to explain concisely how bindings of variables are propagated, how entailment and disentailment is detected, and how two spaces are merged.

The LVM is a virtual machine which serves as an intermediate level between the high-level language L and a concrete hardware. It hides the platform-specific details and serves as a well defined target language for the compilation of L programs.

In this thesis we present a modular, open, and extensible design and implementation of the LVM. The main modules of the virtual machine are the store and the engine.

The store represents the data structures of the language. It is described with a refined graph model which makes essential properties of the implementation explicit, e.g. the usage of registers and heap memory.

The engine consists of a scheduler, a worker, and an emulator. The scheduler maintains the runnable threads using a simple round robin scheduling policy. The LVM supports extremely light-weight threads and thousands of threads can be created and scheduled efficiently.

The LVM has a single worker to execute threads. The worker maintains the tasks of a thread and implements exception handling. The state of the worker is lean to allow for efficient context switches between concurrent threads.

The state of the worker is rich enough for the efficient execution of machine programs through a threaded-code emulator. The byte-code of machine programs is compact and adapted for emulation. The byte-code contains direct references to nodes in the store, which allows for certain optimizations, e.g. avoiding dynamic type tests.

Transients are defined in the LVM as a generalization of unknown values, including logic variables, futures, and constraint variables. The common properties of transients are the single-assignment property and automatic synchronization of threads on their determination.

The LVM supports the representation of high-level symbolic data-structures with gracefully degrading performance wrt. expressivity. Simple data-structure like lists, integers, and literals are represented highly optimized. The performance decreases smoothly only when more expressive primitives, like records with dynamic arities and feature constraints, are used.

The LVM is extensible in multiple ways. New data structures and transient types can be integrated with varying degree of efficiency and complexity. At the bottom layer a sophisticated tagging scheme allows to efficiently represent the central data structures, which include integers, optimized variables and futures, list elements, and literals. At a medium layer the vast majority of data structures are represented, e.g. procedures, records, and objects. At the highest layer new data types can be integrated easily using an object-oriented approach with late binding.

The LVM design is open for experimentation with new features and concepts. Beside the extension of data structures it also allows to easily extend the engine. It is for example easy to integrate new functionality as built-in procedures and byte-code instructions.

We show techniques for the efficient implementation of the store on standard hardware. The representation of dynamically typed values in the store is implemented as a hybrid mix of tagged pointers and tagged objects. We explain the automatic memory management of the LVM, which is based on a free lists and a stop-and-copy garbage collector. A liveness analysis performed during the garbage collection allows to release memory which is referred from unused registers of the LVM.

Erweiterte Zusammenfassung

In dieser Arbeit beschreiben wir das Design und die Implementierung einer virtuellen Machine LVM für die Ausführung von Oz Programmen. Oz ist eine nebenläufige, dynamisch getypte, funktionale Sprache. Aus didaktischen Gründen beschränkten wir uns auf eine Teilsprache von Oz, die wir L nennen.

Der Schwerpunkt unserer Arbeit liegt auf untypischen Erweiterungen von funktionalen Sprachen. Diese Erweiterungen umfassen unter anderem logische Variablen zur Repräsentation von noch nicht bekannten Werten und Futures, die nur-lese Zugriffe auf Variablen definieren. Diese und andere Arten von unbekannten Werten werden generalisiert zu Transients.

Neben der Synchronization auf Transients, erlaubt L auch die by-need Synchronization, die es unter anderem erlaubt, die Auswertung von Ausdrücken zu verzögern, bis sie benötigt werden. Zur Repräsentation von Datenstrukturen unterstützt die Sprache L Bäume und ihre partielle Beschreibung durch Records, Feature-Constraints und Width-Constraints. L erlaubt multiple Berechnungsräume, die die Grundlage für die Implementierung von Suchmaschinen bilden. Berechnungsräume werden zur Implementierung von bedingten Anweisungen mit tiefen Wächter eingesetzt, die es erlauben Erfüllbarkeit und Unerfüllbarkeit zu entscheiden.

Wir definieren die Semantik der Sprache informell als ein Graphersetzungssystem auf dem Sprachgraphen. Der Sprachgraph definiert die Repräsentation der Datenstrukturen der Sprache. Die Sprache wird definiert durch eine kleine Menge von Ersetzungsregeln angewendet auf den Sprachgraphen.

Wir zeigen, wie das Model des Sprachgraphen erweitert werden kann, um multiple Berechnungsräume zu erklären. Das erweiterte Graphenmodell erlaubt es präzise zu erklären, wie die Bindung von Variablen propagiert wird, wie die Erfüllbarkeit bzw. Unerfüllbarkeit entschieden wird, und wie zwei Berechnungsräume verschmolzen werden.

Die LVM ist eine virtuelle Maschine, die eine Abstraktionsebene zwischen der Hochsprache L und einer konkreten Hardware realisiert. Sie verbirgt irrelevante plattformspezifische Details und dient als wohldefinierte Zielsprache für die Kompilierung von L Programmen.

In dieser Arbeit präsentieren wir einen modularen, offenen und erweiterbaren Design sowie eine Implementierung der LVM. Die zentralen Module der virtuellen Maschine sind der Speicher und die Verarbeitungsmaschine.

Der Speicher repräsentiert die Datenstrukturen der Sprache. Er ist beschrieben als verfeinertes Graphenmodell, das wichtige Eigenschaften der Implementierung explizit macht, zum Beispiel die Verwendung von Registern und dem Haldenspeicher.

Die Verarbeitungsmaschine besteht aus einem Scheduler, einem Worker, und einem Emulator. Der Scheduler verwaltet die rechenfähigen Threads durch eine einfache zyklische Warteschlange (round-robin). Die LVM erlaubt extrem leichtgewichtige Threads, wobei Tausende von Threads effizient erzeugt und verwaltet werden können.

Die LVM hat einen einzigen Worker zur Ausführung eines Threads. Der Worker verwaltet die Aufträge des Threads und implementiert die Ausnahmebehandlung. Der Zustand des Workers ist sehr kompakt, um die effiziente Threadumschaltung zu ermöglichen.

Der Zustand des Workers ist reich genug, um die effiziente Ausführung von Maschinenprogrammen durch einen ,,threaded-code" Emulator zu erlauben. Der Bytecode für Maschinenprogramme ist sehr kompakt und zugeschnitten auf einen Emulator-basierten Ansatz. Der Bytecode enthält direkte Referenzen auf Knoten im Speicher, die bestimmte Optimierungen, wie zum Beispiel die Vermeidung dynamischer Typüberprüfungen, erlauben.

Transients werden in der LVM als Verallgemeinerung unbekannter Werte, wie zum Bespiel logischer Variablen, Futures und Constraint Variablen, eingeführt. Die wichtigsten Merkmale von Transients sind, daß sie genau einmal gebunden werden können und Threads automatisch auf ihre Determiniertheit synchronisieren.

Die LVM unterstützt die Repräsentation von hochsprachlichen, symbolischen Datenstrukturen mit einer Performanz, die sich an die gewünschte Expressivität anpaßt. Auf der untersten Ebene steht ein elaboriertes Tag-Schema zur Verfügung, das die effiziente Repräsentation wichtiger Datenstrukturen, wie zum Beispiel ganze Zahlen, optimierte Variablen und Futures, Listenelemente und Literale, erlaubt. Auf der mittleren Ebene wird der größte Teil der Datentypen, wie zum Beispiel Prozeduren, Records und Objekte, realisiert. Auf der höchsten Ebene erlaubt eine einfache Schnittstelle, basierend auf Objekten mit später Bindung, die einfache Integration neuer Datentypen.

Das Design der LVM ist offen, um Experimente mit neuen Ideen und Konzepten durchführen zu können. Neben der Erweiterung von Datenstrukturen erlaubt die LVM auch die Erweiterung der Verarbeitungsmaschine. Zum Beispiel ist es einfach möglich zusätzliche Funktionalität durch eingebaute Prozeduren und Maschinenbefehle zu realisieren.

Wir zeigen Techniken für die effiziente Implementierung des Speichers auf aktuell verfügbarer Hardware. Die Repräsentation von dynamisch typisierten Werten im Speicher ist implementiert als eine hybride Mischung von markierten Zeigern mit markierten Objekten. Wir erklären die automatische Speicherverwaltung der LVM, die auf Freispeicherlisten und einem ,,stop-and-copy" Speicherbereinigungsalgorithmus basiert. Eine Lebendigkeitsanalyse wird während der Speicherbereinigung durchgeführt, die es erlaubt den Speicher von nicht verwendeten Registern freizugeben.

Acknowledgments

I thank foremost the whole team of the Programming Systems Lab at DFKI and at the University of Saarbrücken. The atmosphere was stimulating and a lot of fruitful discussion took place over the years.

I thank my advisor, Prof. Gert Smolka, as a great source of inspiration and new ideas and as a very knowledgeable expert in the field of programming languages. I admire his competence to explain and analyze complicated topics in a precise and clear manner. He communicated his insights and ideas to us, such that we were able to convert them into a practical useful system.

For seven years I shared my office with my colleague Ralf Scheidhauer and we developed many ideas presented in his and my thesis in close collaboration. I especially thank him for his clear mind wrt. the concrete realization of interesting but often too abstract and generic solutions of mine.

I thank Martin Henz, Denys Duchier, Ralf Scheidhauer, and Leif Kornstaedt for reading and commenting early drafts of this thesis. They gave me valuable hints, but I'm to be blamed for not following them.

I thank my colleagues Prof. Seif Haridi, Prof. Peter Van Roy, Kostja Popov, Per Brand, and Erik Klintskog for the international atmosphere in our project. I enjoyed the workshops with you very much, because you showed me that there is often more than one right opinion and how compromises can be found in a friendly environment. Kostja was in the core team for the implementation of Mozart from the beginning and it was always possible to discuss with him all the nasty but nevertheless essential details of the implementation.

I thank my employer, the German Research Center for Artificial Intelligence (DFKI), for supporting me and my work and for giving our project room for the basic research on programming languages with no immediate practical application. My work at DFKI was funded by the German Government (BMBF) under grant ITW 9105 and ITW 9601. The German Telekom, my current employer, gave me some support in the final stage of this work.

At the end, but not for the smallest part, I want to thank my family, especially my wife, my parents, and my parents in law, for their help, support, and patience

during the very very time consuming preparation of this thesis. My kids, Lukas and Julius, deserve thanks for their effort to show me that life is not only work.

Michael Mehl, January 1999

Contents

List of Figures

Chapter 1

Introduction

In this thesis we explain the implementation of the language Oz. Oz is a multi-paradigm programming language integrating concurrent constraint programming with first-class functions, high-level constraint based data structures, concurrent objects, powerful synchronization primitives, state of the art constraint systems, and flexible search engines.

We present the implementation as a virtual machine LVM which adds an intermediate abstraction between the high-level language Oz and the low-level details of concrete machines.

1.1 Concepts behind Oz

The foundation for Oz was laid in the γ-calculus [94] for concurrent programming, which integrates logic variables, names, first-class functions, and cells into a formal calculus. Seminal contributions to the foundation of Oz are the introduction of first class spaces and search combinators as a generalization of deep guard combinators [93, 90] and the integration of spaces, search combinators, and finite domain constraints into a constraint programming framework [91].

The full language Oz is defined and explained in [95, 96, 35, 36].

Mozart is the third release of the Oz system [72, 73, 66]. Mozart implements the language Oz and provides additionally the infrastructure needed for application development with Oz.

The structure of the Oz implementation is outlined as a pyramid in Figure 1.1.

To explain the implementation of the LVM we use a top down approach. Concepts, techniques, and insights are introduced at the highest possible layer and more and more details are added in lower levels.

The following paragraphs introduce basic concepts of Oz.

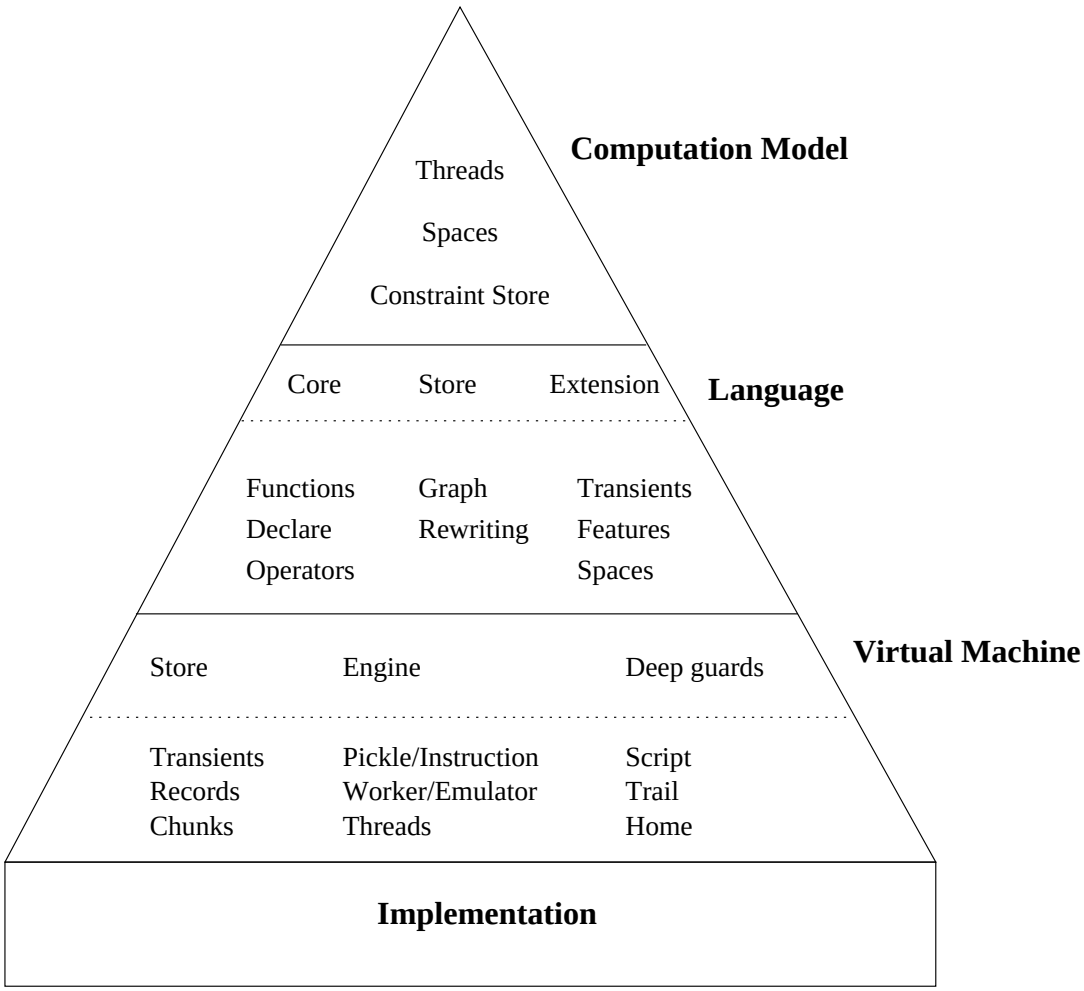


Figure 1.1: Overview of the layers.

1.1.1 First-class functions

Oz has first-class functions[1]. Functions are dynamically created closures encapsulating the environment in which they are defined. Functions can be passed as arguments to functions and returned as values of functions; they can be stored in data structures; and they even can be stored persistently on files.

First-class functions are a distinguishing feature of functional languages like Standard ML [63], Haskell [75], Lisp [50, 31], and Scheme [51].

1.1.2 Transients: Logic variables, futures

Oz supports logic variables, which are not yet known values. A logic variable can be assigned once and is than transparently replaced by this binding. Logic variables are a powerful concept to express partial data structures, to synchronize multiple threads, and to efficiently support call-by-reference output arguments.

A future is a read-only view on a variable, which allows to build safe partial data-structures, which can be modified only by a producer and not by consumers. Futures are transparently bound when their variable is bound.

Transients are defined as a generalization of unknown values, including logic variables, futures, and constraint variables. The common properties of transients are the single-assignment property and automatic synchronization of threads on their determination.

The use of logic variables in programming languages starts with Prolog [55]. The idea of futures occured in Multilisp [34] for expressing the results of parallel computations.

1.1.3 Threads, exceptions, and by-need synchronization

Oz is a concurrent language with extremely light-weight threads. Thousands of threads can be executed simultaneously. Threads in Oz are fair and preemptively scheduled. Threads in Oz allow for coarse-grained concurrent programming, although the implementation can handle thousands of threads.

Threads are a well-known concepts in operating systems, but their support in programming languages is still in the early stages. Threads are now standardized for the C/C++ language in the POSIX environment [43] and the language Java [30] also support these POSIX-like threads.

[1] In the literature first-class functions are sometimes called higher-order functions.

The exception mechanism of Oz allows to raise and handle first class exceptions. Exceptions offer a well-defined interface to handle errors. Exceptions are found in all modern languages, e.g. Java [30], Standard ML [63], and C++ [16].

By-need synchronization integrates lazy concurrent programming into Oz. By-need synchronization returns a future, which will be bound by a concurrent thread. This thread is only created, when the value of the future is requested. The future need not be explicitly requested, but when a thread synchronizes on the value of the future it is implicitly requested.

A well-known lazy functional programming language is Haskell [75].

1.1.4 Records and feature constraints

Oz has records as powerful data-structures to describe rational trees. A rational tree is a possibly infinite tree with labelled links and primitive values at the leaves. A record is a description of a node and all its links. With logic variables records allow to express trees where some nodes are unknown.

Feature constraints allow to express incomplete trees where not all links are known. A feature constraint defines that a certain link exists, without defining all other links.

Oz supports several other constraint systems beside trees, e.g. finite domains and finite sets, and it is extensible for other constraint systems. In this work we take feature constraints as an example to show how constraint systems are integrated with Oz.

Constraints over rational trees were introduced in Prolog II [19]. The foundation for records in Oz was laid in [98].

1.1.5 Cells and built-in abstract data types

State is introduced in Oz through a primitive entity called a cell. A cell is a container for one value. The content of a cell may be accessed and exchanged with a new value.

Records, cells, and first-class functions allow to build a state-of-the-art object system [42]. In this work we will show how to integrate a restricted form of objects as built-in abstract data types into the LVM.

1.1.6 Deep guard conditional and spaces

Oz supports multiple computation spaces to build powerful search engines. A computation space encapsulates a computation. A computation space has a cer-

tain state, namely running, entailed, stable, or disentailed. Threads can synchronize on this state. Two spaces can be merged together and a space can be copied to create an independent clone.

We show the deep guard conditional as an instance of the general concept of spaces, which allows to discuss how the synchronization on entailment and disentailment works and how spaces are merged.

Deep guards were first introduced in AKL [48], which was the first language implementing the concurrent constraint programming model [86]. Concurrent constraint programming integrates the paradigms of concurrent logic programming [92] and constraint logic programming [44, 45, 46]. In Oz deep guards are generalized to first-class computation spaces, which allow to express many different deep guard combinators and to build flexible search engines [90, 88, 89].

1.2 Contributions

In this thesis we present the design and implementation of a virtual machine for a subset L of the full language Oz. L is a multi-paradigm language which includes records, feature constraints, logic variables, futures, functions, threads, exceptions, and conditionals.

This thesis presents idealizations of the real VM that we have implemented in the Mozart system [66]. The thesis provides sufficient information to reconstruct the implementation.

A huge amount of our work, beside the design described here, went into engineering, coding, and supporting a practical, useful, and stable implementation.

The efficiency of the LVM is comparable to the implementation of modern high-level languages, e.g. Standard ML, Java, Prolog, Lisp, Smalltalk. A detailed evaluation of the LVM is given in [87].

An idealization of the LVM for rational tree constraints, first class functions, local computations for deep guards, and preemptive and fair scheduling was published in [62]. The integration of feature constraints and their gracefully degrading representation was described in [108].

Modular and open design

The design of the LVM is modular and orthogonal to cope with the complexity. The modules of the LVM correspond closely to the primitives of the language. The modules define a regular lean interface.

The design is open in the sense that

1. Design decisions and possibilities, especially with respect to the trade-off between efficiency and simplicity, are made explicit.

2. The hooks needed for the integration of new features, e.g. new data and control structures, are identified.

The virtual machine of Oz subsumes well-known virtual machines for logic, functional, and imperative languages.

The top-level modules of the LVM are the store, the engine, and spaces.

Store

The store implements the efficient representation of values, variables, futures, and constraints. We describe the store with a refined graph model, which makes central aspects of the design explicit. In this graph model different representations with varying complexity and efficiency are expressible.

Records and feature constraints give the expressivity to define high-level data structures. We show how this expressivity maps to a gracefully degrading representation wrt. the expressivity. Closed records can be represented with an efficiency similar to structures in the WAM. The performance overhead for the creation, access, and decomposition of records with symbolic features is minimal compared to structures in Prolog implementations. Only when the additional expressivity provided by the dynamic creation of arities, first class features, and feature constraints is used, a moderate cost has to be paid.

We show an abstraction, called transients, to support logic variables, futures, and constraint variables. Transients are generalized to allow for the integration of new types of unknowns. We analyze the cost of adding transients to a language which only has determined values.

Records with named features allow to define abstract data-types. Abstract data-types can be built into the LVM with a small interface. We describe a layered approach to implement abstract data-types with varying performance and complexity.

The store is subject to automatic memory management using a stop-and-copy collector and elaborated techniques to reuse memory as soon as possible. We explain the liveness analysis to ensure that unused registers are detected and discuss the impact of the optimized representation of variables in fields to memory management.

Engine

The engine takes care of the execution of machine programs. We present a compact machine model consisting of the scheduler, the worker, and the emulator.

Threads are managed by a round-robin scheduler with priorities. The techniques, which allow to create and maintain thousands of threads efficiently, are explained.

The LVM is a sequential implementation with a single worker to execute threads. We describe the context switching overhead for the efficient installation and deinstallation of threads by the worker, which is due to a compact representation of the state of the worker.

The worker executes first-class functions with call-by-reference arguments using logic variables for passing output arguments. The worker implements exceptions, where the trade-off between an efficient installation of exception handlers and an efficient lookup for the handler in the case of an exceptional condition is discussed.

Although the state of the worker is compactly represented it is well-suited for an efficient execution of the byte code by the emulator.

We present a classification of the machine instructions, which shows how much support for various language concepts is required.

The idea of having pickles, which define an external representation of Oz data-structures, allows for a novel account to byte code where instructions can directly refer to data-structures in the store. The loader creates an internal representation from a pickle. The transformation and optimizations of the byte code performed by the loader at run time are explained.

Spaces

Spaces allow to express encapsulated computations with constraint propagation and are an essential building block for constraint programming and search. We use conditionals as an instance of the general concept of first class spaces to discuss the complexity introduced to the LVM for supporting first-class computation spaces.

We define an extension of the single store graph model to a multiple store graph model which allows to explain at an intermediate level between the high-level constraint view and the low-level implementation the key aspects of spaces.

We show the hooks needed in the LVM to support spaces and the implementation of the script technique for representing multiple bindings of variables. We compare the script technique with the binding window technique.

1.3 Structure of the thesis

The top of the pyramid is the computation model and an informal definition of the subset of the full Oz language in Chapter 2. The computation model is defined as a number of threads computing over a shared store. We introduce the units represented in the store, i.e. values, variables, and futures, and the operations performed on the store when executing threads.

The next step down the pyramid is the explanation of the VM in Chapter 3. Its main parts are the store and the engine. At the virtual machine level a refined graph model is defined which allows to discuss many aspects of the representation of dynamically typed units.

We define a sequential imperative register-based machine for Oz, which consists of a machine language, the scheduler, and the worker. The connection between the high-level language and the machine language is explained by showing the compilation of L expressions into machine programs.

The loader is presented as a translator for an external representation of machine programs, called a pickle, into a internal graph and threaded-code representation.

The scheduler is the component which is responsible for the fair, preemptive scheduling of the runnable threads. It selects a thread which is then executed by a worker. The worker is responsible for switching contexts when a new thread must be installed or deinstalled. The worker executes the tasks of a single thread and emulates the instructions.

The issues introduced with the integration of spaces to the LVM are discussed next. We identify the hooks required in the other parts of the VM, explain the script and binding window technique for representing multiple bindings of variables, the propagation of bindings, and the algorithm for deciding entailment.

Then we compare the LVM with other virtual machines for high-level languages and summarize the main design goals.

After this discussion of the high-level aspects of the LVM we explain the implementation aspects in Chapter 4.

We explain how the different unit types are represented. The transient abstraction is introduced and its specialization to logic variables, futures, and constraint variables. The next part defines record constraints and their gracefully degrading implementation. We explain the extension mechanism for defining abstract data types and explain how they can be integrated smoothly into the LVM.

This part on the description of the store is completed with an explanation of the automatic memory management.

The thesis concludes in Chapter 5 with a summary, engineering considerations, and some remarks about future work.

1.4 Context of the thesis

The LVM was designed and implemented in close collaboration with my colleague Ralf Scheidhauer. Many parts of my work overlap with his thesis [87]. He describes the implementation of the core of the functional language L, which is based on dynamically typed Standard ML extended by concurrency, logic variables, and complex synchronization conditions for patterns. His focus is on the efficient implementation of the core language, a performance analysis of the Mozart implementation of L, discussion of complex synchronization conditions, and the comparison of Mozart with a VM, based on functions. My focus is on the non-standard extensions of the functional core and their gracefully degrading integration into the VM.

Spaces and constraint inference engines which exploit the power of first class computation spaces are introduced and discussed in [90, 88, 89]. The focus of my work wrt. to spaces is their interaction with the different modules of the LVM and an analysis of implementation techniques for maintaining multiple bindings of variables.

The design of the object system for Oz is explained by Martin Henz [42]. The focus of his work is on the impact of concurrency for the design and the usage of an object system. Objects are a high-level abstraction built on top of the low-level concept of extension interface-types, which is described in my thesis.

Finite domain variables are an instance of transients, which allow for the efficient representation of constraints over finite domains of integers [91]. The applicability of the constraint solving capabilities of Oz was demonstrated with the scheduling workbench [116, 117, 118].

The addition of finite set variables [69, 68] as another instance of transients also uses the extension interface to integrate finite set values as an abstract data type.

For the efficient implementation of constraints, which implement propagation of information between constraint variables, propagators were introduced as a refinement of threads, which are completely implemented in C++ to avoid the overhead for the worker and the emulation [70].

Recently a distribution model [40, 107, 39] was developed and implemented, which allows the transparent distribution of the store among multiple sites.

Chapter 2

The language L

In this chapter we define the language L[1]. L is a subset[2] of Oz, which contains only a minimal core language and the extensions relevant for my work.

The syntax and semantics of L is based on Standard ML [63, 74]. A major deviation is the replacement of the static type system of Standard ML by a dynamic type system [97]. L extends Standard ML with logic variables and futures, flexible records and feature constraints, concurrency, and deep guard conditionals. The core of L is the same as the language described in [87].

2.1 Overview

We introduce the language in a concise and informal manner to show the requirements for our implementation at a high-level. We assume basic knowledge of Standard ML. We use evaluation rules and a graph rewriting model to define the semantics of the language.

In the following sections we describe a computation model and a graph model for the data structures of the language. After that we explain the semantics of core language and of our extensions, namely logic variables, futures, threads, by-need synchronization, record constraints, and deep guard conditionals. Finally we show the expressiveness of the language by discussing selected examples.

The core language of L is given in Figure 2.1. We use some syntactic sugar which is summarized in Figure 2.2.

In addition to the Standard ML syntax we use strings with ' as delimiters to denote atoms, which are first-class symbolic constants in L, e.g. `'person'`.

[1] The name of L is spelled out as **L**anguage.

[2] The language we define is subset with minor modifications for a better idealization and to simplify the explanation.

Expressions			
e	::=	y	identifier
	|	c	constant
	|	{c_1 = e_1,...,c_n = e_n}	record construction
	|	**fn** x => e	function definition
	|	e e'	application
	|	**let** d **in** e **end**	declaration
	|	**case** e **of** r_1|...|r_n	pattern matching
d	::=	**val** x = e	value declaration
	|	**name** N	name declaration
c	::=	i	integer constant
	|	a	atom constant
	|	M	name identifier
r	::=	p => e	match rule
p	::=	{c_1 = p_1, ..., c_1 = p_n}	record pattern
	|	c	constant pattern
	|	x	variable pattern

Core operators		
+,-,...	: int * int -> int	arithmetic
<,<=	: int * int -> bool	comparison
record	: (fea * T) list -> rec	dynamic records
select	: rec * fea -> T	field selection

Figure 2.1: Expressions and core operators of L.

Abbreviation	Core syntax	

`True`
`False`
`()`
	`name` `True`
	`name` `False`
	`name` `()`
		boolean true
		boolean false
		the singleton value

`x::y`
`[x1, ..., xn]`, $n \geq 0$
`(y1, ..., yn)`
	{`Head = x, Tail = y`}
	`x1::...::xn::nil`
	{`1 = y1, ...., n = yn`}
		list element
		list
		tuple ($n > 1$)

`let` d; d' **`in`** e **`end`**	**`let`** d **`in let`** d' **`in`** e **`end end`**	declaration sequence
$e; e'$	**`let val`** x `=` e **`in`** e' **`end`**	expression sequence
`if` y **`then`** e **`else`** e'	**`case`** y **`of`** `True` **`=>`** e`|`x **`=>`** e'	simple conditional

`fn` `p1 => e1`
`| ...`
`| pn => en`
	`fn` `x =>` **`case`** `x` **`of`** `p1 => e1`
	`| ...`
	`| pn => en`
		functional pattern

`fun` `x p1 => e1`
`| ...`
`| x pn => en`
	`val` `x =`
	`let` **`val`** `x = lvar () in`
	`unif (x,` **`fn`** `p1 => e1`
	`| ...`
	`| pn => en);`
	`x`
	`end`
		recursive functions

Figure 2.2: Syntactic sugar.

We use capitalized identifiers N, M for names. Names in L are first class citizens, which can be used as expressions and as field names of records. In the record construction and in patterns the field names are integers, atoms, and statically bound names.

Features are integers, atoms, and names used as field names of records.

Records can be dynamically constructed with the `record` operator. It takes a pair-list of pairwise distinct features and corresponding field values and constructs a record. Fields of records can be accessed with the `select` operator, which takes a record and a feature as arguments and returns the field value under the selected feature.

In the syntax we use the letter x resp. N for a binding occurrence of an identifier and y resp. M for a free occurrence. L has the same scoping rules as Standard ML. Patterns must be linear, i.e. all identifiers in binding position of record patterns are pairwise distinct. The synchronization conditions for patterns are explained in Section 2.8.

We use the usual Standard ML precedences and allow to use parentheses () to group expressions.

The references of Standard ML are called cells in L. We use the name cell in this thesis to avoid confusion with the reference nodes introduced at the LVM level (see Chapter 3).

Most of the language primitives can be nicely factored out from the expression syntax by using predefined functions, called operators. Figure 2.3 shows the operators for implementing our extensions. These extensions will be explained in the following sections.

The operators are shown with their type to guide the intuition of the reader. This type language is not used in L and differs from the type language in Standard ML. The type restrictions shown in Figure 2.3 are enforced at run time (dynamically) and not statically. The type names and the type hierarchy are listed in Figure 2.4.

We assume a type `T` at the top of the type hierarchy, which allows for example to use `T list` for lists of arbitrary values. Cells and other container types in L can contain arbitrary values.

2.2 Computation model

Computation in L is organized in computation spaces (see Figure 2.5). A *computation space* contains a number of threads executing over a shared store.

The *store* represents the data structures. The main focus of our work are the operations performed on the store. The control aspects are basically the ones known from Standard ML.

Exceptions		
`catch`	`: (()->'a)*('b->'a)->'a`	install handler
`throw`	`: 'a -> 'b`	raise exception
Cells		
`ref`	`: T -> ref`	new cell
`:=`	`: ref * T -> ()`	assign
`!`	`: ref -> T`	access
`exchange`	`: ref * T -> T`	exchange
Variables and futures		
`lvar`	`: () -> T`	logic variable
`unif`	`: 'a * 'a -> ()`	unification
`future`	`: 'a -> 'a`	future
Threads		
`spawn`	`: (() -> ()) -> ()`	thread creation
`waitOr`	`: T * T -> ()`	synchronization
`byNeed`	`: (() -> 'a) -> 'a`	by-need synch.
Tree constraints		
`featureC`	`: rec * fea * val -> ()`	feature constraint
`widthC`	`: rec * int -> ()`	width constraint
Deep guards		
`cond`	`: ('a->())*('a->'b)*(()->'b)->'b`	conditional

Figure 2.3: Extensions.

Type	Description
`T`	top
`rec`	record
`int`	integer
`fun`	function
`ref`	cell
`lit`	literal (`name` or `atom`)
`atom`	atom
`name`	name
`()`	singleton
`bool`	boolean value
`fea`	feature (lit or int)
`'a list`	list of 'a

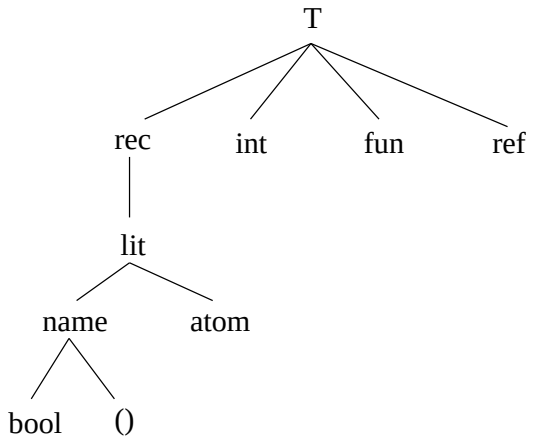


Figure 2.4: Type names and the type hierarchy of L.

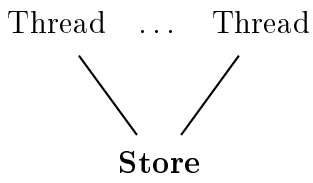


Figure 2.5: A computation space.

A *thread* is the sequential control for the evaluation of closures. A *closure* consists of an expression of the language and an environment. The *environment* defines how the free identifiers of expressions are bound to nodes in the store.

Threads are the only active entities in the computation model. The execution of a thread happens in steps. A step is defined by an evaluation rule for a closure. The evaluation rules for expressions and operators of the core language follow the Standard ML semantics and they are summarized in Section 2.4.

Threads communicate only via shared nodes in the store. Threads can read from and write into the store and they can synchronize on certain conditions of nodes.

The computation is *interleaved* and *fair*. Interleaved means that the execution steps are atomic and do not overlap. Fairness requires that a possible execution step of a thread will eventually happen.

2.3 The language graph

The semantics of our language is defined as a graph rewriting engine. The data structures of the language are modeled as nodes in a directed graph with labelled nodes and labelled directed links. This graph is called the language graph.

The language graph is built from units. A *unit* is a labelled node with a finite number of links. Figure 2.6 shows all units of our language.

A unit can be added to a graph by connecting its open links to already existing nodes in the graph. When a unit is added to a graph no dangling references remain. Figure 2.7 shows an example of a graph.

In our language it is not possible to create a cycle in the graph by adding new units. Cycles can be created through explicit graph rewriting steps, which are cell assignment (see Section 2.8) and variable binding (see Section 2.6).

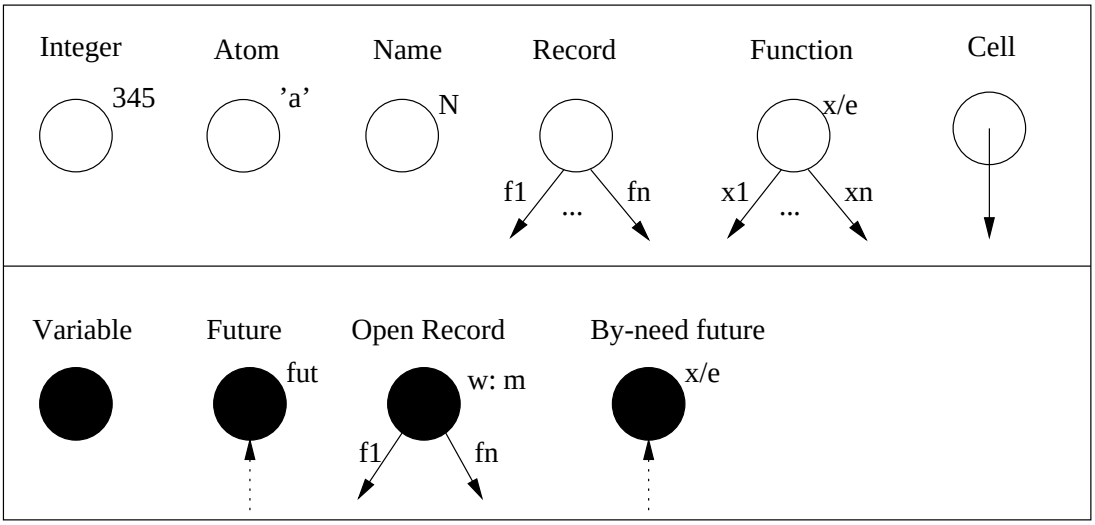
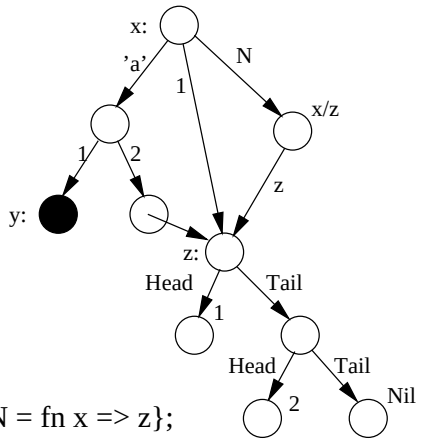


Figure 2.6: Units of L.




```
con N;
val y = lvar ();
val z = 1::2::nil
val x = {'a' = (y, ref (z)), 1 = z, N = fn x => z};
```

Figure 2.7: An example of a language graph.

2.3.1 Values

The graph represents *values*. Values are stateless mathematical entities. The values of L are primitive values (numbers and symbols), and (infinite) trees with labelled directed edges. The leaves of these trees are the primitive values.

Primitive values Numeric values and symbolic values are primitive values of L. For every primitive value a unit exists which is labelled with this value. With these units leaf nodes of trees with no departing link are created.

The numeric values of L are integers $0, 1, -1, 2, -2, \ldots$ of arbitrary size, with the usual mathematical meaning.

The symbolic values are atoms and names. *Atoms* are finite strings over a finite set of characters. *Names* are an infinite set of distinct values with no further structure.

An essential property of names is that they are only available through a generator. Whenever a name unit is added to the store it obtains a fresh name, which is distinct from all existing names in the store.

We consider in many aspects cells and functions (which are introduced later) also as primitive values similar to names, e.g. cells and functions can be leafs of trees.

Records Compound trees are represented in the store using record units. A *record* is a node with a finite number of departing links. These links are labelled with pairwise distinct features. A *feature* is an integer, an atom, or a name.

The set of features is called the *arity* of the record. The number of features is called the *width* of the record.

The pair of a feature and the node at the end of the link labelled with this feature is called a *field*. The feature is then called the *field name* and the node is the *field value*. The operation to traverse a link from a record is called *field selection* or *field access*.

Records in our language are flexible records, which are very different from static records of Standard ML. In L features are first-class values and it is possible to select a field without knowing all the other features of the record. It is furthermore possible to create records whose feature are not known at compile time, e.g. feature passed as arguments to functions.

Figure 2.8 shows how a tree can be constructed from units.

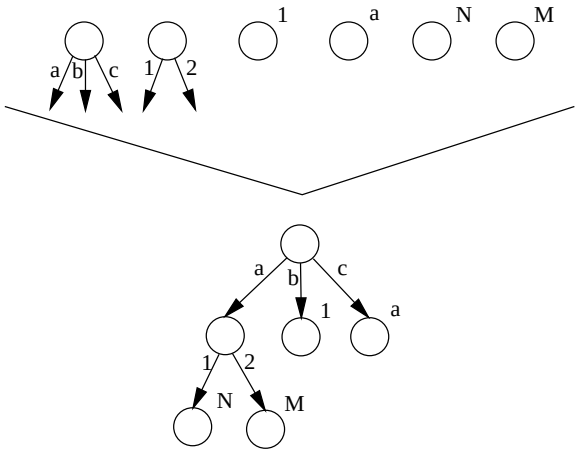


Figure 2.8: Records and trees.

2.3.2 Invariants for graph rewriting

Exactly three graph rewriting operations are performed during the execution of threads:

node creation New nodes can be created and added to the graph.

binding Transient nodes can be bound to other nodes. In the graph model this operation superimposes the new node onto the transient nodes. This makes the transient node transparent. The transient node disappears from the graph and all incoming links are redirected to the new node. Another metaphor for binding a node v to a node n is that all edges to v are redirected to n (see Section 2.6).

assignment Cells can assigned to new values. In this case the content link of the cell is redirected to a new node.

These strong invariants on graph rewriting simplify the reasoning about L programs. They are also very useful for building parallel and distributed implementations, but in our sequential and imperative implementation of the LVM these invariants are not exploited.

2.4 Sequential execution

In this section we explain the execution of a single thread. A thread is the sequential control for the evaluation of closures.

A closure of an expression e is a pair of an environment u and the expression e written as $\langle u,e \rangle$. The environment is a mapping of every free identifier x in the expression e to a node n in the store. In the following we use the notation x also for the node bound to x in the environment u. Furthermore we use the notation x also for the value of the node if it represents a primitive value. The context allows usually to disambiguate the different meanings easily.

An execution step can side effect the store and evaluates a closure. The evaluation of a closure has one of the following outcomes:

- It evaluates to a node in the store.
- It reduces to one or more new closures.
- It raises an exception.

In the following we use formulations like "if x is a node of type ..." then this means that

- The thread has to synchronize on x until it is no variable and no future. Synchronization is explained in section 2.8 where threads are introduced.
- If the node x is of a different type an exception is raised. The exception mechanism of L is introduced in section 2.5.

2.4.1 Data structures

Identifiers The closure $\langle u,y \rangle$ evaluates to the node bound to y in u.

Atoms The closure $\langle u,a \rangle$ adds an atom node with label a to the store and evaluates to this node.

Integers The closure $\langle u,i \rangle$ adds an integer node with label i to the store and evaluates to this node.

Record construction The evaluation of $\langle u,$`{`c_1 `=` y_1`, ...,`c_n `=` y_n`}`$\rangle$ tests first if $c_1, \ldots, c_n$ are pairwise distinct features.

If the test succeeds a record node with the arity $\{c_1, \ldots, c_n\}$ is added to the store. For all $i \in \{1, \ldots, n\}$ the link labelled with the feature c_i is connected to the node y_i. The record construction evaluates to this node.

If $y_1, \ldots, y_n$ are not pairwise distinct features the record construction raises an exception.

2.4.2 Functions

A function is a closure of a function definition expression **fn** x **=>** e. We use abstractions $\lambda x/e$ as compact notation for the function definition.

Functions are represented with function units. A function unit is a node labelled with a function definition and labelled links for the free identifiers in the function definition.

Function definition The evaluation of the function definition $\langle u$,**fn** x **=>** $e\rangle$ adds a function unit to the store which labelled with the function definition. The links for the free identifiers of the abstraction are connected to their binding in u. The function definition evaluates to the just added function node.

Application The evaluation of the application $\langle u,e\ e'\rangle$ first evaluates $\langle u,e\rangle$ to y and then $\langle u,e\rangle$ to y'. Then it tests if the y is a function.

If y is a function labelled with an abstraction $\lambda x/e''$ the application evaluates to the closure $\langle u',e''\rangle$. The new environment u' contains the bindings of the free identifiers of the abstraction and the binding of the formal argument x to the actual argument y'.

2.4.3 Pattern matching

The evaluation of $\langle u$,**case** y **of** r_1**|...|**$r_n\rangle$ sequentially tests if y matches one of the patterns $p_1,\ldots,p_n$ in the match rules $r_1,\ldots,r_n$.

The record pattern **{**c_1 **=** p_1**, ...,** c_n **=** p_n**}** **=>** e matches if y is a record with the arity $\{c_1,\ldots,c_n\}$. Then the field values are sequentially matched against the patterns $p_1,\ldots,p_n$. If all these matches are successful the case expression evaluates to the closure $\langle u',e\rangle$, where u' is derived from u by adding the bindings for the binding identifiers in the patterns.

The constant pattern c **=>** e matches if the value of y is equal to the primitive value c. Then the case expression evaluates to the closure $\langle u,e\rangle$.

The variable pattern x **=>** e matches always and evaluates to the closure $\langle u',e\rangle$, where u' is derived from u by adding the binding of the identifier x to the node y.

2.4.4 Declarations

The evaluation of the value declarations $\langle u,$**let val** x = e **in** e' **end**$\rangle$ creates two new closures: the expression $\langle u,e\rangle$ and the abstraction $\langle u,\lambda x/e'\rangle$. The expression $\langle u,e\rangle$ is evaluated first and then the abstraction $\langle u,\lambda x/e'\rangle$ is applied to result of this evaluation.

Sequential execution of closures can be explained such that the thread has a stack of closures to execute and the value declaration pushes the abstraction $\langle u,\lambda x/e'\rangle$ on this stack and evaluates first the expression $\langle u,e\rangle$. Only when this has finished, the closure found on the stack is executed.

The evaluation of the name declarations $\langle u,$**let name** N **in** e **end**$\rangle$ adds a new name node to the store and evaluates to the expression $\langle u',e\rangle$, where u' is derived from u by adding the binding of N to the new name.

2.4.5 Core operators

The arithmetic operators `+`, $-$, $*$, `div`, `mod`, $<$, $<=$ evaluate with their usual mathematical semantics. We use the infix notation for these operators.

The `select` operator takes two arguments a record and a feature and evaluates to the field value of the record selected by the feature.

The `record` operator allows to create records dynamically. It takes a list of pairs containing field names and field values as argument an creates a record.

2.4.6 Syntactic convenience

Sequences A sequences of declarations can be combined into one declaration using a semicolon as separator.

let d; d' **in** e **end** is an abbreviation for **let** d **in let** d **in** e **end end**.

A declaration **let val** x = e **in** e' **end** can be simplified into the sequence $e;e'$ if the identifier x does not occur free in e'

Tuples A record with an arity of $\{1,\ldots,n\}$ is called a tuple. Tuples are eminent, because they are optimized in the LVM. A tuple {1 = y_1,...,n = y_n} ($n > 1$) can be written as (y_1,...,y_n). A tuple with two fields is called a pair.

Names We assume that the following identifiers are bound to distinct names in every execution environment and cannot be redeclared:

- **true** and **false** for boolean values.
- () for the singleton value.
- **Head**, **Tail**, and **Nil** for constructing lists.

Lists As a convenient syntax for lists the notation `x::y` is used for the record written as {`Head:x, Tail:y`}. The empty list `Nil` can be written as `[]`. A list with a fixed number of elements $x_1, \ldots, x_n$ can be written as $[x_1, \ldots, x_n]$.

The tuples and list syntax is also allowed in patterns and expands to the corresponding record pattern.

Functions The core syntax has only single argument functions. Multiple arguments are passed as tuples. For convenience the syntax

```
fn p1 => e1 | ... | pn => en
```

is an abbreviation for

```
fn x => case x of p1 => e1 | ... | pn => en
```

This allows for example to write a function with two arguments as

```
fn (x,y) => ...
```

Boolean conditional **`if`** y **`then`** e **`else`** e' is an abbreviation for **`case`** y **`of true =>`** e | x **`=>`** e', where x is an identifier not occurring free in e'.

2.5 Exceptions

Exceptions are a powerful concept to handle errors and to built non-standard control structures [28, 29]. In this section we explain the semantics of the exception mechanism in L.

An exception is a condition detected during the evaluation of an expression which cannot be handled locally. In such a situation an exception is *raised*.

An *exception handler* can be installed for an expression. When an exception is raised during the evaluation of the expression it is *caught* by the exception handler. When an exception is caught the control is transfered to the handler.

Information can be passed from the point where an exception is raised to the handler of the exception. This information is called the *exception value*, which is usually abbreviated to "the exception". In L the exception value is an arbitrary node in the store. The handler is a function in L and when the exception is caught this function is applied to exception value.

Exception handlers can be nested. In this case the innermost handler catches the exception and calls its handler. The exception handler is deinstalled when it catches an exception, i.e. further exceptions are caught by the next handler.

Threads install a default exception handler before evaluating an expression, such that exceptions cannot escape their thread. The default exception handler typically prints a message[3].

2.5.1 Exception handlers

The `catch` operator is applied to a pair of two functions (x, y). The application of the handle operator installs the exception handler y during the evaluation of the function x applied to the singleton value `()`.

When the evaluation of x returns a node n, the exception handler is removed and the `catch` operator also evaluates to the node n.

When an exception is raised during the evaluation of x the exception handler is removed and the `catch` operator evaluates to the application of the handler function y to the exception value.

2.5.2 Raising an exception

The `throw` operator has an exception value as argument. The evaluation of this operators never returns, but transfers control and the exception value to the innermost installed exception handler.

Exceptions are raised implicitly, when an error occurs, e.g. record construction raises an error if its features are not pairwise disjoint and the application raises an exception if the first argument is no function.

2.5.3 Discussion

The main problems and the design space for exception handling have been known since a long time [28, 29]. The exception mechanism of L is similar to the one defined in Standard ML.

[3]Failure exceptions in spaces are handled specially (see Section 2.10).

Typed exceptions Many languages like Standard ML [63], C++ [16, 53] and Java [30] use typed exceptions and the exception mechanism is extended such that an exception handler is only used if it matches the type of the exception value.

In L this can be expressed by writing exception handlers such that they analyze the exception value. In the case that they cannot handle an exception they simply re-raise it.

Finally A *finally expression* allows to protect the evaluation of an expression such that independent of the success or failure of this evaluation a cleanup expression is evaluated. This can for example be used to ensure that allocated resources are released.

In L finally can be implement with the following function:

```
val finally = fn (body, final) =>
let
   name Suc; name Exc;
   val result = catch ( fn ()  => (Suc, body ()),
                        fn exc => (Exc, exc) )
in
   final ();
   case result
   of (Suc, value) => value
    | (Exc, exc) => throw exc
end
```

The finally function is applied to a pair of two functions. The first function is the body which is executed and might raise an exception. The second function is the final cleanup which is applied regardless of the success or failure of the first function.

2.6 Logic variables

A logic variable is a place holder for a not yet known value. Logic variables were introduced as a language primitive with the language Prolog [55, 56, 71] as the foundation for logic programming. Logic variables have been also recognized as powerful concept for synchronization in concurrent languages [94]. For constraint logic programming logic variables have been extended with attributes to represent domain information.

A logic variable is represented with a variable unit in the store. A variable unit is a node with no departing links. The `lvar` operator adds a variable node to the store and evaluates to it.

2.6.1 Unification

The graph rewriting operation on variables is *binding*. A variable can be bound to another node of the store. Binding a variable makes it *transparent*, i.e. the variable node disappears and all incoming edges are redirected to the node it is bound to.

Binding is not a primitive operation in L, but it is implicitly performed by unification. *Unification* is a complex graph rewriting operation to make two nodes equivalent wrt. to the equivalence relation defined below. If it is possible the unification performs a minimal number of variable bindings until two nodes are equal. If this is not possible the unification fails.

We first define an equivalence relation on nodes. Then we present an unification algorithm.

Equivalence of nodes The *equivalence relation* of nodes is defined as the greatest relation, which satifies the following conditions:

- Every node is equivalent to itself.
- Two primitive nodes are equivalent iff they represent the same value.
- Two record nodes are equivalent iff they have the same arity and if the equivalence relation holds for every pair of corresponding field values.

The unification algorithm The unification algorithm implemented in the LVM is a variation of the unification algorithm for rational trees resp. cyclic structures [18, 98, 38]. An overview of the algorithm is given in Figure 2.9.

The unification algorithm maintains a todo stack and an explored set. The todo stack contains pairs of nodes which must be unified. The explored set contains pairs of already unified records. Initially the explored set is empty and the todo stack contains the pair of the two nodes to unify. In every step of the unification algorithm a pair of nodes is popped from the todo stack and *processed*. The algorithm terminates if the todo stack is empty and returns a *termination status*, which is either **succeed** or **fail**.

Two nodes are processed in the following ways

- If both nodes or their values are the same, or if they are in the explored set the processing step succeeds and nothing needs to be done.
- If both nodes are records with the same label and arity, then they are added to the explored set and corresponding pairs of fields are pushed on the todo stack.

```
INPUT:
  node n1;
  node n2;
OUTPUT:
  enum {SUCCEED, FAIL} status;
INIT:
  todo = new stack();
  todo.push(n1, n2);
  explored = new set();
  status = SUCCEED;
LOOP:
  while (!todo.isEmpty())
     (a, b) = todo.pop();

     if (a != b)
        if (isVar(a))
           bind(a,b)
        else if (isVar(b))
           bind(b,a)
        else if (member({a,b},explored))
           // nothing
        else if (isRecord(a) &&
                 isRecord(b) &&
                 arity(a) == arity(b))
          explored.add({a, b});
          for (f in arity(a))
             todo.push(select(a,f), select(b,f))
        else
          explored.add({a, b});
          status = FAIL
```

Figure 2.9: A graph unification algorithm

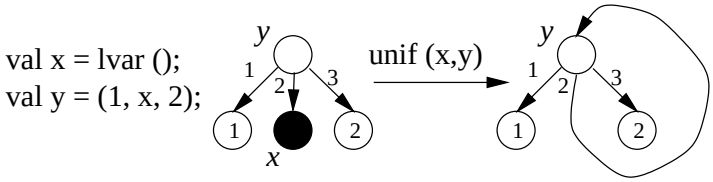


Figure 2.10: Binding variables.

- If a node is a variable it is bound to the other node.
- In all other cases the nodes are put into the explored set and the unification status is set to failed.

The algorithm terminates because in every step

1. the open set becomes smaller or
2. an element is added to the explored set or
3. a variable is bound.

The graph is finite and no new nodes are added during the unification. Therefore the number of elements in the explored set must be finite and only finitely many bindings of variables can be done. This means that eventually the open set must be empty.

Note that the unification continues even in the case that failure is detected. We do this to ensure that the unification algorithm is independ of the order in with the fields of records are explored.

The `unif` operator The `unif` operator is applied to a pair of nodes and performs their unification. If the unification fails the evaluation of the `unif` operator raises an exception, else it evaluates to the singleton value.

The exception raised by the `unif` operator is specially marked, because in nested computation spaces it is treated in as disentailment condition (see Section 2.10). The exception is called a failure exception.

Binding variables can introduces cycles into the graph. Figure 2.10 shows an example of a record y with a variable x under feature 2 and the cycle introduced by the unification of x and y.

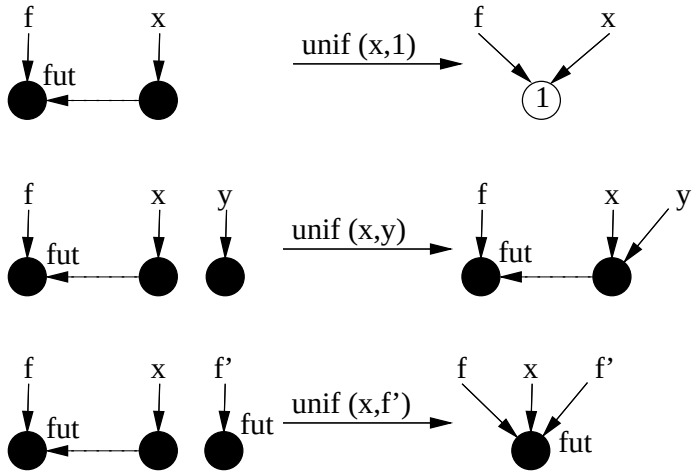


Figure 2.11: Unification with futures

2.7 Futures

Futures are read-only views of logic variables. With futures the scope where a variable can be bound can be statically limited.

When the variable is bound to a non-variable a future of this variable is bound simultaneously to the same node as the variable. Futures are represented in the store as a future unit and variables are extended with a link to their future.

Figure 2.11 shows some interesting cases for binding variables with futures. In the first case when the variable is bound to a determined node both the variable and its future are superimposed by this node. The second case show what happens when a variable is bound to another variable: only the variable is bound the future is unchanged except that it is now a future for a different variable. The third case shows that if the variable is bound to another future f' this future is superimposed on the variable x and its future f.

The `future` operator takes one argument. If this argument is a variable which does not yet have a future, a future node is created. The `future` operator evaluates to this future of the variable. If the argument is no variable the future operator evaluates to its argument.

Extending unification Futures require to extend the unification algorithm. When a future and a determined node are unified it is not allowed to bind the future. In this case it is not yet decidable, if the future and the determined node are equivalent or not. Therefore the unification has a termination status to signal this case, which is called **suspend**.

A second aspect of futures is the extension of the equivalence relation such that a future is equivalent to its variable, i.e. `unif (x, future (x))` must succeed.

To avoid that the semantics of unification depends on the order how the nodes are processed the unification algorithm continues after detecting the suspend status. Thus it is possible that later on failure is detected.

The pairs of nodes which could not be unified due to futures are collected and when the unification does not fail they are saved to restart the unification, when one of the futures is bound. The next section on threads explains how threads are suspended and resumed.

An interesting case is `unif ((f,x),(1,1))`, where `f` is the future of `x`. In this case the unification algorithm first discovers that the equivalence of `f` and 1 is not decidable, but later `x` and simultaneously `f` is bound to 1. In this case the unification is restarted and in this second run it returns successfully.

Transients and determined nodes We call variable and future nodes in the store *transients*, because they are only temporarily visible and disappear when they are bound. Non-transient nodes are called *determined*.

Discussion Futures are useful for example to implement ports [49] with safe streams. A safe stream is a stream, where the open tail is a future, which cannot be corrupted by readers. Only the writer has access to the variable behind this future.

Note that the name future is used with various meanings in the literature. Our futures are only concerned with the read-only aspect of logic variables. Futures in the style of Multilisp [34] are related to futures with by-need synchronization and they are discussed below.

2.8 Concurrency

In this section we explain how concurrency is integrated in L.

2.8.1 Threads

Multiple threads of control can be created with the `spawn` operator. The `spawn` operator is applied to a function as only argument and creates a new thread which has as the initial closure the application of this function to the singleton value.

After the creation of the new thread the `spawn` operator evaluates to the singleton value without any synchronization on the new thread. Communication and synchronization only happens through nodes shared with the spawned function. A thread can for example communicate with other threads through the binding of variables and cell exchanges.

Threads are executed concurrently, they are independent, and they are scheduled fairly. Concurrency in L means that the evaluation steps are interleaved, but do not overlap. The threads are independent in the sense that the only connection between them is through shared nodes in the store. Fairness requires that if an evaluation step on a thread is possible it will eventually happen.

2.8.2 Synchronization and suspension

Threads synchronize on the determination of transients. We explain the synchronization technique with the `waitOr` operator. The `waitOr` operator is applied to two arguments and evaluates to the singleton value, if at least one of its arguments is a determined node.

When both arguments are transients the `waitOr` operator cannot be evaluated and blocks the further execution of its thread. The thread is said to *suspend* on the transient arguments. The `waitOr` operator and the suspended threads becomes executable if one of the transients is bound to a determined node.

The synchronization on transients is a monotonic condition. If an evaluation of an expression is possible at a certain moment, it can be evaluated also after any change in the store. This holds because the binding of a transient is a monotonic operation[4].

The `waitOr` operator allows for example to express timeouts. For example by waiting concurrently on a thread producing a result and another thread producing a timeout condition.

Wait The function `wait` defined below is a simplification of the `waitOr` operator which suspends on a single argument.

```
fun wait x = waitOr (x, lvar ());
```

Other suspensions Any operator which expects a determined value suspends when it is applied to a transient, e.g. arithmetic operators suspend until both arguments are determined and the application $e\ \ e'$ suspends until e is determined.

[4] In spaces bindings are retracted and the monotonicity might be violated, but it makes only a difference when the space fails anyway (see Section 2.10).

In our language we use a very simple synchronization condition for pattern matching. Pattern matching in our language is flattened out and suspends if one of the sequential simple matches is not decideable. Scheidhauer [87] analyses more complex synchronization conditions, where for example the match

```
case (x,x)
of (1,2) => e1
 | y => e2
```

reduces to $e2$ even when x is not determined. In our language this example is equivalent to

```
case (x,x)
of (x1,x2) =>
  case x1
  of 1 =>
    case x2
    of 2 => e1
     | y => e2
   | y => e2
 | y => e2
```

2.8.3 By-need synchronization

A different kind of synchronization is by-need synchronization, which essentially allows for lazy programming.

To explain it we first define the notion of a requested transient. A transient is *requested* if a thread is suspended and waits until this transient is bound. For example if x is a variable and a thread tries to evaluate $x+1$ then x is requested.

By-need synchronization is introduced with the `byNeed` operator. The `byNeed` operator is applied to a function f and evaluates to a future for a newly created variable x. When this future is requested a new thread is spawned which unifies the variable x with the result of the application of the function f to the singleton value. Figure 2.12 shows how a by-need future is bound when it is requested.

Discussion The by-need synchronization in L is similar to the concept of futures in Multilisp [34, 26]. Multilisp distinguishes two operators for futures. `(future E)` returns a future and starts the computation to evaluate `E` in a concurrent resp. parallel thread. With `(delay E)` the evaluation of `E` only starts when the value of the future is requested.

Futures are proposed as extensions for C++ and Java [57, 85]. In these proposal futures are not defined as transparent data types, but explicit operations are required to cast a future into a determined value. A major problem of this

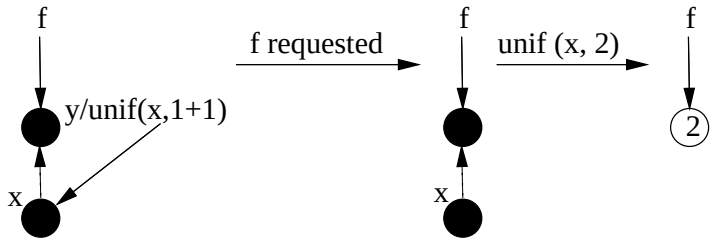


Figure 2.12: By-need sychronization.

approach is that for every function a decision has to be made if futures are allowed or not. This especially requires to redesign all libraries.

By-need synchronization allows to easily express the lazy functional programming style as promoted by lazy functional languages, e.g. Haskell [75]. In Section 2.11 the lazy creation of hamming numbers is shown as an example.

2.8.4 Cells

Cells are the only stateful data structures of L. In connection with concurrency stateful nodes must be handled carefully, e.g. concurrent access and assign operations must be properly synchronized.

The `exchange` operator is a generalization of the assignment operator `:=` of Standard ML. `exchange` assigns a new node to the cell and returns the old content of the cell in a single atomic step. This extension is essential because it provides a powerful synchronization primitive.

Locks The `exchange` operator with logic variables allows to express locks for mutual exclusion. A lock is implemented as a cell where the content indicates if the lock is free or not. The usage of the cell is defined such that the operation to acquire the lock exchanges the content of the cell with a fresh variable and waits until the old content is determined. When the lock is released the just created variable is bound to the singleton value.

```
(* create a new lock *)
fun newLock () = ref ();

(* aquire lock, execute body, release lock *)
fun sync (lock,body) =
    let val new = lvar ();
        val old = exchange (lock, new)
    in
        case old of () =>
```

```
        let val result = body ()
        in
            unif (new, ());
            result
        end
    end;
```

The function `newLock` creates a cell with the singleton value as initial content. The function `sync` takes a lock and a procedure as arguments. It exchanges the content of the cell with a fresh variable and waits until the old content is determined. The the body is executed and with the unification of the fresh variable with the singleton value the lock is released.

Without logic variables the exchange primitive is already expressive enough to implement locks, but the implementation does not have the following properties of our implementation

- The implementation is simple.
- The thread which must wait for a lock needs no busy waiting.
- No starvation can happen. Every thread competing for the lock will eventually obtain it, when it is released properly.

Cell access With logic variables `access` can be expressed with the `exchange` operation.

```
fun access cell =
let val new = lvar ();
    val old = exchange (cell, new)
in
    unif (new, old);
    new
end
```

In L `access` is a primitive operator, because it has a different semantics wrt. multiple computation spaces. The content of a cell can be accessed, but not changed when the cell is global in a space (see Section 2.10).

Abstract data types We consider cells in this thesis because we want to explain how the VM supports built-in abstract data types, which are a generalization of records and cells. The built-in abstract data types are for example the data-structures on which the object implementation of Oz is built. Henz [42] discusses how an object system can be build on top of a concurrent constraint language with cells.

An example of an such an abstract data type is a bit array. A naive implementation which represents a bit array as a list of cells with content 0 or 1 is given below.

```
let
   name BitArray;
   fun unbox (b, i) =
       nth (select (b, BitArray), i);
   fun box (id, l) =
       { id = (), BitArray = l };
   fun new1 size =
      if size>0
      then ref 0::new1 (size-1)
      else [];
   fun new size =
      let con Id
      in
         box (Id, new1 size);
      end

   fun set   (b,i) = exchange (unbox (b,i), 1);
   fun clear (b,i) = exchange (unbox (b,i), 0);
   fun get   (b,i) = ! (unbox (b,i));
in
   {'new'=new,     'set'=set,
    'clear'=clear, 'get'=get}
end
```

In Chapter 4 we show how efficient native C/C++ implementations of such abstract data-types can be easily integrated into the LVM with a generic extension mechanism.

2.8.5 Discussion

Java/POSIX threads Threads in L are very different from threads in Java [30]. The semantics of threads in Java is driven by the available technology in modern operating system. These are typically based on the POSIX 1003.1 standard [15, 43].

The POSIX standard cares a lot about memory cache effects and makes explicit that only when using sychronization primitives the (possibly cached) memory is updated. In L no caching effects are visible at the language level. If an implementation uses memory caches it has to guarantee that the illusion of a unique store is not violated.

POSIX does not specify a scheduling policy. The standard allows but does not require that conforming implementations support different scheduling methods. This means that for example preemptive scheduling is platform dependent and an application cannot rely on fairness assumptions.

Parallelism Concurrency does not prevent parallelism, but a parallel implementation has to preserve the invariant that overlapping evaluation steps are not visible [80].

2.9 Feature constraints

In this section we extend records such that it becomes possible to represent incomplete partial information about branches in trees.

With records and logic variables it is already possible to describe partial trees, where some of the nodes are not yet known. Feature constraints extend this model and allow to describe record nodes where the features are partially known.

Feature constraints allow to represent for example information about paths in a tree without knowing the whole shape of tree, i.e. the arities of some record nodes are underspecified. Feature structures in natural-language processing systems are an example where this is useful.

Records and feature constraints in L are based on records for logic programming [98] and on the work done on ψ-terms in LIFE [3, 78]. The implementation of efficient record constraints for concurrent constraint programming in the Oz system was described in [108]

In the following sections we first describe a generic set of constraints over trees and show then how records and feature constraints of L fit into this model.

2.9.1 Constraints over trees

The structure underlying the tree constraint system [98] of L contains infinite sets of features, integers, and rational trees. Rational trees are possibly infinite trees with directed links labelled with features. The constraint system is closed under conjunction and existential quantification of domain variables. The constraint system has the following basic constraints.

- The *feature constraint* $feature(t, f, t')$ states that t is a rational tree with a link to the tree t' which is labelled with the feature f.

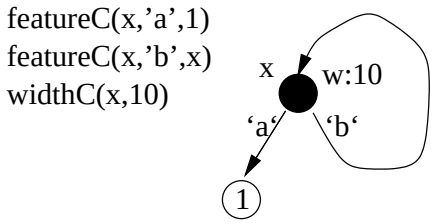


Figure 2.13: An example of an open record.

- The *width constraint* $width(t, n)$ ($n \in \{1, 2, ...\}$) states that t is a rational tree with exactly n outgoing links.
- The *equality constraint* $t = t'$ states that the trees t and t' are equal.

In full generality this constraint system is not analyzed yet [102, 103, 10]. In the following we explain the implemented subclass of L.

2.9.2 Open records

Records as introduced in Section 2.3 above are an instance of the tree constraint which is restricted to constraints of the form:

$$\exists t, t_1, \ldots, t_n \, width(t, n) \wedge \forall i \in \{1, \ldots, n\} \, feature(t, f_i, t_i).$$

The features f_i and the number n in this constraint are constants and the features must be pairwise distinct.

Open records are records where not all features are known. Open records are described by the constraints $width(t, n)$ resp. $feature(t, f, t')$, where the width n and the feature f are constants.

In the store open records are represented as variables with attributes. Attributes allow to attach information to a variable. The semantics of some operations, e.g. unification, is extended for variables with attributes.

Variables representing open records have the attributes *width* and *fields*. The width attribute if defined contains a number and the fields attribute contains a set of pairs of a feature and a node (see Figure 2.13).

The constraints on the attributes of a variable are

- Every feature occurs at most once in the fields attribute.

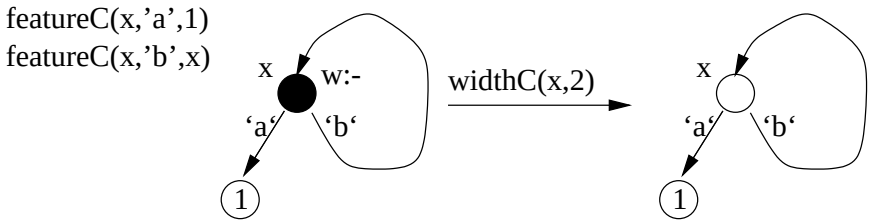


Figure 2.14: Closing an open record.

- The number of elements in fields attribute is less than the value of the width attribute.

An open record is automatically *closed*, when its width attribute becomes equal to the number of elements in the fields attribute. Closing means that the variable is bound to a record, where the fields of the records are exactly the elements of the fields attribute (see Figure 2.14).

The `featureC` and `widthC` operator implement the feature resp. width constraints and the unification is extended to support the equality constraint on open records.

The `featureC` operator The `featureC` operator is applied to three arguments (x, f, y) and suspends until f is a feature and x is not a future. The following cases occur

Condition	Action
x is a variable which does not contain the feature f in its field attribute.	The feature f and the field value y are added. Implicitly the open record may be closed.
x is a variable with the feature f and field value y' in its field attribute or x a record with a feature f and the field value y'.	Then the `featureC` operator reduces to the unification of y and y'.
Otherwise.	A failure exception is raised.

The `widthC` operator The `widthC` operator is applied to two arguments (x, n) and suspends until the first argument is no future and the second argument is a positive integer.

Condition	Action
x is a variable which does not have the width attribute and not more than n entries in the field attribute.	The width attribute with value n is added to the variable. Implicitly the open record may be closed.
x is a variable with width attribute equal to n or x is a record with width n.	Nothing needs to be done.
Otherwise.	A failure exception is raised.

Extending the unification algorithm The unification algorithm must be extended to support open records. If an open record `x` is unified with another node `y`, then it bound as usual and its attributes are imposed to the new binding. Imposing means that the attributes of `x` are added to the node `y` as if the `widthC` and `featureC` operators for these attributes are applied.

- If `x` has a width attribute n, then `widthC (y, n)` is executed.
- For all features `fi` with field values `zi` in the fields attribute of `x` the operator `featureC (y, fi, zi)` is executed.

2.10 Spaces

Multiple computation spaces are the basis for building flexible search engines in the concurrent constraint programming paradigm [90, 91, 88, 89]. In this thesis we focus on the implementation of entailment, disentailment and merging of spaces. Therefore we define one operator, namely the deep guard conditional, which requires exactly the abilities to detect entailment and disentailment and to merge spaces.

We first define a multiple store graph model with introduces situated nodes. After that we explain the deep guard conditional operator.

2.10.1 The multiple store graph model

A computation space is a number of threads execution over a shared store. The execution of a thread can create new *subordinated* computation spaces. The new computation space is initialized with a copy of the current store and an initial thread.

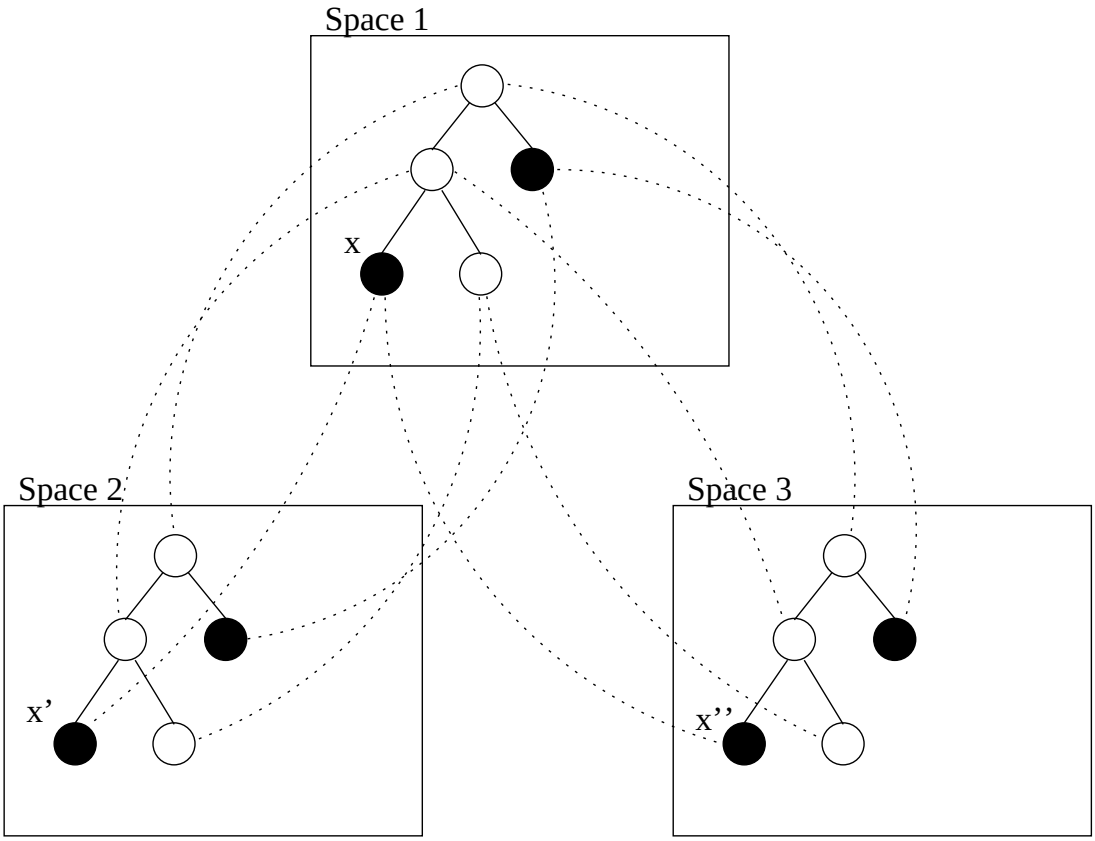


Figure 2.15: A tree of computation spaces.

Every node in the copy is linked to its original. This is essential to define propagation and merging. The basic invariant between spaces is that the graph in a subordinated space is a extension of the graph in the store of its parent space. The intuition should be that subordinated spaces see every change in their parent's store, but not vice versa.

With this construction a tree of computation spaces can be build (see Figure 2.15). The figure shows how a graph and its copy in a subordinated space are linked together. The top-most space is called *root* or *toplevel space*.

When a space is created a new variable is created in this space, which is called the *root variable*. The root variable is used to communicate computation results between a space and its parent space. The initial thread executes a function which is applied to this root variable.

Situated nodes The theoretical foundation [95] of computation spaces is based on a declarative semantics where the store is modeled as a constraint with existen-

tially quantified variables. In the graph model we replace the notion of existential quantification with the notion of situated nodes.

Transients and cells are situated nodes. The space where a situated node is created is called the *home space* of a node. A situated node is called a *local node* in its home space and a *global node* in subordinated spaces. Figure 2.15 shows a global variable x' in the copy and its corresponding local variable x in the original space.

In the following we restrict situated nodes to logic variables. Other types of situated nodes are introduced later.

Store invariant The store invariant ensures the consistency of stores in a tree of computation spaces. It is defined such that the graph in a subordinated space is an extension of the graph in its parent:

- A subordinated graph contains all nodes and links of the graph of its parent. When new nodes are added then these nodes are copied to subordinated spaces. The copies preserve the connection to their original nodes.
- A subordinated graph can contain additional units and links.
- Global variables in spaces can be bound. Such a binding is called *speculative*. A speculative bindinge can be retracted.

Binding and propagation When a variable is bound this binding is *propagated* to all subordinated spaces. Propagation ensures that the first requirement of the store invariant holds.

Propagating a binding retracts already existing speculative bindings in subordinated spaces and replaces these speculative bindings with the new binding.

Retracting a binding means that an assumption made during a previous unification is invalidated. To ensure that no information is lost a new thread is created in the subordinated space which unifies the old and the new binding.

Figure 2.16 shows how a binding is propagated to a subordinated space.

Binding order When two variables must be bound and one is global and the other is local, the local variable is bound to the global variable. This ensures that a minimal number of speculative bindings are done per space.

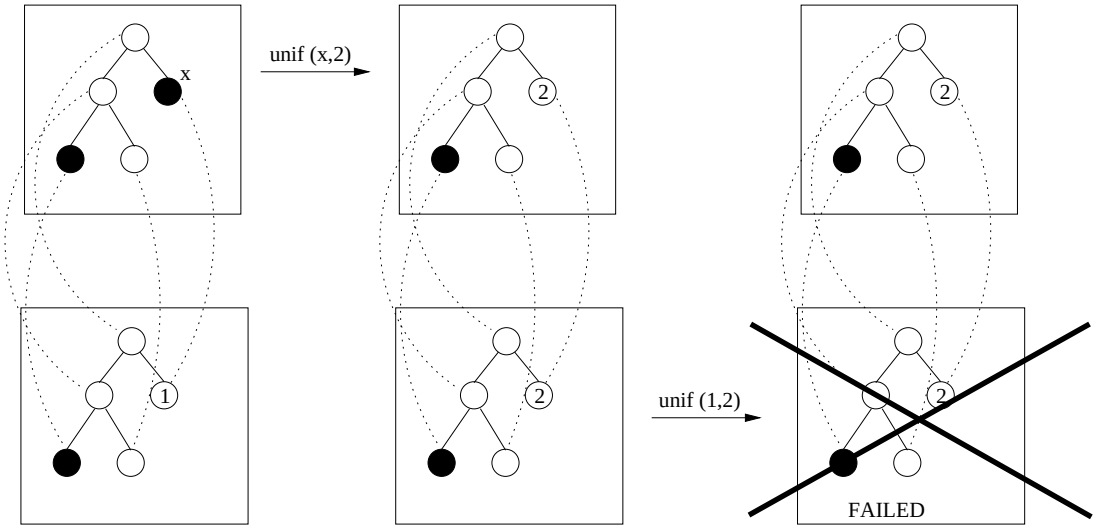


Figure 2.16: Propagation of a binding.

2.10.2 Entailment

The distinction of local and global variables is essential to decide entailment of a space. A space is entailed if

- all threads are terminated and
- the constraint represented in a store is entailed by the constraint of its parent store.

The second part of the entailment condition expressed in terms of our graph model means that no global variable is speculatively bound in the store.

Figure 2.17 shows a simple example how entailment is detected after propagation. In step (1) the unification of a local variable z with a global variable x binds the local variable. In step (2) a speculative binding of x to 1 is added. In step (3) x is bound in its home space to y. This binding is propagated to the subordinate space. This requires a unification step, which leads to the speculative binding of y to 1. In step (4) y is bound in its home space to 1. After the propagation of this binding the subordinated space is entailed.

2.10.3 Disentailment

The detection of disentailment is build on top of the exception mechanism of L. When the `unif` or another constraint operator detects failure they raise a special exception, called a *failure exception*.

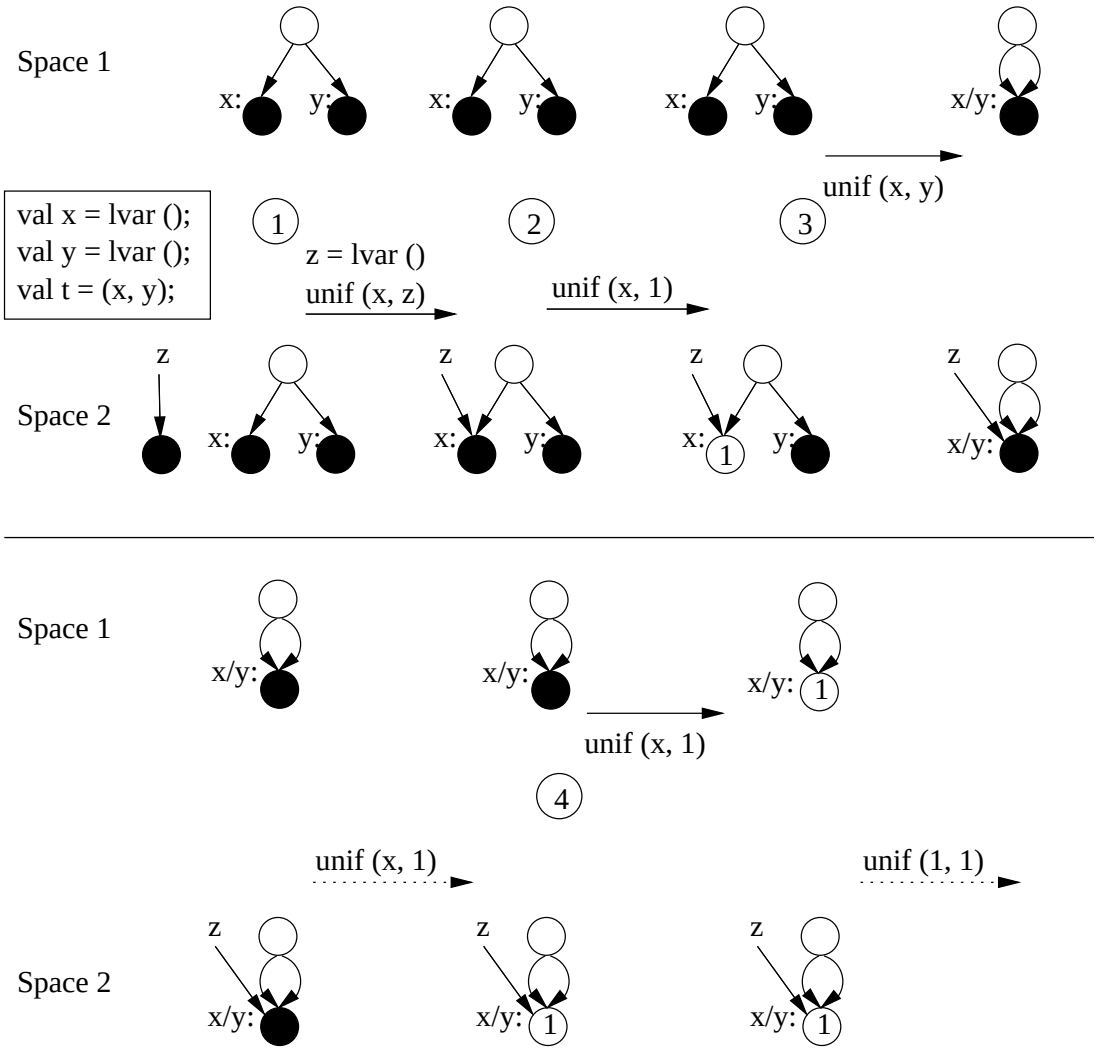


Figure 2.17: Entailment after propagation.

When such a failure exception reaches the default exception handler of a thread, the space is marked as failed. A space marked as failed is disentailed. All subordinated spaces of a failed space are marked as canceled. The threads in failed and canceled spaces are not further executed.

In L we use records with the single feature **`name`** `Failure` as indication for failure executions. The field value of this exceptions can contain an arbitrary value which can be used for debugging purposes.

2.10.4 Merging

A space can be merged into its parent space. The purpose of merging is to make the computation of a subordinated space available in its parent.

Merging involves the following operations

- New nodes and links are copied from the merged space to its parent. Local nodes of the merged space become local nodes of their parent.
- The node of the root variable is typically made available (see conditional below).
- All threads of the merged space are moved to its parent.
- All subordinated spaces of the merged space are merged to its parent.
- Speculative bindings in the merged space are turned into unification operations in the parent space.

For the deep guard conditional only the first two operations are relevant. When a space is entailed it has no threads, no subordinated spaces, and no speculative bindings.

2.10.5 Deep guard conditionals

The deep guard conditional `cond` is an operator which takes three functions `(guardF, thenF, elseF)` as arguments. The evaluation of `cond` happens in two steps.

In step one a new space is created as defined above. The new space has an initial thread containing the application of the function `guardF` to the root variable. The second step of the evaluation happens when the space is entailed or disentailed.

If the space is entailed it is merged with its parent and the `cond` operator evaluates to the application of the `thenF` function to the root variable.

If the space is disentailed the `cond` operator evaluates to the application of the `elseF` function to the singleton value.

2.10.6 Other situated nodes

Cells in spaces Cells are situated nodes. When the content of a local cell is changed this change is propagated to all subordinated spaces.

The content of a global cell can be accessed, but it cannot be modified. This is the reason why L has two built-in operators for cells.

The `exchange` operator applied to a global cell raises an exception. An alternative design decision would be to suspend the exchange operation on global cells. In L we have choose exception, because it is easy to implement. Suspending the thread does not seem really useful and would add an unnecessary complexity to the implementation.

Futures in spaces Global futures loose the read-only protection and are treated as logic variables. Speculative binding of global futures in spaces is allowed.

Only when a space is merged the speculative binding is redone in the parent space using unification, which will suspend if the future is local. Note, that in L this situation does not occur, because only entailed spaces are merged.

Treating global futures in the same way as local futures, i.e. every binding attempt suspends has an unwanted effect. The problem which occurs is the following: assume an expression `unif (x, 1); if x = 1 then ...` is executed in a space where x is a global variable. Later an expression `unif (x, f)`, where f is a future is executed in the home space of x. In this situation the speculative binding of x must be retracted and a thread unifying f and 1 must be executed in the subordinated space. This thread will of course correctly suspend, but the decision based on speculative binding cannot be retracted.

A speculative binding forces the lazy computation of futures introduced with the `byNeed` operator.

Feature constraints in spaces Feature constraints are represented as variables with attributes. These attributes play a similar role as variable bindings wrt. to spaces.

Variables with attributes preserve the invariant that attributes in subordinated spaces inherit all attributes from their parent. Global variables may have additional attributes not available in parent spaces.

Propagation of attributes is similar to the propagation of bindings. If an attribute conflict occurs during propagation the attribute is replaced by the new one. The old attributes are restated with the `widthC` resp. `featureC` operator as in the unification case.

A space is not entailed if the attributes of a global variable are stronger than the attributes of the variable in the parent space.

2.10.7 Discussion

Stability, cloning and injecting With the deep guard conditional it is possible to synchronize on entailment or disentailment of a space. To express constraint programming and flexible search engines spaces must support stability, cloning, and injection.

Stability is the property that a space is neither entailed, nor disentailed, but it has no threads which can execute and no change in the store of a parent space can ever change this situation.

Injection allows to add a thread to a subordinate space. The injected thread executes a user-defined function applied to the root variable. With injection it is possible to add for example new constraints into a space.

Cloning of a space creates an independent copy. In a clone global nodes are still linked to the corresponding nodes in the parent, but all local nodes are fresh. For example a clone of a local cell is a new cell independent of it original. The clone of a global cell on the other side is connected to the corresponding cell in the parent space.

In this work we will not explain how stability, cloning, and injecting is implemented. These concepts are discussed further in [90, 91, 89].

Pattern matching Pattern matching can be explained as an instance of the conditional. The case expression

```
case y of {c1=x1,...,cn=xn} => e
```

can be expressed with the deep guard conditional as

```
cond (fn x => let val ... xi = lvar () ... in
                 unif(x, (x1, ..., xn)),
                 unif(y, {c1 = x1, ..., cn = xn})
              end,
      fn x => let val ... xi = lvar () ... in
                 unif(x, (x1, ..., xn));
                 e
              end,
      fn x => throw ...)
```

The case statement is well suited as a primitive of the core language. The case statement can be explained without introducing spaces. Its implementation is much simpler and much faster then with spaces.

Semantically it is convenient to define pattern matching with the deep guard conditionals to have a single semantic foundation instead of two slightly different models. Especially when using elaborated synchronization conditions for case statements the semantics with deep guards has advantages. The major disadvantage of this semantics is that a lot of effort has to but into the optimization of the simple case [87, 14, 78].

In [87] an extension of pattern matching is discussed which allows coreferences in patterns and rejects matches of records with coreferences early, e.g. the evaluation of the expression `val x = lvar (); case (x,x) of (a,b) => ...` would suspend in our language forever. This matching rule would be rejected immediately in the extension of the case statement discussed in [87].

2.11 Examples

To show the usefulness of the language L a few simple examples in different programming paradigms are shown.

2.11.1 Functional programming: Append

As a language based on Standard ML it is trivial in L to write functions like `app` for concatenating lists or `map` for applying a function to all elements of a list.

Note that these function do not require any explicit code to synchronize on transients. The execution of the pattern matching on the input arguments blocks automatically if an incomplete list is provided and resumes its execution if the list is further instatiated. Furthermore the `map` function does not block, if the list elements are transients.

```
(* Functional append *)
fun appF (nil, ys) = ys
  | appF (x::xr, ys) = x::appF (xr,ys);

(* Functional map *)
fun map (nil, f) = nil
  | map (x::xr, f) = f x :: map (xr,f);
```

A major extension of L are logic variables and futures. Beside their usage as powerful communication primitives they allow to write an efficient tail-recursive version of the list concatenation.

```
(* Tail-recursive append with futures*)
fun appFut (nil, ys, zs)   = unif (zs, ys)
  | appFut (x::xr, ys, zs) =
    let val zr = lvar () in
        unif (zs, x :: future (zr));
        appFut (xr, ys, zr)
    end;
```

This implementation is efficient because the tail-recursion does not need memory for creating and unwinding the recursion stack. This approach of creating recursive data structures top-down, can be also used in language with destructive operations. It is unclear if a compiler can automatically transform a function like `appF` into an equivalent function using destructive operations internally, which are not visible. E.g. the following transformation of `appFut` to `appD` is safe, because no intermediate undefined values are ever visible outside of the function:

```
fun appHelp (nil, ys, zs) =
    replaceTail (zs, ys)
  | appHelp (x::xr, ys, zs) =
    let val zr= x::Undefined
    in
        replaceTail (zs, zr);
        appHelp (xr, ys, zr)
    end;
fun appD (nil, ys) = ys
  | appD (x::xr, ys) =
    let val zs= x::Undefined in
        appHelp (xr,ys,zs);
        zs
    end;
```

An advantage of `appFut` as opposed to `appD` is that it can be used as an agent in a concurrent application which consumes a stream `xs` and produces a stream `zs` even in the case that `xs` is not fully determined and has an open end.

In this scenario `appFut` is furthermore safe, because the reader of the output stream cannot corrupt the open tail, because it is always a future, which cannot be bound.

2.11.2 Concurrent lazy programming: Hamming

The lazy generation of hamming numbers is a small example which shows how by-need futures support lazy functional programming.

```
(* Hamming numbers *)
```

```
(* A lazy stream merger *)
fun m (xs, ys) =
    byNeed (fn () =>
            case xs of x::xr =>
                case ys of y::yr =>
                    if x<y then x::m (xr, ys)
                    else
                        if x>y then y::m (xs, yr)
                        else x::m (xr, yr));

(* A lazy n times generator *)
fun t (xs, n) =
    byNeed (fn () =>
            case xs of x::xr =>
                n*x :: t (xr, n));

(* hs is a lazy stream of Hamming numbers *)
val hs = lvar ();
unif (hs, 1 :: m (m (t (hs, 2),
                     t (hs, 3)),
                  t (hs ,5)));

(* h is the 10000th hamming number:
 *   288325195312500000 *)
val h = nth (hs, 10000);
```

The example is also useful as a benchmark for threads in L, because for every request of a by-need future a new thread is spawned.

2.11.3 Feature constraints: Paths

As an example for feature constraints we define a function to impose path constraints on trees. A path constraint defines that a certain path exists in a tree and returns the node at the end of this path.

```
fun path (rs, p::pr) =
    let
        val rr = lvar ()
    in
        featureC (rs, p, rr);
        path (rr, pr)
    end
  | path (rs,[]) = rs;

(* example *)
val r = lvar ();
```

```
val p = path (r, [1,2,3,4]);
unif (p,5);
```

The path equality used in a deep guard conditional tests if the node at the end of two path starting at the same node are the same.

```
fun pathEq (n,p1,p2) =
    cond (fn m =>
          (unif (m, path (n,p1));
           unif (m, path (n,p2))),
          fn n => n,
          fn () => false);
```

The following examples shows how the path constraint and the path equality test can be used.

```
(* entailment of records *)
val z = lvar ();
val y = ((1, (z, z), 3), 1);
val v = pathEq (y, [1,2,1], [1,2,2]);
        (* returns z *)

(* entailment of open records *)
val y = lvar ();
val z = path (y,[1,2,1]);
unif (z, path (y,[1,2,2]));
val v = pathEq (y, [1,2,1], [1,2,2]);
        (* returns z *)

(* disentailment *)
val z = lvar ();
val y = ((1, (1, 2), 3), 1);
val v = pathEq (y, [1,2,1], [1,2,2]);
        (* returns false *)
```

Chapter 3

The virtual machine LVM

In this chapter we describe a virtual machine (LVM) for L.

3.1 Overview

The virtual machine is a refinement of the language model defined in the previous chapter.

- The graph model of the store is refined to make essential aspects of the representation explicit.
- The language of the LVM is defined as an imperative low-level machine language, which is well suited for an emulator based approach.
- The machine language allows to integrate stateless data structures, i.e. records and procedures, into the bytecodes of machine programs. An external format, called pickles, is defined to represent machine programs and stateless data structures.
- The control for the execution of machine programs is defined as a single threaded engine.
- The machine language supports procedures with multiple arguments. Functions are implemented with a new variable as output argument.
- A compact representation of multiple computation spaces is defined using the script technique for maintaining multiple bindings of variables in different spaces. As an alternate technique for this binding windows are discussed.

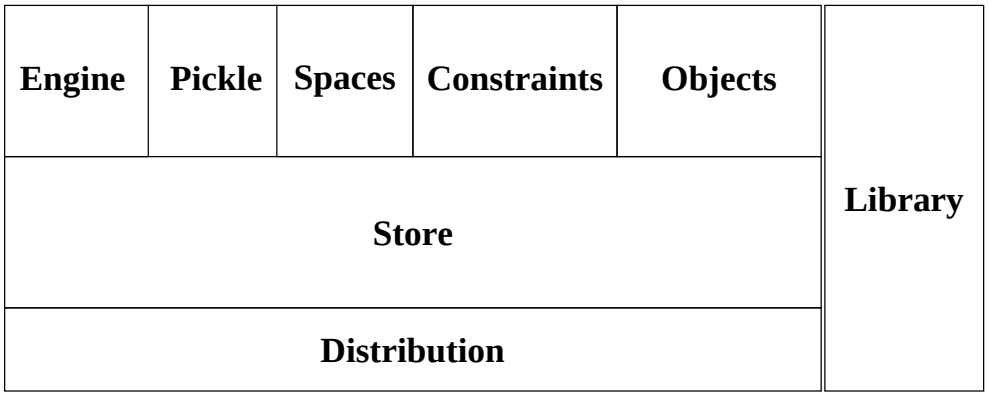


Figure 3.1: The modules of the LVM.

3.1.1 Modules of the LVM

The LVM is modularized as follows (see also Figure 3.1):

store The store of the LVM is a high level abstraction for storing dynamically typed values. It is at a high level compared to the linear storage model of standard hardware, but it provides a good intermediate model for explaining the design decision for representing data structures (see Section 3.3).

engine The engine is the sequential control for the execution of programs. The engine has machine registers and stacks, and executes an imperative machine language. This part of the LVM architecture maps very well to common hardware architectures (see Section 3.4).

pickling Executable programs are stored in an external format, called a pickle. A loader is responsible to transform a pickle into an internal representation, which consists of a graph in the store and of the program code as threaded code suited for emulation. Pickles can be created from the internal representation of a graph (see Section 3.2).

spaces For the maintenance of multiple computation hooks are supplied in the engine and store modules, e.g. when a thread terminates entailment must be checked and when a global variable is bound the space management must be involved (see Section 3.6).

constraints Other constraint systems are integrated into the LVM as extensions of logic variables with attributes to represent domain information. For the efficient implementation of constraint propagation a refinement of threads, called propagators, is used, which allows to implement specialized threads in C++.

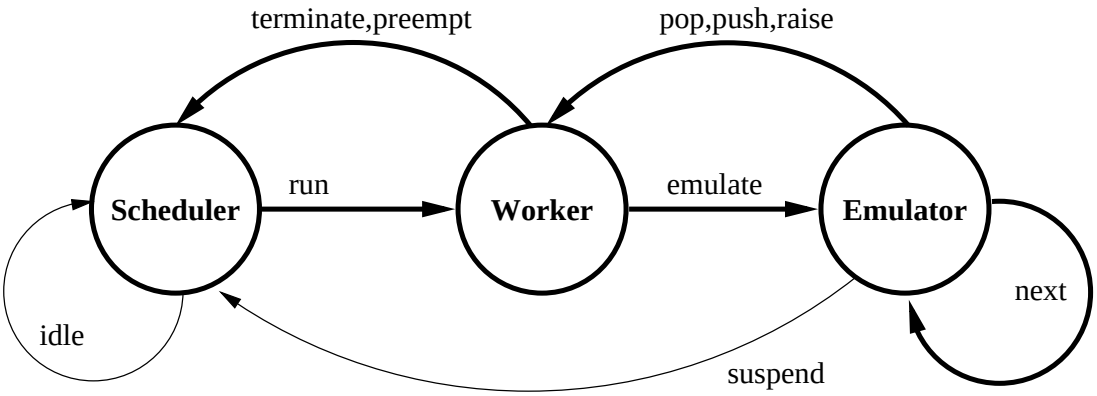


Figure 3.2: The engine of the LVM.

In this work we only consider the representation of open records. Other aspects of the constraint extensions of Oz are discussed in [70, 69, 118].

distribution The LVM supports the transparent distribution of the store among multiple sites. In this thesis we describe only the centralized system without distribution. Aspects of distribution in Oz is explained in [40, 107, 39].

objects The support for objects in the LVM is only partially touched in our work when we explain how to integrate new built-in abstract data-types. Other parts e.g. the support for efficient first-class messages, the maintenance of the self register, and the efficient access to attributes and object features is not part of our work. Objects in Oz are discussed in [42].

library Other parts of the LVM are common libraries and functions, e.g. for stacks, queues, characters, and strings, and an interface to the operating system, e.g. for I/O and memory management.

The description of the LVM in this thesis is an idealization of the concrete implementation *Mozart* [66]. The LVM is explained at such a level that the main design decisions and design alternatives are made explicit. The description is detailed enough to understand the Mozart implementation and it allows for the reconstruction of the Mozart VM.

3.1.2 The engine

The engine is the sequential control for the execution of concurrent threads. The main parts of the engine are the scheduler, the worker, and the emulator.

A high-level object model of the engine, where the scheduler, the worker, and the emulator are objects sending messages to each other, is shown in Figure 3.2. The objects and messages are explained in the following paragraphs.

The LVM is a single threaded operating system process. The light-weight threads of the language are implemented as user-level threads with a round robin scheduling policy. The *scheduler* is responsible for the fair and preemptive scheduling of concurrent threads. When bi runnable thread exists the scheduler runs in the *idle loop*, typically waiting for I/O. When one or more threads are runnable the scheduler selects one using a fair strategy and invokes the worker to *run* this thread.

The *worker* executes a single thread until it is finished or until the preemption condition is reached. In the first case the worker sents the *terminate* message to the scheduler and in the second the *preempt* message.

A thread contains tasks, which are executed sequentially following a stack discipline. A *task* is a closure containing the bytecode, a procedure environment, and a local environment. The environments are mappings from indices to nodes in the store. The *procedure environment* is allocated per procedure and is accessible through the G registers. The *local environment* is allocated per *procedure activation* and is accessible through the Y registers.

The worker executes the tasks and sents the *emulate* message to the emulator to execute the machine code of a task. The *emulator* interprets instruction per instruction of the bytecode indicated with the **next** message, until it reaches the end of the instruction sequence (*pop*), until a new task is created (*push*), or until an exception is raised (*raise*). In these cases control is passed back to the worker. Control is passed to the scheduler with the *suspend* message, when the execution of an instructions must block, e.g. when a determined node is expected, but a transient node is found.

The main parts comprising the *state* of the engine are shown in Figure 3.3 and an overview of their role is given in the following paragraphs.

Store The graph store, the *atom* and *arity table*, and the operations on the graph are discussed in Section 3.3. For the introduction of the LVM it is sufficient to understand that the graph has labelled nodes, with directed labelled links. The nodes in the store are referenced through machine registers and from the machine code.

Instructions and built-in procedures The operations performed by the engine are defined by the instruction set and by a number of predefined procedures, called built-ins. The instructions have the advantage that they are part of the worker with full access to the state of the LVM and with an efficient dispatch.

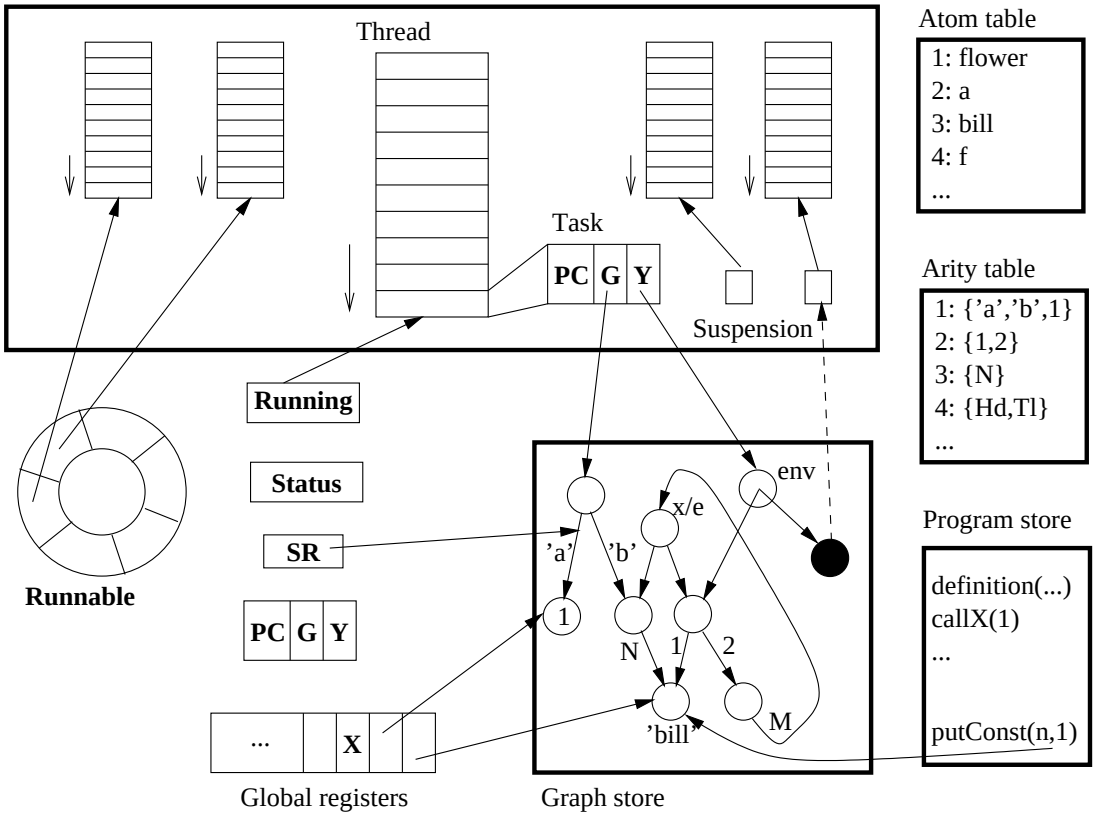


Figure 3.3: The state of the LVM.

Built-in procedures on the other side allow to factor out parts of the engine to make the emulator lean. The overhead for built-in procedures is a function call with the preparation of its arguments and the test of the return status.

X and SP A worker maintains the state for the execution of a single thread. The worker has a fixed number of *global registers* **X** to store temporary references to nodes and to pass arguments to procedures.

The register **SP** is the *structure pointer* which is used to read or write the fields of record incrementally (see Section 3.4).

Threads and tasks A thread has a stack of tasks. A *task* consists of a triple (PC, Y, G), where PC is the address of the next instruction in the program store, Y is a local environment with a number of registers, and G is a reference to the current procedure. The tasks of the LVM are similar to stack frame in imperative languages. The worker executes the tasks on the stack sequentially. A task is executed by emulating the instruction at the PC using the local environment and the environment defined by the current procedure. The local environment is implemented as a node in the store with a fixed number of modifiable fields.

Program store The *program store* contains machine programs. A *machine program* is a sequence of machine instructions. A *machine instruction* consists if a bytecode and arguments. Every instruction has an *address*. The *program counter*, which is stored in the PC register, contains the address of the currently executed instruction.

The internal representation of the program store uses threaded code for an efficient emulation. This internal representation is not relevant for this overview of the design of the LVM. In the following we use a readable assembler syntax for instructions, which is summarized in Figure 3.8 on page 62 and Figure 3.9 on page 63.

Implementation Implementing the model presented above directly in C++, where the scheduler, worker, and emulator are objects sending messages to each other, is not possible. It would create deeply nested recursion stacks, because the C++ standard does not require tail-call optimization and only very few C/C++ compiler implement it.

The implementation is therefore broken down into a single procedure with labels and gotos as outlined in Figure 3.5. The main registers of the engine are summarized in Figure 3.4.

type	register name	description
`Space*`	`space`	current space
`Thread*`	`running`	running thread
`ThreadQueue`	`runnable`	runnable threads
`ProgramCounter`	`PC`	program counter
`Tagged[]`	`X`	global registers
`Tagged*`	`Y`	local environment
`Procedure*`	`G`	procedure environment
`volatile unsigned union {`	`SR`	status register
`Tagged exception;`		raised exception
`Tagged suspendVarList;`		transients list
`}`	`retInfo`	return info

Figure 3.4: The registers of the engine.

The scheduler is implemented with the entry points `Schedule`, `Suspend`, `Terminate`, and `Preempt` (see Section 3.5). The worker is implemented with the entry points `Raise`, `Pop` and `Run`. The push method to create a new task is directly implemented in the corresponding instructions (see Section 3.4). The emulator uses the threaded code technique [11, 21, 54] as an efficient method to dispatch on the instruction[1].

3.2 The machine language

The machine language of the LVM is an imperative language with instructions and built-in procedures. A compiler translates the high-level language L into this machine language.

3.2.1 Pickles

A pickle is a closed representation of a graph spawned by a node in the store. Pickles contain stateless replicable nodes and code. Replicable nodes are nodes which have no state. Records and procedures are replicable and cells and transients are non-replicable. If a graph spawned by a node contains a non-replicable node it cannot be represented as a pickle.

[1]The GNU C++ compiler supports the nonstandard feature of computed labels, which is need for threaded code generation. The implementation provides a compilation switch to disable threaded code.

```
engine() {
  runnable = ... // initialize

Schedule:
  if (SR) handleEvents();
  while (runnable->empty()) idle();
  running = runnable->get();
  startTimer(TimeSlice);
  goto Run;
Suspend:
  running->saveX(X);
  goto Schedule;
Terminate:
  goto Schedule;
Preempt:
  running->saveX(X);
  runnable->add(running);
  goto Schedule;

Raise:
  running->raise(retInfo.exception);
  goto Run;
Pop:
  goto Run;
Run:
  if (statusReg) goto Preempt;
  if (running->empty()) goto Terminate;
  (PC,Y,G) = running->popTask();
  goto *PC;  // threaded code emulator

MOVE_X_X: ...
  PC+=3;
  goto *PC;
CALLX: ...
  running->push(...);
  goto Run;
RETURN:
  goto Run;
...}
```

Figure 3.5: The main procedure of the engine.

e	::=	$int(s)$	integer
	|	$atom(s)$	atom
	|	$name(s)$	global name
	|	$rec(n, e_1, e'_1, \ldots, e_n, e'_n)$	record
	|	$tup(n, e_1, \ldots, e_n)$	tuple
	|	$cons(e, e')$	list element
	|	$proc(s, [e_1, \ldots, e_n], lbl, ...)$	procedure
	|	$bi(s)$	built-in procedure
	|	$v : e$	labelled expression
	|	$ref(v)$	reference
v	::=	an identifier	label of a node
s	::=	a string	
lbl	::=	an identifier	code label

Figure 3.6: The pickle format.

Pickles allow to create *persistent* representation of nodes and code. The creation of such a representation is called *pickling* and the operation to internalize a pickle is called *loading*. Pickling takes a node and creates the pickle representation of the graph spawned by the node. The load operation reads the pickle description, creates an internal representation, and returns the node which was used for pickling.

A pickle consists of two major parts: the representation of the nodes and the representation of the bytecode. Figure 3.6 shows an overview of the representation of the nodes v. The bytecodes are summarized in Figure 3.8 and Figure 3.9.

Integers, atoms, and records The representation of a node starts with a tag, e.g. *int*, *atom*, followed by a number of arguments. Integers $int(s)$ and atoms $atom(s)$ are represented using a string representation for their numeric resp. symbolic value. Records are represented as $rec(n, e_1, e'_1, \ldots, e_n, e'_n)$ with their width n, their features $e_1, \ldots, e_n$ and the corresponding field values $e'_1, \ldots, e'_n$. Tuples $tup(n, e_1, \ldots, e_n)$ are represented as compact records without the features and list elements $cons(e, e')$ also without the features and the width.

Names For the representation of names as $name(s)$ the LVM generates a unique string s. This string s is build of several components: a unique identifier for the LVM process and a unique counter value which is choosen when a new name is created.

The unique identifier for a LVM process is created from the internet address of computer (ip address), the time when the LVM was started (timestamp), the

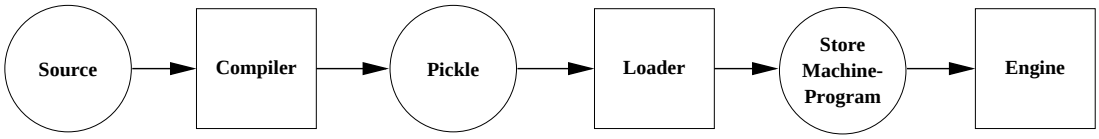


Figure 3.7: From Oz source to the LVM.

process id (pid), and a random number. Under the assumption that all hosts have a unique internet address this ip address, the timestamp, and the pid would already give a unique identification of an LVM process, but many hosts do not have a unique ip address therefor some form of randomness is added.

References Cycles in the graph are represented using labelled nodes $v : e$. A labelled node is referred by a reference $ref(v)$. For example $v : tup(2, ref(v), ref(v))$ is the representation of the tree generated by the expression

```
let val x = lvar () in unif (x, (x,x)); x end
```

.

Procedures The representation of a procedure $proc(s, [e_1, \ldots, e_n], lbl, ...)$ has as first argument a globally unique string as defined above for names. The following argument contains the nodes $e_1, \ldots, e_n$ stored in the G registers. The last argument lbl is the code label of the start of bytecode for the procedure body. A procedure has further arguments, e.g. a print name and other debugging information, which are irrelevant here.

Built-in procedures are represented as $bi(s)$, where s is a unique name of a built-in procedure, e.g. 'record' or 'newName'.

Compiling and loading Pickles are created by the Oz compiler. The Oz compiler translates an Oz source files using a given environment into a pickle (see Figure 3.7). The pickles created by the compiler are functors. A functor[2] is a data structure which consists of a specification of its dependency (imported modules), a procedure, and a specification of the resulting module. When the pickle is loaded into the LVM the import dependencies are resolved. Then the procedure of the functor is applied to the nodes obtained by this resolution. The application returns a module.

The *loader* converts the pickle format of the bytecode to the internal format executing the following steps:

[2]We do not explain the details of functors here (see [22] for more information).

- Create the graph representation.
- Internalize strings to atoms, static names, and integers.
- Internalize feature lists to arities.
- Convert the bytecode into threaded code [11, 21, 54].
- Initialize the inline caches of certain instructions.
- Internalize built-in names to built-in procedures.
- Internalize switch tables for the indexing instructions.
- Resolve optimized calls.
- Platform dependent byte order conversion.

3.2.2 Instructions

The instructions of the LVM are summarized in Figure 3.8 and Figure 3.9. The number of instruction is less than 150, which is an indication that the bytecode of the LVM is very compact. In this section we give only an overview of the existing instructions. In the following sections we introduce them step by step.

The instructions are structured into the following categories

Store operations The creation and access of symbolic data structures is an essential property of the LVM and it has a number of instructions to efficiently maintain them.

The LVM does some optimizations for numeric data by implementing some of the arithmetic operators as instructions, but we have not spent much effort to compete with other languages wrt. numeric calculations.

Control The LVM has extensive support for simple tests and pattern matching on records. Furthermore instructions for threads, exceptions, locks, and deep guards are available.

Procedures Procedures are at the heart of the LVM. Many instructions support the definition and application of procedures and the maintenance of the local environment.

Unification The LVM has a number of instructions to support the efficient compilation of unification. The major reason for optimized unification is that the LVM uses variables to pass output arguments.

Store operations (28)	
`moveXX`(i, j) `[/XY/YX/GX]`	register move
`moveMoveXYXY`(i, j, i', j') `[/YXYX/YXXY]`	multiple register move
`putRecordX`(ar, i) `[/Y]`	create record node
`putListX`(i) `[/Y]`	create list node
`putConstant`(vc, i)	load node in register
`setVariableX`(i) `[/Y]`	put new var in field
`setVoid`(n)	put n new vars in fields
`setValueX`(i) `[/Y]`	put value in field
`setConstant`(vc)	put constant in field
`select`$(i, vc, j, ckey, cind)$	field selection with caching
`createVariableX`(i) `[/Y]`	create new variable
`createVariableMove`(i, j)	... combined with move
`inlinePlus`(i, j, k)	addition
`inlinePlus1`(i, j)	add one
`inlineMinus`(i, j, k)	subtraction
`inlineMinus1`(i, j)	subtract one
`testLT`(i, j, l)	less than test
`testLE`(i, j, l)	less or equal test
Control (23)	
`matchX`(i, ht) `[/Y]`	indexing
`getVariableX`(i) `[/Y]`	get value from field
`getVarVarXX`(i, j) `[/XY/YX/YY]`	... double value
`getVoid`(n)	skip fields
`testConstantX`(i, vc, l) `[/Y]`	equality test
`testRecordX`(i, ar, l) `[/Y]`	test arity
`testListX`(i, l) `[/Y]`	test list element
`testBoolX`(i, l, l) `[/Y]`	test boolean
`testBI`(bi, loc, l)	built-in application and test
`try`(l)	install exception handler
`popEx`	deinstall exception handler
`lock`$(l,\ i)$	require lock
`cond`(l, l')	conditional
`branch`(l)	forward jump
Procedures (35)	
`definition`$(i, procBody)$	procedure definition
`definitionCopy`$(i, procBody, vcopy)$	... optimized
`endDefinition`(l)	marker
`callX`(i, n) `[/Y/G]`	first-class application
`tailCallX`(i, n) `[/Y/G]`	... tail-recursive
`directCall`(v, n)	first-order application
`directTailCall`(v, n)	... tail-recursive
`callBI`(v, loc)	built-in application
`return`	end of task
`allocateL`(i)	environment allocation
`allocateL1` `[/2/3/4/5/6/7/8/9/10]`	... with fixed size
`deallocateL`(i)	environment deallocation
`deallocateL1` `[/2/3/4/5/6/7/8/9/10]`	... with fixed size

Figure 3.8: Instructions (Part I)

Unification (17)	
`unifyXX`(i, j) `[/XY]`	unification
`getRecordX`(ar, i) `[/Y]`	... with record
`getListX`(i) `[/Y]`	... with list
`getListValVar`(i, j, k)	... combined
`getConstantX`(vc, i) `[/Y]`	unification with constant
`unifyVariableX`(i) `[/Y]`	read/write variable in field
`unifyVoid`(n)	read/write variables in fields
`unifyValueX`(i) `[/Y]`	read/write value in field
`unifyValVarX`(i, j) `[/Y]`	... combined
`unifyConstant`(vc)	read/write constant in field
Objects (14)	
`getSelf`(i)	read self register
`setSelf`(i)	write self register
`inlineAt`$(vc, i, ckey, cind)$	attribute access
`inlineAssign`$(vc, i, j, ckey, cind)$	attribute assignment
`sendMsgX`$(v, i, ar, ckey, cval)$ `[/Y]`	message sending
`tailSendMsgX`$(v, i, ar, ckey, cval)$ `[/Y]`	... tail-recursive
`applMethX`(ami, vc) `[/Y/G]`	method application
`tailApplMethX`(ami, vc) `[/Y/G]`	... tail-recursive
Debugging (9)	
`skip`	no operation
`raiseError`(v, v', v'', v''')	raise error exception
`debugEntry`$(\ldots)$	enter procedure
`debugExit`$(\ldots)$	exit procedure
`globalVarname`(vc)	print name of G register
`localVarname`(vc)	print name of Y register
`clearY`(i)	mark register unused
`profileProc`	start profiling
`endOfFile`	marker

Figure 3.9: Instructions (Part II)

i, j, k	register indices
n	positive number
v	a label of a node
vc	label of a constant node
l	code label
ar	record arity
pri	procedure info
dci	direct call info
ami	application method info
ht	hash table
c	cache

Figure 3.10: Instruction arguments.

Objects We will not explain the instructions which support objects. They are listed here just to give an impression how much support is given for objects in the LVM.

Debugging The compiler can generate extra code, which allows a debugger to relate the bytecode to the source code and to profile the code.

The identifiers used for arguments are summarized in Figure 3.10. We explain them when we introduce the instructions.

Direct nodes An unusal aspect of the Oz bytecode is the direct reference to nodes in the store from the bytecode. In the instruction tables the arguments containing such *direct nodes* are indicated with a v prefix.

Direct nodes in instructions provide for certain optimizations:

- Nodes can be accessed directly without an indirection through registers.
- Nodes need not to be stored in procedure environments.
- It becomes possible to use unboxed representation of some data structures. The optimized first-order application is for example transformed at run-time into an internal instruction using an unboxed representation of the procedure.
- Some data structures, e.g. strings, atoms, and names, can be created at load time and need no resources at run time.

Direct nodes are inserted by the compiler. The compiler can create these nodes at compile time, e.g. strings, atoms, and names. Direct nodes may be also taken

void*	CodeLabel
int32	Arg1
...	
int32	ArgN

Figure 3.11: Instruction format

from the compiler environment, e.g. references to already loaded procedures for first-order applications. When the compiler creates a pickle all nodes referred to from the bytecode are pickled too.

The possibilities opened by using direct nodes in the compiler-VM interface are not fully explored yet, but the current usage shows already that they are very useful.

Internal format The program store is represented as an array of 32-bit words. An instruction starts with a pointer to the native code implementing the instruction (threaded code). The following words are the arguments of the instruction and their number depends on the type of instruction (see Figure 3.11). The number of words needed for an instruction is called the size of the instruction.

In the internal format more instructions are supported than listed above. In the following we will explain these extensions to the bytecode when they are needed.

3.2.3 Addressing modes

The instructions of the virtual machine can use three different addressing modes for refering nodes in the graph store:

- The X addressing mode uses the global X registers, which are allocated per thread.
- The Y addressing mode uses the local environment, which is allocated per procedure invokation.
- The G addressing mode uses the procedure environment, which is allocated per procedure definition.

In the assembler notation the symbol R_i represents one of these modes plus an index. Register indices start with zero. For example the register G_5 refers to the sixth entry in the current procedure environment.

Supporting all addressing modes for all instructions makes the instruction set very regular, but a drawback is that too many opcodes are needed. Three opcodes are for example necessary for instructions with one register argument and nine opcodes are required for instructions with two register arguments.

The LVM instruction set is designed such that frequently used addressing modes are directly supported, e.g. the `call` instruction supports all three addressing mode. When an addressing mode is used infrequently at least the X addressing mode is supported, because it is always possible to load any register into an X register with additional moves.

3.2.4 Discussion

Threaded code Threaded code [11, 21, 54] is the state of the art method for a very efficient dispatch on the bytecodes of instructions. Threaded code requires that the implementation language supports computed jumps. In our case the C++ language does not support computed jumps, but the GNU C++ compiler has an extension which supports them.

A drawback of threaded code is that the emulator is one huge C++ procedure, which makes it hard for the C++ compiler to generate highly optimized code.

An alternative which was recently proposed by Magnusson, et al. [61] is based on the assumption that a C++ compiler does the tail call optimization and many machine registers are available. In this case every instruction can be implemented as a function which does a tail-call to the next instruction. The state of the emulator is passed in the arguments of these functions.

Stack machines Many virtual machines use an operand stack instead of global registers, e.g. the JVM [60]. A major advantage of a stack machine is that no register allocation is necessary in the compiler. For these machines advanced runtime optimizations resp. optimizations when translating the machine code to native code are necessary [23, 24].

Closure conversion The G addressing mode can be removed using a compilation technique called closure conversion [7]. The closure conversion adds additional arguments to every procedure through which the free variables are passed when the procedure is applied. A drawback of closure conversion is that it may be necessary to save the free variables from the additional arguments in the local environment. This is not necessary in our approach, because the free variables are stored in the global environment.

Closure conversion could also be applied to our language. It would reduce the number of instructions, but it would not give any speed up, because the G addressing mode does not incur an overhead in our emulator-based LVM.

3.3 A refined graph model

This part of the thesis describes a refined graph model for the *store* of the LVM. The store is a module of the LVM which is independent of the execution model. It provides hooks to support multiple computation spaces which are explained in Section 3.6.

The level of detail exposed in the refined graph model is such that the key design decisions and optimizations of the implementation can be discussed, e.g. optimized representation of variables in structures, usage of registers, storage consumption, and memory management.

The refinements of the graph model which are explained below can be summarized as follows

tagged nodes Units are represented as tagged nodes.

three-level tagging scheme A unit is either represented as a single tagged node, a tagged node with a heap node, or a tagged node with a generic node.

reference nodes Binding of variables is implemented with reference nodes.

efficient cycle check The cycle check in the unification algorithm is implemented with a destructive operation on the graph.

3.3.1 Node classification

Figure 3.12 shows a classification of nodes in the LVM. In the following paragraphs the properties of the different node types are defined.

The nodes in the LVM store can be classified into tagged nodes and heap nodes, which are defined below.

Tagged nodes are small nodes. Tagged nodes have a label, called the *tag*. The tag discriminates different kinds of units. Tagged nodes are small nodes, because they must fit into one machine word of the real machine. All data structures represented in the graph are referred to through a tagged node.

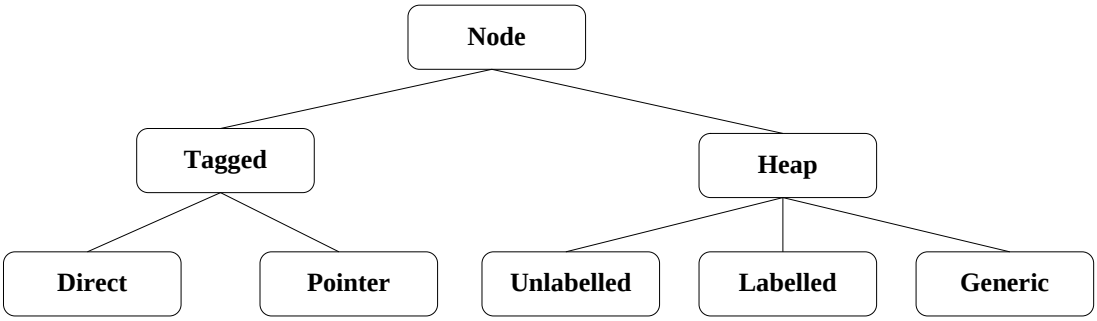


Figure 3.12: Classification of nodes.

Direct nodes are tagged nodes with an additional label. The tag and this label is sufficient to represent a unit directly.

Pointer nodes are tagged nodes with have a single link to a heap node. Pointer nodes store only the type information of a unit directly. Other parts of the representation are stored in the heap node.

Heap nodes are nodes of arbitrary size. Heap nodes are only referred to through pointer nodes. They represent those parts of a unit which does not fit in the tagged node.

Unlabelled heap nodes are heap nodes with do no have a secondary tag. The primary tag in the pointer node is sufficient to discriminate the type of the unit.

Labelled heap nodes are heap nodes with a *secondary tag*. The and the secondary tag together discriminate the type of the unit.

Generic heap nodes are heap nodes which hide the details of their representation. These nodes are only accessible through a number of interface functions.

A unit is either represented as a direct node or as a pointer node and a heap node (see Figure 3.13).

Figure 3.14 shows an overview of the tags in the LVM. The concept of tagged nodes is essential for the design, because:

1. Every tagged nodes needs the same amount of memory. This means a memory cell storing such a node can be used and maybe updated to store different nodes of this class. Especially for a dynamically typed language this property is needed, because nodes of arbitrary types can for example be passed as arguments and stored in fields.

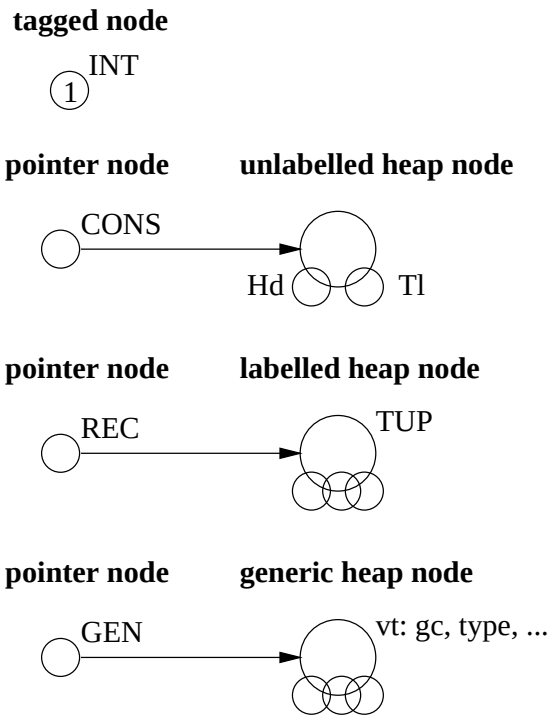


Figure 3.13: Examples of node representations.

Tag	Direct	pointer to	
REF		tagged	reference
WREF		tagged	write reference
VAR	space		optimized variable
FUT	space		optimized future
TRANS		labelled	gen. transient
CONS		unlabelled	list element
REC		labelled	record or tuple
LIT		labelled	atom or name
INT	int value		small integer
FLOAT		unlabelled	float value
EXT		labelled	labelled extension
GEN		generic	generic extension

Figure 3.14: Tagged nodes.

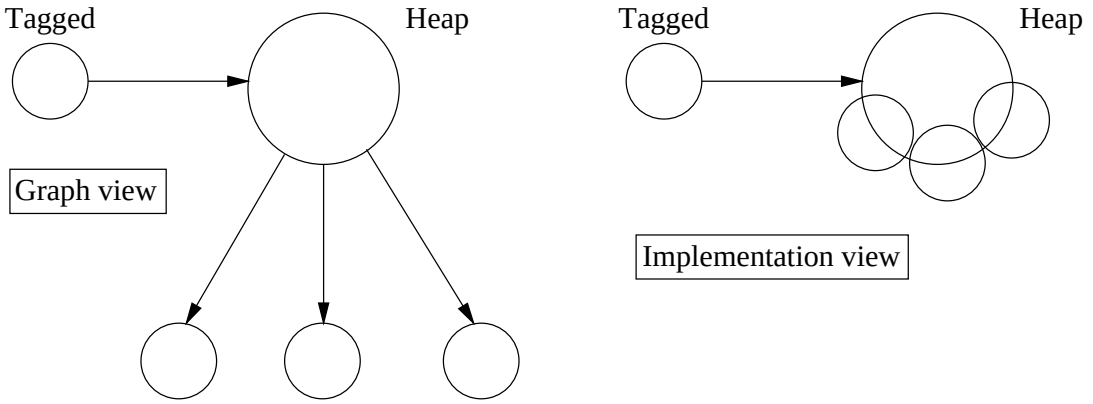


Figure 3.15: Fields are glued with their heap node.

2. The word size of tagged nodes is the natural size for operations of processors, e.g. load, store, and arithmetic instructions typically operate most efficiently on words.

Fields Heap nodes in the LVM have a regular structure. They can have multiple labels, e.g. a secondary tag or an arity, and a number of fields. The number of fields is called the field width. The fields are ordered and they are accessed by numbers $\{1, \ldots, n\}$.

A field has a field value, which is a tagged node. In the LVM all field values can be modified. When new heap nodes are created all field values are initialized to the *tagged zero*, which is a special tagged node, with tag zero and pointer field zero, used to indicate an exceptional value. The initialization of the heap nodes updates this tagged zeros to useful values.

An essential aspect of fields is that a heap node with n fields has enough storage to represent the n tagged nodes in the fields. When we draw a graph (see Figure 3.15) we use arrows between the heap node and its fields values, but these arrows are special because they do not need any memory. A picture which gives a better intuition is that of a heap node with directly glued tagged nodes.

Changes to the graph invariants A consequence of storing tagged nodes in fields and registers is that these nodes can be overwritten and thus destroyed. This is a major change with respect to the language graph, because in the refined graph one has to be very careful when creating links to tagged nodes, that this link is not broken unintentionally by overwriting the field resp. register.

To alleviate this problem no links to nodes in registers can be created and only links to nodes in fields which are not modified are created in the LVM.

Register nodes Register nodes are a subclass of tagged nodes which can be stored in registers of the LVM. The unique property of register nodes is that they can be replicated without changing the meaning of the unit they represent. Except for transients (TRANS, VAR, FUT) all tagged nodes of the LVM have this property.

This property is for example needed to make the register allocation independent of the store. The compiler can move and copy nodes between registers freely. Another example is the initialization of fields in new heap nodes. They can be initialized by copying register nodes into the fields values.

3.3.2 Records

The LVM supports different representations for records: as names and atoms, as list elements, as tuples, and as other records.

Literals Literals are names and atoms. They are represented as tagged pointers with the tag LIT. Their heap node has a secondary tag to distinguish atoms and names.

The heap node of an atom is labelled with the string of characters for the atom. A string is internalized into the LVM through an atom table which guarantee that every atom is represented with an unique node. The atom table maps a string uniquely to an atom node in the store.

The heap node of a name is labelled with a number and its home space. The number is used for generating a hash value for the efficient implementation of the arity (see below). A second reason for a number is that names must be ordered to simplify the creation of new arities. Names are situated in spaces and need therefore a home space (see Section 3.6).

Non-primitive records List elements are represented as tagged pointers (CONS) with an unlabelled heap node with fields for the head and tail of the list. List elements obtain special optimizations because they are the most frequently used kinds of records.

The representation of tuples and other records is not really different. Only the representation of the arity (see below) is optimized in the case of tuples. Records are represented as tagged pointers with the tag REC. The heap node has the arity as label and fields. The number of fields of the heap node is equal to the width of the record.

Records are always represented in a canonical form. This means that every operation producing a record needs to normalize it, if it is a list element or a

tuple. The reason for this is that the equivalence test in the unification algorithm becomes simple. Two records are only equal if at least the tags in the tagged nodes are the same and also the arities in the case of non-list records.

Arities A record arity is a partial function from the set of features to a integer. The features $\{c_1, \ldots, c_n\}$ are mapped to the numbers $\{1, \ldots, n\}$.

The arity has the additional functionality to efficiently implement the member function to test if a feature is in the domain of the arity function. The arity function is therefore extended to a total function mapping the features not in the domain to the index 0.

Arities are uniquely represented in the LVM. For every set of features a unique entry in the arity table is used. The costs for creating resp. finding a unique arity have to be paid when new records are created. In many cases the arity can be created at compile resp. load time. Only when arities are created dynamically the costs for creating a unique arity must be paid at run-time.

Unique arities allow to test the equality of two arities very efficiently. This is for example necessary for inline-caching of field selections and for the efficient unification and matching of records.

For the efficient compilation of record construction and record match (see Section 3.4) a global order on all features must exist. This order must be consistent with the mapping of the arities: if $f < f'$ wrt. to the global order then in every arity containing f and f' the mapping of f must be less than the mapping of f'.

3.3.3 Transients

An essential change in the refined graph model is the representation of transients. In the language graph the binding of transients was explained as superimposition of a new node on the transient. It is practically not possible to implement this operation directly, because all links to the transient cannot be redirected to its binding.

References Transients in the LVM use a variation of the representation introduced in the WAM for logic variables. A transient is only accessible through an indirection, called a reference. A *reference* is tagged pointer with the tag REF where the pointer refers to another tagged node.

Transients are represented as tagged pointers with tag TRANS and a labelled heap node, which contains a secondary tag for the different kinds of transients, the home space, the suspensions, and possibly attributes.

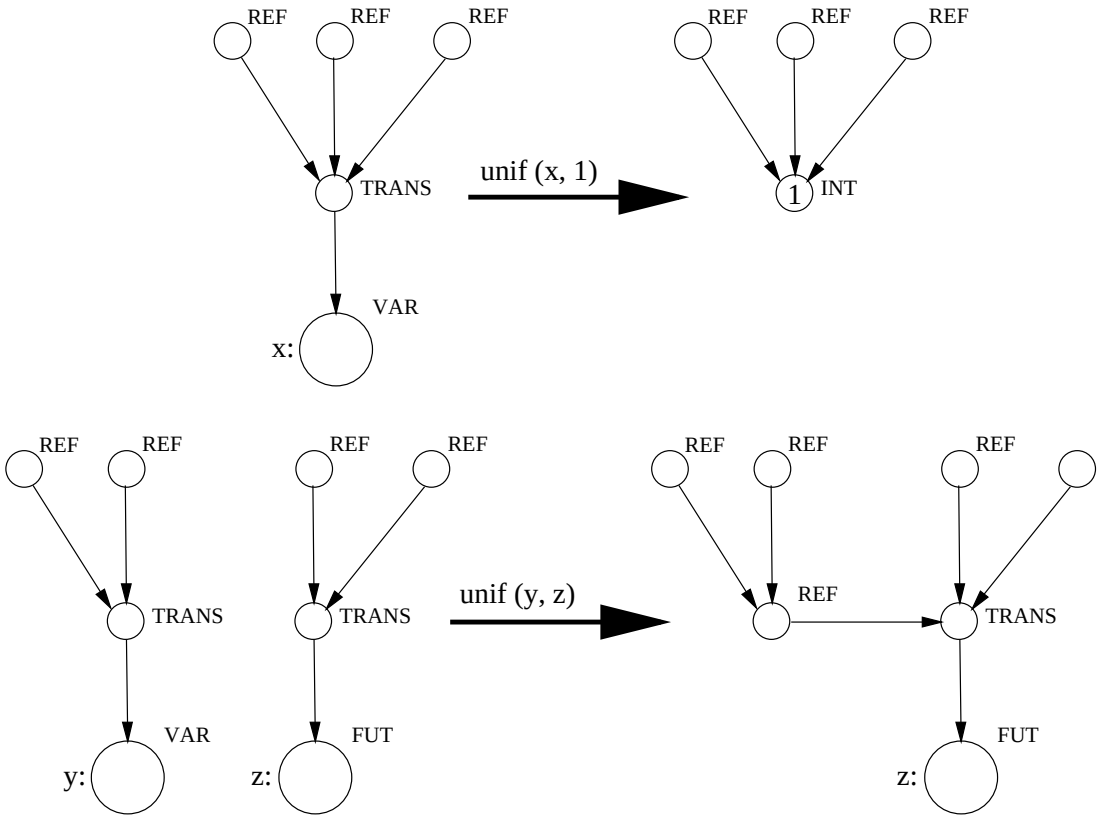


Figure 3.16: Binding transients with multiple references.

Binding A transient is bound by overwriting its tagged node with a new tagged node. Figure 3.16 shows a variable `x` with multiple references which is bound to the number 1 and a unification of a variable `y` with a future `z`.

Dereferencing The reference nodes are not changed when a variable is bound and remain in the graph. When binding a transient to another transient a chain of references is created. A reference node can therefor refer to a transient node, another reference node, or a determined node.

The LVM handles these cases by transparently dereferencing tagged nodes, before using them. The dereference operation follows a chain of reference nodes until the end. The dereference operation is performed whenever the type of a node is needed.

Van Roy [104, 105, 106] uses an alternative design for dereferencing for high-performance Prolog implementations. In this approach references are not dereferenced transparently, but an explicit operation to dereference a node is used. This scheme is especially useful if the compiler finds out, e.g. with global analysis techniques, where no references ever occur.

In most cases the dereference operation is needless, because only very few reference nodes exist in typical programs. The LVM can circumvent the problem of useless dereference operation, because it is dynamically typed. Whenever a node of a certain type is expected, e.g. an integer in an arithmetic operation, a type test has to be performed anyway to ensure that the node is of the expected type. In the LVM the test for the expected type is done before the dereference operation. Only if the node is not of the expected type a dereference operation is performed and the type test is repeated.

The following program fragment shows the example of an operation to add one to a node, which is expected to be an integer.

```
Tagged plus1(Tagged a) {
  if (!isInteger(a)) {
    a=deref(a);
    if (!isInteger(a)) error;
  }
  // perform operation on integer node
  ...
}
```

Safe dereferencing As already pointed out transient nodes are no register nodes and they cannot be duplicated. A problem which occurred frequently during the implementation was the replication of transients after using dereferencing. One has to be very careful that the node obtained by the dereference operator is only stored in registers if it is no transient.

To circumvent this kind of bugs an alternative to the dereferencing until the end of a reference chain is the safe dereferencing which guarantees that only register nodes are returned. A reference node is only returned if it is the last reference in a chain which points to a transient.

Shorten reference chains The virtual machine guarantees that no cyclic reference chain can be created, but reference chains can be arbitrary long. Possible means to shorten reference chains[3] are:

- A heuristics which binds newer to older transients is useful for the functional programming style, where two types of variables occur frequently: short lived temporary variables which are bound quickly after their creation and long lived variables which are for example bound at the end of a recursion.

[3]With spaces using the scripting technique the shortening of chains needs special care, because it must be possible to undo bindings of transients.

- When the garbage collector traverses the graph store it shortens the reference chains, such that only references to transient nodes remain.
- Nodes can be dereferenced before they are stored in a field. Under the assumption that references are rare and most nodes are accessible without a references the overhead for this technique is to high for a little gain and is not used in the LVM.
- Similar is the technique to shorten reference chains when accessing a field, which is also not performed in the LVM.

Transients in fields Transient nodes are not stored in the registers of the LVM directly. They can be stored only on the heap and have to be referenced indirectly with reference nodes in registers.

It is however possible to store transients directly in fields. This is useful to save memory. Especially with the optimized representation explained below some variables need no memory at all. Transients in fields are called *direct transients*.

When a transient in a field is accessed, e.g. to store it in another field or a machine register, a complication occurs, because transients cannot be replicated. The access to such a field needs to create a reference to this field which can then be stored in registers and other fields.

To avoid that every field access introduces a sometimes superfluous reference node a test is performed for every field access if the field contains a direct transient or not. Allocating transients in fields requires special care in the copying garbage collector to ensure that direct transients are not copied out from their fields (see Chapter 4).

In the WAM representation of variables no such problem occurs because variables are represented as self references and an access resp. copy of such a self referencing pointer automatically turns it into a reference to the variable.

Transients cannot be stored directly in fields of cells, because these are overwritten and potentially created references to this transient will refer to a wrong value after an exchange.

Optimized variables The LVM supports an optimized representation of variables, with a single tagged pointer node with tag VAR. The pointer field of this node refers to the home space of the variable (see Section 3.6).

The optimized variable is a variable with no suspensions and no attributes. Whenever a suspension or attributes are added to this variables its representation is transformed into the unoptimized transient representation.

The major reason for the introduction of optimized variables is that the LVM uses procedures with variables as call-by-reference parameters for returning output and has no support for functions with a return value. Variables are thus created frequently which are only introduced for the output argument and their optimization has a real influence on the performance of almost every program.

The second effect of optimized variables is that they can be directly stored in fields of records without requiring additional memory. In connection with the call-by-reference ability this means that structures can be efficiently constructed top-down with tail-recursive procedures.

In the following example of the append procedure `app` to concatenate two lists the output list `zs` is constructed top-down. The temporary variable `zr` needs no memory, because it can be directly allocated in the tail field of the list `x::xr`. The recursive application of `app` then gets a reference node to the tail field as third argument.

```
fun app (nil, ys, zs)   = unif (zs, ys)
  | app (x::xr, ys, zs) =
    let val zr = lvar () in
        unif (zs, x :: zr);
        app (xr, ys, zr)
    end;
```

Optimized futures It is often useful to use futures instead of variables in structures which are visible to concurrent threads to protect them. For example in a consumer-producer application where the communication channel is implemented as a stream it is usually desirable that only the consumer is able to write to the stream. In this case the consumer would create a stream where the tail is a future. The corresponding variable would be only visible to the consumer. With the implementation of futures described above memory for a variable and a future would be needed besides the memory for the stream.

The procedure `appFut` shows an append procedure with futures. The tail of the list is the future of `zr` to avoid that a concurrent reader can write on the output stream.

```
fun appFut (nil, ys, zs)   = unif (zs, ys)
  | appFut (x::xr, ys, zs) =
    let val zr = lvar () in
        unif (zs, x ::  future zr);
        appFut (xr, ys, zr)
    end;
```

A variable with a future can be represented similarly to the optimized variables described before. An optimized future is a tagged node with tag FUT and a

pointer to the space. Similar to the optimized variable it is turned into the transient representation when a thread suspends on it.

To represent the variable of this future we introduce a second kind of references, namely *write references*, with the tag WREF. The variable of a future is then represented as a write reference to the optimized future. When this variable should be bound the dereference operation discovers that the reference is a write reference to a future and the binding operations replaces the future with the new binding.

A variable can be assigned only when the chain of references to the future contains only write references. When a usual reference is found in the chain this means that the variable represented with the write reference was already bound. Only in the case that a transient must be bound the dereferencing operation has to be extended to test that only write references are found.

When a field with an optimized future is accessed a usual reference is created. When a field with an optimized variable is accessed a read-write reference is generated.

3.3.4 Unification

The basic idea of a practical implementation of the unification algorithm is to implement the equivalence classes by binding one structure to the other and creating a reference similar to binding variables.

This algorithm has quadratic complexity, because the reference chains can grow to the size of the tree, but for practical programs this does not occur and the overhead for this implementation is much smaller compared to overhead for maintaining the equivalence set.

This implementation of unification creates sharing of common structures. In some cases this is a desired feature to reduce the memory consumption and it also is a kind of memorization. To avoid problems with spaces the sharing must be retractable. Therefore the unification algorithm trails every structure binding and undoes all binding when the unification terminates (successfully or not).

The destructive unification is only possible because the LVM has a single worker and unification is a non-interruptible atomic operation.

For an optimized implementation of unification it is essential to try the frequently used cases first. Because the LVM implements output arguments of functions as call-by-references parameters, it occurs very frequently that a variable is created before a function application with is bound to a value inside the function. The unify instruction therefore first tests for this very common case.

3.3.5 Discussion

Three-layered representation scheme The LVM supports many built-in data types, e.g. small integers, big integers, atoms, names, records, logic variables, futures, cells, and procedures, and it is extensible to support even more types. This is possible because it uses a scheme with three layers: tagged nodes, tagged extensions, and generic extension.

The bottom layer are tagged nodes. Tagged nodes allow to implement frequently used data types like small integers, lists, literals, variables and futures, efficiently.

Tagged extensions are not as efficient as tagged nodes, but there overhead is very small compared to the cost of operations on the data they represent, e.g. arithmetic on big numbers (see Chapter 4).

Generic extensions allow through a small set of interface function the integration of arbitrary new data types. This interface is very convenient to experiment with no types and to add data types where unbox, box, and type tests are not performance critical (see Chapter 4).

The same layered approach is also used for transients, with optimized representations as tagged nodes for variables and futures, and a generic representation as transient heap nodes. In Chapter 4 we show the virtual function interface for transients which allows to integrate other types of transient values.

Other transient representations The representation of variables in the most popular machine for Prolog, the WAM [110, 111, 1], inspired much of the representation of transients in the LVM. The representation of variables as self-references from the WAM which is extremely useful for making the allocation of variables in fields and their access efficient cannot be used in the LVM, because we support multiple computation spaces and a variable needs to represent its home space.

The WAM allocates variables also in the registers of the environments. These unsafe variables have to be treated carefully such that they are moved to the heap, if they extend the lifetime of their activation record. In the LVM variables are never allocated in registers, but it should be possible to integrate this technique into the LVM. It is questionable what the gain of this optimization could be under our assumption of an infrequent use of logic variables.

Return value placement Van Roy [105] proposes an optimized representation of uninitialized variable for high-performance Prolog implementations. In the LVM we do not use this technique because the number of variables used for output arguments of functions which are not allocated in fields is very small. It is furthermore unclear how to integrate spaces and uninitialized variables.

In logic programming and in the LVM return values are passed in memory using logic variables as call-by value parameters. Functional languages typically use machine registers to place return values. Both approaches have advantages and disadvantages, e.g. the logic approach works very well for the tail-recursive top-down construction of structures and the functional approach works very well for numeric problems.

Bigot and Debray [13] discuss how to combine the placement of return values in logic programming and functional programming and how to provide compiler support for an optimal placement policy.

Scheidhauer [87] analyses the difference between the two placement policies for Oz.

Taylor's scheme Taylor [101] proposed a scheme to represent variables such that no references remain after a variable is bound. This scheme was analyzed in [59] and the authors came to the conclusion that for Prolog the gain is doubtful.

Taylors scheme is not compatible with the idea of tagged register nodes in the LVM, because their essential property is that they are replicable and keeping track of all valid replicas incurs to much overhead.

In the functional programming style references occur very infrequently and as explained above the possibility of reference chains does not have an effect on the efficiency of programs which do not have references.

3.4 Sequential execution

In this section we explain how a single thread is executed by the worker. We explain the instructions to create and access nodes, procedure definitions, procedure applications, and pre-defined built-in procedures.

3.4.1 Worker

The worker executes the tasks of a thread in sequential order. The tasks on the thread are of different types, namely continuations, save tasks, and handler tasks (see Figure 3.17).

A *continuation* task (PC, G, Y) is a closure of a machine program starting at the code address PC. G and Y are the environment for the execution of the instructions. G is the reference to the procedure node in the store and Y is a reference to the local environment.

task type	task content
continuation	(PC, G, Y)
save task	$save(X_1, \ldots, X_n)$
handler task	$ex(PC, G, Y)$

Figure 3.17: Tasks.

The worker executes continuations by loading them into the corresponding *task registers* PC, G, and Y. A continuation is then executed by an emulator in the a fetch-decode-execute cycle. Instructions are fetched from the program store at the address PC and executed using the G, Y and X registers to address nodes in the store.

In the literature a continuation task is sometimes called procedure invocation or activation record of procedures.

Saving X registers The worker maintains a single set of global registers X, but it provides the illusion that every thread has its private set of X registers. The illusion is preserved by saving all valid X registers when a thread is preempted or suspended and restoring them when the thread is restarted.

A *save task* contains all currently valid nodes in the X registers. When the worker restarts the execution of this thread the first task to execute is the save task, which restores the values of the X registers.

The valid X registers are only approximated when a save task is created. The LVM saves all X registers from zero to $maxX$, where $maxX$ is the maximal number of X registers used in a procedure. This number is calculated by the compiler and stored in the procedure definition instruction. During garbage collection the exact number of used X registers is calculated using a liveness analysis algorithm (see Chapter 4).

Exception handler task A *handler task* is created for exception handling. They are never executed directly, but they are used as a marker on the stack of a thread, when an exception is raised (see Section 3.4.7 below).

3.4.2 Store operations

In this section we give a brief overview of the instructions for creating and accessing nodes. An example for creating a record node is the function f as follows

```
fun f z =
let val x=lvar ();
```

```
    val y={'a'=1, 'b'=x, 'c'=z}
in
    ...
end
```

It compiles into the following snippet of a pickle

```
v0: proc(s,[],lbl,...)
v1: int(1)
...
% X[0] contains z
% X[1] contains x
% X[2] contains y
lbl:
createVariableX(1)
putRecordX(ar('a','b','c'),2)
setConstant(v1)
setValueX(1)
setValueX(0)
...
```

The `createVariableX(1)` instruction adds a variable node to the store and puts a reference to it into the register X_1. The `putRecordX` instruction adds a record node with arity $\{a, b, c\}$ to the store and stores it into register X_2.

The fields of the record are not yet initialized. 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.

The instruction `setConstant(v1)` writes the node represented at the pickle label `v1` (the integer one) into the first field and increments the structure pointer (SP). `setValueX(1)` resp. `setValueX(0)` write x stored in X_1 resp. z stored in X_0 into the remaining fields of the record.

The structure pointer (SP) is a generalization of the technique known from the WAM to access the fields of tuples. In the LVM it allows to access the fields of records. The insight here is that if the arity of a record is known at compile time then the compiler can already compute the mapping of features to indices. This mapping defines the order of the `set` instructions such that the fields can be consecutively written.

Similar to the WAM the unification of records is optimized using `get` and `unify` instructions. For example the function

```
fn x =>
let val y=lvar()
in
  unif(x, {'a':y 'b':y})
end
```

is compiled into the bytecode

```
% X[0] contains x
% X[1] contains y
getRecordX(ar('a','b'),0)
unifyVariableX(1)
unifyValueX(1)
```

3.4.3 Control

In this section we briefly explain the basic ideas for compiling pattern matching. In detail the control aspects of the core language are discussed in [87].

A case statement is compiled into a `match`(i, ht) instruction, which contains a hash table ht which maps primitive values and record arities to code labels. We use the notation $ht(c_1 : l_1, \ldots, c_n : l_n, \ldots, ar_1 : l'_1, \ldots, ar_m : l'_m, else : l_e)$ for a hashtable which maps the constants c_i to the labels l_i and the arities ar_i to the labels l'_i. The match instruction has the else label l_e, which is used if no other match is found in the hash table. The instruction suspends if the register X_i is a transient value[4].

The following case expression

```
case x of {'a'=x1,'b'=x2} => unif(o,x1+x2)
        | 1 => 2
        | x => 3
```

is compiled to

```
v2: int(2)
v3: int(3)
...
% X[0] contains x
% X[1] contains o
matchX(0,ht(1:l1,ar('a','b'):l2,else:l3))
l1:
  getConstant(v2,1)
  return
l2:
  getVarVar(0,2)
  inlinePlus(0,2,0)
  unifyXX(0,1)
  return
l3:
  getConstant(v3,1)
  return
```

[4] In mozart the match instruction is extended to support early failure for attributed variables.

To efficiently decompose records the `match` instructions initializes the structure pointer (SP) such that `getVariableX` instructions can be used to read the field values of records. The instruction `getVarVar`(i, j) is a combination of two `getVariableX` instructions and reads the next two fields into the registers X_i and X_j.

As optimization of the `match` instruction with a single case the **`test`** instructions are provided, e.g. `testConstantX`(i, vc, l) is equivalent to the instruction `matchX`$(i, ht(vc : l_1, else : l))$, where l_1 is a label added to the directly following instruction.

3.4.4 Procedures

Functions of the language L are represented as procedures in the LVM. Functions are converted to procedures by adding an implicit argument, which is used as call-by-reference argument for the result value. This means every function **`fn`** `x => e` is transformed into a procedure with two arguments. In the our syntax the resulting function would be **`fn`** `(x, y) => unif (y, e)`.

In the LVM procedures with many arguments are allowed. The technique how single argument functions can take advantage of the multiple argument calling convention of the LVM is not discussed in detail here. Briefly every procedure and every procedure application knows the expected resp. supplied arity and during the application the proper conversions are done. When a procedure which expects a single argument is called with multiple arguments these are packaged into a single record. When a procedure which expects multiple arguments is called with a single argument this is unpacked during the application.

Procedures are first class values and they are dynamically created. First class value means that procedures are nodes in the graph store, which can for example be passed as parameters to procedures and stored in other structures.

Dynamic creation means that procedures not only have a static part, the code, but also a dynamic part, the procedure environment. The procedure environment encapsulates the values of the free variables of a procedure at the moment of the procedure definition.

To store temporary values during a procedure invocation a local environment can be allocated (see below).

Procedure definition Procedures are created dynamically with the instruction `definition`$(i, procBody)$. The procedure body $procBody$ contains the static information about the procedure. We use the notation $pb(code : lbl, arity : n, g : pe(r_0 : i_0, \ldots, r_m : i_m), maxX : k, \ldots)$ for the procedure body. The fields of the procedure body are

- A code label *lbl* for the start of the bytecode of the procedure.
- The arity n of the procedure which defines the number of arguments.
- The procedure environment $pe(r_0 : i_0, \ldots, r_m : i_m)$, where $r_l \in \{'x', 'y', 'g'\}$ and i_l is an index. $r_l : i_l$ means that the lth entry of the procedure environment is in register R_{i_l}, where R is X resp. Y resp. Z) if r_l is 'x' resp. 'y' resp. 'z'. The node in R_{i_l} can be addressed with the G-addressing mode as G_l in body of the procedure.
- The maximal number ($maxX$) of X registers used in the procedure. This number is used for saving the X registers for context switches.
- Further static information, e.g. debug information like the procedure name, the file, and line number.

The instruction `definition`(i, pb) where pb is $pb(code : lbl, arity : n, g : pe, maxX : k, \ldots)$ and pe is $pe(r_1 : i_1, \ldots, r_m : i_m)$ creates a new procedure node in the graph store with m fields, which are initialized with the nodes stored in $R_{i_1}, \ldots, R_{i_m}$. The procedure node is labelled with the procedure body pb. A reference to the procedure node is written into the register X_i.

As an example we show the compilation of the function `f` with argument `x` and a free occurrence of `c`.

```
val c=1;
fun f x = x+c;
```

It is compiled to the pickle

```
v1: int(1)
...
% X[0] = c
% X[1] = f
putConstant(v1,0)
definition(1,pb(code:lbl, arity:2,
                g:pe(x:0), maxX=2,...))
...
lbl:
  moveGX(0,2)
  inlinePlus(0,2,0)
  unifyXX(0,1)
  return
```

Procedure application The procedure application `callX`(i, n) waits until X_i is a determined node. If X_i is no procedure or the number of actual arguments n does not match the expected number of formal arguments an exception is raised.

If R_i is a procedure node with label $pb(code : lbl, arity : n, g : pe, maxX : k, \ldots)$ then a continuation $(lbl, -, R_i)$ is created. The local environment in this continuation is initially empty.

The worker saves the current continuation from the task registers on the task stack and starts with the execution of this new continuation.

Return The code of a procedure is terminated with the `return` instruction. The execution of this instruction informs the worker to execute the next task from the thread.

Tail-call Tail-call optimization is essential in languages without loop constructs. The compiler inserts the instruction `tailCallX`(i, n) for a sequence `callX`(i, n); `return` of an application and a return instruction. When the worker executes `tailCallX`(i, n) it creates a new continuation task as for the `callX`(i, n) instruction, but does not save the current continuation from the task registers onto the stack.

For tail-calls the task stack does not grow and therefore arbitrary deep recursions are possible. Tail-call optimization is trivial in LVM, because there are no inter-task references. In other words communication between tasks is done only through the global X registers and the graph store. This is in contrast to many other imperative languages, where references to local stack frames can be passed as arguments or where stack frames are linked together.

Calling convention The LVM has a single calling convention for user-defined procedures. A procedure has a fixed number of input arguments and no output arguments. The arguments can be seen as call-by-reference parameters, because only references to nodes in the store are passed as arguments.

The parameter are passed in the X registers, where $X_0, \ldots, X_n$ contain the actual arguments. The content of the other X registers is undefined.

We use a caller-save model for registers which means that the caller is responsible for saving X registers into the local environment before an application. After an application the content of the X registers is undefined.

Optimized application The instruction `directCall`(v, n) is an instance of the call-instruction where the compiler statically knows that the procedure is a fixed value and will not change.

The virtual machine optimizes this case by using an unboxed representation for the procedure. Furthermore the test if the number of actual and formal arguments

match is performed only once. The details of these optimized calls are explained in [87].

The performance difference between the optimized and the non-optimized application is approximately a factor of two. A direct call is almost as efficient as a jump. A small overhead has to be paid for the preemption test.

Local environment Local environments allow to store temporary values during a procedure activation. A local environment which allows to store n references to nodes is created with the instruction `allocate`(n). Local environments are addressed with the Y addressing mode.

In the virtual machine the allocation of the local environment is separated from the creation of procedure tasks to allow for optimized allocations in different branches of the computation, e.g. in many procedures no local environment is needed in one of the branches of the computation.

Local environments have the property that they are single referenced, which is an important invariant for memory management. After the deallocation the storage of local environments can be immediately reused. This reuse provides for locality of memory usage which maximizes the use of caches.

Local environments are explicitly deallocated with the `deallocate`(n) instruction. The explicit deallocation allows to reuse memory as soon as possible. An alternate design would be the implicit deallocation when the task terminates. This design would limit the possibilities of a compiler to allocate and deallocate many different environments on one paths of a procedure, e.g. to trim the environment to the current need, and it would incur an overhead even for procedures which do not need an environment.

Example As a very small example we show the bytecode generated for the append function to concatenate two lists

```
fun app (nil, ys, zs)   = unif (zs, ys)
  | app (x::xr, ys, zs) =
    let val zr = lvar () in
        unif (zs, x :: zr);
        app (xr, ys, zr)
    end;
```

The function `app` is compiled into the following pickle

```
vApp:
 proc(s,[],lbl)

...
 definition(0,pb(code:lbl, arity:3, g;[],maxX:3,...))
```

Name/In/Out	description
record/1/1	dynamic record construction
select/2/1	field selection
newCell/1/1	cell creation
cellAccess/1/1	cell access
cellExchange/2/1	cell exchange
newName/0/1	name generation
$*$,div,mod/2/1	arithmetic
future/1/1	future
waitOr/2/0	synchronization
byNeed/1/1	by-need synchronization
featureC/3/0	feature constraint
widthC/2/0	width constraint
raise/1/0	raise an exception
spawn/1/0	fork a thread

Figure 3.18: Built-ins of the LVM.

```
...
lbl:
  matchX(0,ht(nil:l1, cons:l2 else:l3))
l1:
  unifyXX(1,2)
  return
l2:
  getVarVar(3,0)
  getListValVar(2,3,2)
  directTailCall(vApp,3)
l3:
  raiseError(...)
```

3.4.5 Built-in procedures

Similar to the usage of operators in the language definition the virtual machine has built-ins. Built-ins implement core functionality of the LVM which is not directly available through instructions. The built-ins of the LVM are summarized in Figure 3.18.

Built-in procedures are a flexible extension mechanism for adding new functionality to the engine of the LVM.

The designer of the VM has the choice to implement operations as machine instructions or as built-in procedures. The trade-off between these possibilities

Return code	Explanation
`PROCEED`	successful termination
`SUSPEND`	block the thread
`RAISE`	raise an exception
...	other special purpose codes

Figure 3.19: Return codes.

is that the dispatch for instructions is much faster than the application of a built-in. The number of instructions should be small to reduce the complexity of the emulator. The overhead for calling a built-in procedure can, for example, be tolerated if it is much smaller than the time spend for the operation itself, e.g. dynamic creation of an arity. Built-in procedures are also well suited if the functionality they provide is not time critical at all. They are very useful for experimentation.

The instruction `callBI`(vbi, loc) implements the application of built-in procedures, where *vbi* is a reference to a node representing the built-in procedure and *loc* is the mapping of the X registers to the input and output arguments. The built-in procedure is called with the mapping as argument.

Return codes The result of the application of a buil-in function can be successful, it may require to suspend the thread, or it raises an exception. These conditions are signalled with a *return code*. The return codes are listed in Figure 3.19.

When a built-in procedure returns `PROCEED` it was successful and the next instruction is executed.

When a built-in procedure suspends, signaled with the `SUSPEND` return code, it returns a list of transients in the field `suspendVarList` in the register `retInfo`. In this case the worker saves the current task (PC, Y, G) and the X registers. Then it creates a suspension to reschedule the thread when any of the transients in the register `retInfo.suspendVarList` is bound. The application of the built-in procedure is retried when the thread is woken up. The suspension mechanism is explained in Section 3.5.

When a built-in procedure raises an exception then the exception value is put into the `retInfo.exceptionValue` register. The worker is then responsible to search for an exception handler as described in Section 3.4.7.

The `callBI` instruction is a special case of the `call` instruction which is explained in Section 3.4. The compiler generates the optimized built-in call if it statically known that a built-in procedure is applied.

The main difference between the generic application and the built-in application is that the later is an inlined application. For inlined applications the compiler does not generate code to save the global registers X_i into the local environment, because the built-in procedure only modifies the registers marked as output values in the location mapping *loc* and leaves all other registers unchanged.

For example the compilation of the following two functions shows the difference between the inlined compilation of select in `f1` and the non-inlined compilation of a user-defined function in `f2`.

```
fun f1 (x,y) =
  let val z = select(x,y)
  in
     (x,z)
  end

fun f2 (x,y) =
  let val z = g(x,y)
  in
     (x,z)
  end
```

The compilation of `f1` is short and straightforward.

```
% function f1
% X[0]=x
% X[1]=y and z
% X[2]=output
l_f1:
  callBI(vselect,loc([0,1],[1])
  getRecordX(ar(1,2),2)
  unifyValueX(0)
  unifyValueX(1)
  return
```

In the bytecode for `f2` a local environment is needed to save three registers before the application of the function `g`.

```
% function f2
% Y[0]=x
% Y[1]=output
% Y[2]=z
l_f2:
  allocateL3
  moveXY(0,0)
  moveXY(2,1)
  createVariableX(2)
  moveXY(2,2)
  callG(0,2)
```

Bit	0	1	2	3	4 . . . 31
Flag	NeedGC	PreemptThread	IOReady	Timer	unused

NeedGC	Trigger a garbage collection. (see Chapter 4)
PreemptThread	The time slice for a thread is expired.
IOReady	An I/O channel is ready for new data. (see Chapter 4)
Timer	The user timer is expired. (see also chapter Chapter 4)

Figure 3.20: The status register.

```
getRecordY(ar(1,2),1)
unifyValueY(0)
unifyValueY(2)
deallocateL3
return
```

3.4.6 Status register

Before executing a task the worker checks if a bit in the status register is set (see Figure 3.20). The *status register* signals events that have to be handled synchronously to guarantee mutual exclusion for the store. These events are asynchronously detected, e.g. in the memory management layer during the allocation of new memory, when the operating system delivers Unix signals, when preemption or user-defined timers expire, or when I/O channels are ready.

The worker preempts the execution of a thread when any bit in the status register is set. The cost of the synchronization is: reading the status register, a test if it is zero, and a conditional branch.

Discussion Various methods for the efficient integration of I/O are discussed in [81, 5]. For an emulator-based approach our method seems to be well-suited.

One possible optimization is to lower the frequency of synchronization points by using a counter stored in a native register. The counter is decremented at every synchronization point, but the status register is only checked when the counter is expired.

In an implementation of the LVM which supports multiple workers the status register is obsolete. The techniques to synchronize the concurrent workers can be also used to synchronize the asynchronous events.

Another alternative would be to give up fairness of threads and provide primitives at the user level to preempt and yield a thread. This approach is for example chosen for Java: the scheduling policy and fairness assumptions are not specified,

but these are implementation and platform specific. Oz is designed as a language which supports efficient concurrency, which is scalable to thousands of threads. Leaving fairness unspecified would lead to nonportable designs, which depend on certain implementations resp. platforms.

3.4.7 Exceptions

Exception handling is implemented in the LVM with the instruction sequence as follows

`try`(L)
... body ...
`popEx`

The `try` instructions installs the exception handler during the execution of the body and the `popEx` instruction removes the handler.

The `try` instruction first creates a handler task $ex(L, G, Y')$, where Y' is a copy of the current local environment Y, and pushes this handler as a marker on the task stack.

After the installation of the handler the following instructions are executed until an exception is raised or the `popEx` instruction is executed. When no exception is raised during the execution of the exception body the handler task is removed from the top of the stack by the `popEx` instruction.

The generic compilation of `catch (body, handler)` operator does not take advantage of the local and procedure environment. Only if the compiler knows the definition of the `body` resp. `handler` procedure it can generate more efficient code to reuse the environments.

Exceptions are first-class values and the built-in procedure `raise`(i) raises the exception with value X_i. When a built-in procedure returns the exception status code the worker searches for the topmost handler task on the task stack. If such a task $ex(PC, G, Y)$ is found all tasks including the handler task are removed from the stack. Then the exception value is moved to X_0 and the handler task is executed. If no handler is found on the task stack a default handler is executed, which usually prints the exception and terminates the thread.

The main cost factors of the LVM exception handling are

- Two instructions must be executed to install and deinstall the handler if no exception is ever raised.
- For the compiler the exception handler, and the code following `return` are different tasks, i.e. nothing about the content of the X registers, except for X_0 in the exception body, can be assumed.

- Optimizations which reorder instructions have to be very careful to respect the exception semantics, e.g. moving constant expressions out of a procedure is not allowed when this expression could possibly raise an exception.

Handler register A simple optimization of the mechanism to find an exception handler is the introduction of a handler register per thread, which contains a reference to the topmost exception handler task. To allow the efficient update of the handler register all handler tasks are then linked together.

Tail-call optimization Exception handling prevents tail-call optimization for the exception body, because the exception handler has to be explicitly deinstalled with the instruction `popEx`.

It is possible to implicitly discard the exception handler whenever the worker sees such a task at the top of the thread. This would allow to replace the sequence `popEx;return` by a single `return`. A small drawback of this solution is that the local environment cannot be shared between the exception body and the exception handler, but it has to be explicitly copied.

Discussion The LVM exception mechanism is similar to the Standard ML of New Jersey (SML/NJ) implementation of exception handling [7]. In SML/NJ an explicit exception stack of handlers is maintained, which is updated whenever the computation enters and exists the exception body. In the LVM the exception stack and the task stack are integrated, which allows for the tail-call optimization.

In imperative languages, e.g. GNU C++ [99] and the JVM [60], exception handling is implemented with tables, which map a range of program code to an exception handler. When an exception is raised for each stack frame a lookup in the exception table has to be performed. The advantage of exception tables is that no instruction is executed at runtime when no exception is raised.

The LVM design does not use exception tables, because a design goal was that raising an exception should be efficient and enables the use of exceptions as a powerful programming construct for non local exits of recursive functions and blocks.

3.5 Threads

3.5.1 Thread model

The LVM executes at most one thread at a time. A thread can be in one of three states: *runnable*, *running*, or *blocked* (see Figure 3.21).

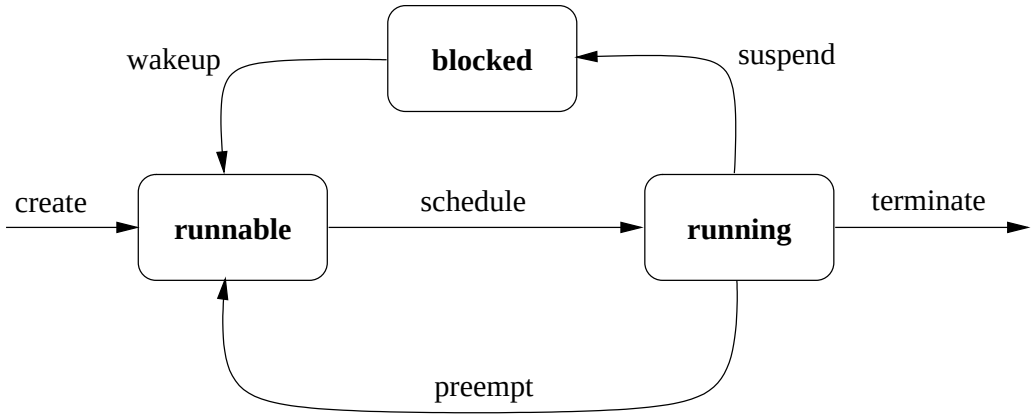


Figure 3.21: Thread states.

A new thread is created with the `spawn` built-in applied to a procedure. The initial task on this thread is the application of the procedure. The new thread is initially in the runnable state, which means that it has the potential to execute its next task.

When a thread is selected for execution its state changes from runnable to running and the worker starts its execution. In the LVM exactly one thread is in the state running, because it has a singe worker.

An execution of a running thread can be *preempted* to guarantee fairness with other runnable threads. In this case the status of the thread is changed from running to runnable.

When the running thread *suspends* on one or more transients it becomes blocked. A blocked thread is *woken up* when a transient on which it suspends is bound.

A running thread *terminates* when its task stack is empty.

3.5.2 Scheduler

The scheduler is responsible for maintaining the runnable threads and assigns a thread to the worker for execution. The scheduler controls the preemption of the thread executed by the worker to guarantees fairness among all runnable threads. The runnable threads are stored in a queue and the scheduler uses a simple round-robin policy to select a thread for the worker.

A preemption timer is started and the worker executes the thread. When the preemption timer expires the time slice for the running thread is over and it is preempted. Preemption of a running thread only happens when the worker is active. During the emulation of instructions the preemption is ignored and delayed until the next synchronization point.

Preemptive scheduling The worker becomes active during the emulation at certain synchronization points. The synchronization points are chosen such that they are met frequently, but not too frequently.

$\delta(t) << d$ The time between to synchronization points $\delta(t)$ should be much smaller then the duration of the time slice d.

$o << \delta(t)$ The overhead at the synchronization points to check if the scheduler requests preemption o should be much smaller than the time between two synchronization points.

The LVM has two synchronization points. The first is the creation of new tasks, e.g. when applying a procedure. The second is when a task is popped from the stack. This scheme guarantees fairness, because unbound computations are only possible through the creation of new tasks[5]. The synchronization point when popping a task is necessary to avoid that the unwinding of a deeply nested recursion does not impose an arbitrary delay on preemption.

Light-weight threads Threads in Oz are extremely light-weight, i.e. thousands of threads can be created and scheduled. The major reasons for the efficiency of LVM threads are that no random preemption takes place and that threads are implemented at the user level and not at the operating system level. Fixed synchronization points for preemption ensure that the state of the engine which has to be saved and restored when scheduling a thread is very small, i.e. the X register, the self register, and the current task have to be saved and restored. The overhead for testing the preemption condition at the synchronization point is small.

Fairness The scheduler guarantees fairness for the execution of all runnable thread by preempting the worker. The preemption condition can be a timeout on a timer provided by the operating system or a timeout on the number of instructions (or tasks) executed by the worker.

Preempting the worker means that the worker returns the thread to the scheduler. It does so only after the execution of the current task is stopped. The fairness condition is fulfilled, because the execution of every task is bound by an upper limit.

One reason for delaying the preemption is that this gives a strong invariant for atomicity: the execution of a task is never interrupted. While executing a task the

[5]Except for naive procedures implemented through the LVM native API. The time for these procedures is potentially unbound.

virtual machine can be in an inconsistent state, e.g. undefined values in registers and in the store, as long as it is consistent again when the execution of the task stops.

The second reason is that the global X registers are shared among all threads. To make this feasible a thread has exclusive access to them during the execution of a task. Whenever a task stops the X registers are saved on the thread by creating a save task, which restores the X registers when the thread is executed again.

Discussion The scheduler is an orthogonal unit in the virtual machine. Therefore it can be extended easily to support sophisticated scheduling techniques, e.g. priorities or resource-based scheduling.

A disadvantage of this user-level thread package is that it cannot take advantage of multiples processors. Two models are proposed to use multi-processors. One model is a parallel implementation of the LVM [80] and the second model is a distributed implementation of the LVM which uses shared memory as an efficient communication layer [39].

3.5.3 Suspensions

Threads can suspend on transients. This means that the thread is removed from the runnable queue of the scheduler until the transient is bound.

Suspending a thread on a transient involves the following steps

- A suspension is created, which contains a reference to the thread.
- This suspension is hooked onto the transient.
- The worker is informed that the current task is suspended.

A suspension is *woken up* when a transient is bound. Waking up a suspension informs the scheduler that the thread is runnable.

Using suspensions as indirection between transients and threads is necessary because it is possible that a suspension is hooked to many transients. To explain this we use the built-in `waitOr`(x, x'), which suspends the thread if both x and x' are transients. If one of these transients is bound the thread is woken up. In this case the suspension has to be unhooked from the other transient to avoid further wakeups.

To optimize the wakeup operation the unhook operation is done lazily. The suspension is marked when the wakeup is done. It is not unhooked from the other transients suspending on the disjunctive condition. Suspensions marked as done are skipped during the wakeup.

3.5.4 Events

The alarm mechanism of the operating system allows to trigger a signal handler after a certain time. In the engine this alarm signal is used to execute a check function at regular time intervals. This function serves different purposes:

- **PreemptThread** The expiration of the time slice of a thread is checked.
- **IOReady** I/O channels are watched for data.
- **Timer** User-defined timer events are handled.

The check function is triggered every 10 ms and sets the corresponding bits in the status register. As explained above the engine eventually preempts the execution of instructions and handles the events detected in the check function.

Threads are preempted at every 5th clock tick, which means that the time slice of a thread is 50 milliseconds. This is implemented with an alarm timer which is initialized when a new thread is scheduled.

In the LVM it is possible to block a thread on the ability to read resp. write an I/O channel. The implementation maintains a list of all threads waiting for I/O and their resp. I/O channels. During execution of the check function the operating system is polled if one of these I/O channels is ready for read resp. write.

An alternative approach to polling I/O would be asynchronous I/O, which has the advantage that the operating system informs the engine when I/O is available. The drawback of asynchronous I/O is that it is not portable between different platforms.

The LVM supports soft real-time control with timers. A thread can be delayed for a certain amount of time with the primitive `delay t` where t is the time to delay in milliseconds. This is implemented with a list of threads. This list is sorted according to the time after the delay. During the execution of the check function only the time after the first delay is tested for expiration.

3.5.5 Discussion

The thread model of L has the property that threads are explicitly created. Before reaching this model we investigated two other approaches: the fine-grained concurrency and jobs as an intermediate granularity.

In the fine-grained model the composition of two expressions is concurrent. Sequential execution can only be specified using data flow synchronization. In

AKL [47] this concurrency model was used. It has the advantage that it supports very well the declarative constraint programming style.

The fine-grained model introduces a huge burden on the implementation, because many optimizations possible in a sequential environment are not possible, e.g. the lifetime of X registers is much shorter. A major disadvantage of fine-grained concurrency wrt. the language definition is that it is very difficult to combine stateful programming with data flow-only synchronization.

A hybrid job model was designed, where a job is a sequence of expressions with a sequential execution strategy. A program is a sequence of jobs, which are also executed sequentially, but when an expression suspends a new concurrent thread is created for the suspended job. This model was designed as a compromise between the fine-grained concurrent constraint approach and the explicit concurrency approach.

At the LVM level this job model has some nice properties, e.g. in most cases the thread creation and scheduling overhead was saved, because jobs did not suspend frequently. On the other side the maintenance of the jobs incurred an overhead, because the tasks on the task stack had to be grouped into jobs. It turned out that the implicit thread creation in the job model was to complicated as a programming concept.

The compromise chosen in Oz is now such that threads must be explicitly created and for constraint programming built-in light-weight threads called propagators are used.

3.6 Spaces

In this section we introduce the extension of the virtual machine, which are needed to support multiple computation spaces.

The basic services provided by the virtual machine are the execution of threads situated in spaces and the detection of entailment and disentailed. The virtual machine is extended with an additional storage area for spaces, with a trail, and with a space register. A single instruction for the deep guard conditional is added to create a new space and to synchronize on entailment or disentailment of this space.

The main refinement of the space model introduced at the LVM layer is the representation of multiple computation spaces multiplexed into a single store. We introduce the script technique for maintaining multiple transient bindings and compare it to the binding window technique.

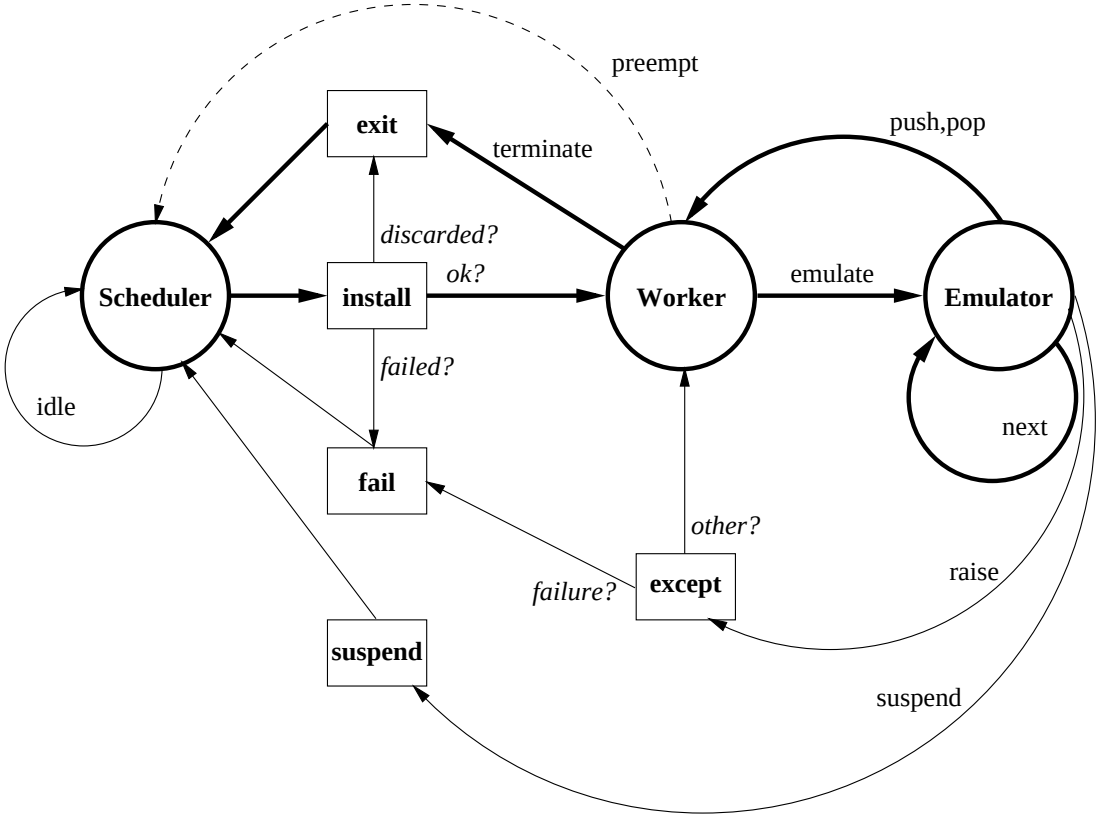


Figure 3.22: The extension of the engine for spaces.

3.6.1 Overview of the extended engine

The engine model is extended for spaces with hooks for the installation, termination, and suspension of threads and for the detection of failure exceptions as outlined in Figure 3.22.

The hooks are drawn as boxes and have the following functions:

install When a thread is selected for execution its space is installed, i.e. the script is executed.

exit When a thread is terminated the entailment and stability condition are tested.

except The exception mechanism is extended to detect failure exceptions.

fail If a failure exception is raised and not handled by an exception handler or the installation of the script fails, then the space is marked as failed and considered as disentailed.

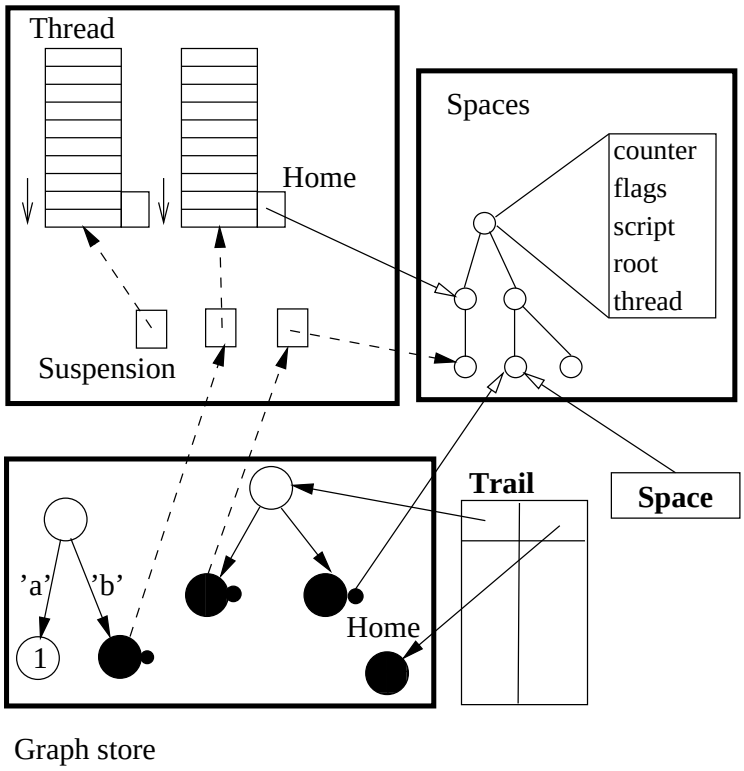


Figure 3.23: Engine state with spaces.

suspend When a thread is suspended a hook is needed for detecting stability, which is not further explained in this thesis.

A new compartment of the engine is the store of spaces. Figure 3.23 shows a store, where situated nodes and threads are labelled with their home space.

The LVM is extended with a *space register* `space` which contains the current space. The `trail` keeps track of the installed speculative bindings.

A space has a reference to its parent space, a counter for the number of non-terminated threads, a script containing the speculative bindings, a number of flags, a reference to the root node, and a reference to the thread containing the conditional which waits until entailment or disentailment of the space is detected.

3.6.2 Threads and spaces

Threads are situated in spaces. This is implemented in the virtual machine by a references from the thread to its space. This means that the thread "knows" its space. No references from the space to its threads are needed. The number of

non-terminated threads is counted per space to decide one part of the entailment condition.

When new threads are created they inherit the space from the current thread. In this space the thread counter is incremented. A new space is equipped with an initial thread.

Because the engine refers to the space of the current thread very often, this is store in a `space` register. The space register is initialized from the thread when its execution starts.

In the LVM all runnable threads are maintained by the scheduler in a global thread queue. An alternate design to a global queue would be an organization of the runnable threads per space. These local queues are used in AKL and have the advantage that the locality of execution is exploited[6].

When a space is failed all threads belonging to this space must be terminated. To avoid references from a space to all its threads this is done lazily. Lazy means that when a thread situated in a failed space or below is scheduled for execution it is discarded during the installation, when the failed space is discovered.

3.6.3 The script technique

The basic problem of deeps guards is to efficiently represent speculative bindings. In this section we describe the *script technique* for maintaining multiple bindings of transients in different spaces.

Every space has a script. The script contains all speculative bindings of global transients of a space. The script contains pairs of nodes: a global transient node and its speculative binding.

To efficiently access the current binding of transients the space of a thread is installed. A space is installed by installing its script. The *installation* of the script makes all the speculative bindings active by executing the unification algorithm with every pair of nodes in the script.

The speculative bindings have to be undone when the worker executes another thread in a different space. For this purpose the speculative bindings are pushed onto a stack, called the *trail*. Speculative bindings may be created during the installation of the script and during the execution of a thread in a space.

The entries on the trail are pairs of a reference to the tagged node which was speculatively bound and its old content, e.g. its old tag and pointer.

[6]This approach is taken for propagators, which implement built-in threads for constraint propagation.

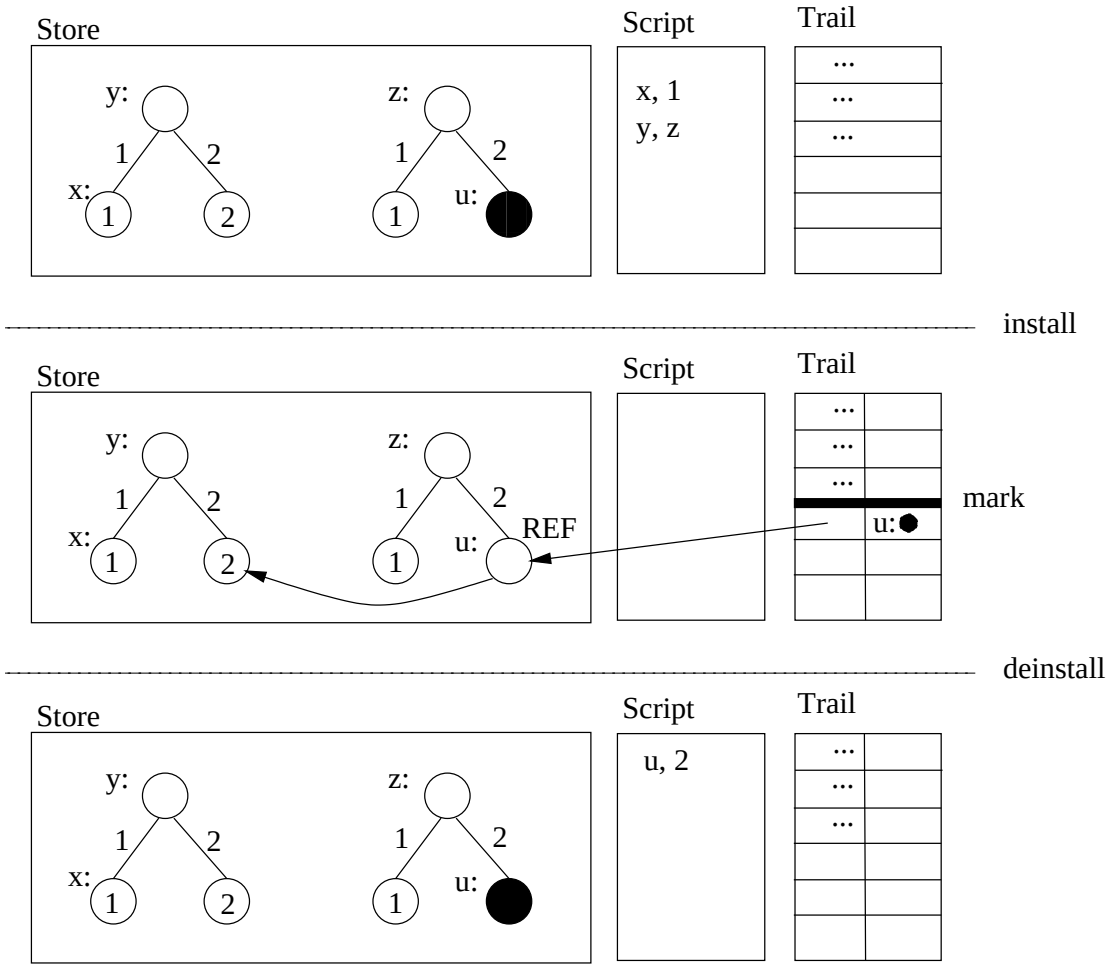


Figure 3.24: Installation and deinstallation.

When the worker leaves a space it is deinstalled. *Deinstallation* writes all speculative bindings from the trail into the script and retracts the speculative bindings in the store.

Bindings of local transients are not trailed and correspondingly never written into the script. These bindings need not to be deinstalled, because the local transients are not visible in the parent space.

Figure 3.24 shows the installation and deinstallation of a space. During the installation the unifcations of `x` with `1` and of `y` with `z` are performed. We assume that `x` was already bound globally and therefore the first unification is a no op. The second unification speculatively binds `u`, which is trailed and during the deinstallation this speculative binding is written to the script.

Installation and deinstallation of paths L allows arbitrarily nested spaces and the worker has to install the scripts in all spaces from the root of the computation tree to the space of the thread.

The algorithm to install a path from the root space to a target space has two phases: a collect phase and an activate phase. The collect phase starts from the target space and collects all spaces on the path to the root of computation tree on a stack. For this purpose every space needs a references to its parent space. In the activate phase the spaces on the stack are installed.

The deinstallation of a path simply starts from the current space and deinstalls all spaces up to the root space.

If the worker deinstalls a space and installs another space this can be optimized by performing the deinstallation only until a common ancestor of both spaces is reached. The installation of the path can started from this common ancestor. To efficiently find the common ancestor spaces are marked when they are installed. The collection phase starts as usual at the target space, but it stops when a space is found which is marked as installed. This space is the common ancestor and the deinstallation and installation procedure can proceed from there.

To write the local bindings into the script of the correct space during the deinstallation the trail has to be segmented with one segment per space on the path. When the worker starts to install a space a new segment is allocated on the trail. When a space is deinstalled the transients in the top segment are deinstalled and this segment is removed from the trail.

The trail/script technique outlined above requires that the binding of a transient in the store can be undone. This implies that the virtual machine is not allowed to shrink existing reference chains, while speculative bindings of transients are stored on the trail. This requires for example special care when doing a garbage collection. In the LVM garbage collection is performed when all spaces are deinstalled and no speculative binding is active.

Propagation The consistency condition for bindings in a tree of computation spaces is that every transient is bound at most once on every path from its home to any descendant space. To preserve this condition a binding is propagated to all child spaces. *Propagation* removes the speculative bindings and reexecutes the unification algorithm with the new and the old binding in the subordinated spaces.

To find all speculative bindings the suspensions are extended for spaces. When a space is deinstalled which has speculative bindings a suspension is created which has a reference to the space. This suspension is hooked to every transient which is speculatively bound in the space.

The propagation of bindings is not done immediately when a new binding is added, but it is done lazily. A wakeup thread is created in all spaces containing local bindings. A wakeup thread has an empty task stack. The purpose of the wakeup thread is to install its space and thereby performing the implicit propagation of bindings.

The propagation happens implicitly during the execution of the script. As explained above the script contains pairs of nodes, which are unified during the installation of the script. In the case of propagation both nodes are possibly determined values. The application of the unification algorithm guarantees that the equality of the two nodes is preserved or that the space is failed.

An interesting property of the installation technique is that constraint propagation is done lazily. Whenever a thread is executed in a space it is ensured that all constraints are propagated to this space, because the script is installed before the execution starts.

A little optimization is implemented in the LVM which ensures that for every space a wakeup thread is only created if needed. When a thread enters the running state its space is marked as `propagated`. If this mark is already set the creation of the wakeup thread is omitted, because a runnable thread situated in this space exists which ensures that the propagation takes place. The mark is deleted when the space is installed.

Failure A space is failed when a failure exception is raised and not handled. The failed space is deinstalled[7] and marked as `failed` to allow for the lazy termination of its threads and the threads in child spaces.

Entailment The entailment condition for a space has two parts: it contains no speculative bindings and and all threads are terminated. For the LVM the first condition is equivalent to the test if the trail resp. script is empty.

The test that all threads are terminated can be implemented with a counter, which is incremented for every new thread and decremented for every terminated thread.

It is sufficient to check for entailment when a thread terminates. Both conditions are only fulfilled together if the last thread terminates.

Merging The merge operation for entailed spaces consists of three parts: making the merged space transparent, merging the script, and merging the thread counter. Merging the thread counter simply adds the counter of the merged space to the current space.

[7] Creating the script is not necessary, because the space will never be installed again.

A merged space is marked as `transparent`, which means that all operations on transparent spaces are redirected to the parent space. Spaces are made transparent to avoid a complex machinery for updating all references from transients, suspensions, and threads to the space. This is similar to the technique for binding transients and has the same overhead for dereferencing.

The speculative bindings stored in the script of the merged space are added to the current space through unification and thereby propagated to the subordinated spaces.

Transient - transient bindings Bindings of transients to transients have to be treated specially. The main reason for potential problems is that transients are not ordered and the unification of two variables may bind them in any order. For example when executing the following code fragment it may happen that in the conditional (1) the transient `x` is bound to `y` and at position (2) `y` is bound to `x`.

```
val x=lvar();
val y=lvar();

spawn fn () =>
  cond (fn () => unif (x, y), ..., ...); (* 1 *)

unif (y, x); (* 2 *)
```

In this situation the wakeup mechanism would not trigger propagation, because in (1) a suspension is only added to `x` and in (2) only the suspensions of `y` are woken.

Two possible symmetric solutions to fix this problem are:

1. Suspensions are added to both nodes if a transients is speculative bound to another transient.
2. The suspensions of both transients are woken, when a transient is bound to another transient.

The first solution is realized in the LVM. It has the advantage that, in the case of binding a local transients to a value, it is not necessary to test that the bound value is a transient. In this case work has to be done only when a speculative binding is written into the script[8].

Both solution have the problem that too many unnecessary wakeups may be performed. Therefore we did an experimental implementation of special kinds

[8] A second reason for this approach is that stability can be detected easily

of suspensions for this case. These suspension allowed to perform the wakeup exactly when needed. It turned out that the case of speculative bindings between two transients occurs very infrequently and no optimization of this case is needed.

3.6.4 Binding windows and relative simplification

Many techniques are proposed in the literature to provide multiple views on trees of constraint stores with shared variables. Especially in the context of OR-parallel Prolog implementations sophisticated techniques are developed. Gupta and Jayaraman [33] give an overview of the known techniques and classify these according to three efficiency criteria: constant time access to the current binding in a space, constant time thread creation, and constant time thread switching. They show that at most two of these criteria can be simultaneously satisfied.

Beside of these three efficiency problems a virtual machine with deep guard operators like the LVM must also implement the entailment test and the merge operation of two spaces, which are not needed in Prolog implementations.

In the following paragraph we present two other techniques to represent multiple bindings.

Binding arrays The binding array method was introduced in the context of Prolog implementations by D. S. Warren [113]. The motivation for the binding arrays was to allow for an exploration of the search tree using other strategies than the built-in depth-first order of Prolog, but keeping the same performance as backtracking. The technique was independently developed for OR-parallel execution of Prolog by D. H. D. Warren [112].

The basic idea of the method is to allocate forwarder lists in every space. These forwarder lists maintains the speculative bindings. When a global variable is bound in a space an entry is added to the forwarder list, instead of modifying the variable node and trailing it. To access the value of a variable a lookup in all forwarder lists up to the home of the variable is executed, until a binding is found or the home space is reached.

The lookup is optimized with a binding array. This is a structure allocated per worker which contains all forwarders on the path from the current space of the worker to the root space. The binding array allows to access variable values in constant time[9].

The complexity of this scheme for a single worker traversing the search space depth-first is the same as for backtracking, because the overhead for dereferencing, binding, and unbinding is constant.

[9]The binding array can be implemented as an array because in each path of the computation tree the variables can be numbered consecutively.

The overhead for switching the context is linear in the number of speculative bindings which is acceptable because context switches are assumed to be infrequent compared to the amount of work done in one space.

The binding scheme of Penny The binding scheme of the parallel AKL system, Penny [65, 64], uses a simple forwarder list without binding arrays. The authors argue that this simple scheme is very good for typical applications, because context switching can be done in constant time and the forwarder lists are typically very short. Furthermore the trees of computation spaces are typically flat and bushy and not deeply nested.

To efficiently find the suspended binding the forwarder list contains, beside speculative bindings, also local suspensions. The suspension for a binding is added to the forwarder list in the parent space. If the parent is the home space suspensions are added to the suspension list of the variable itself.

If a global variable is bound the forwarder list of the current space is searched for suspensions. If a local variable is bound the suspension list of the variable are woken up.

The beauty and the beast The beauty and the beast algorithm [78] is a true incremental algorithm for deciding entailment for flat guards with feature constraints. The basic idea is to avoid any kind of unnecessary recomputation by creating a so-called beast storing all the work already done.

This algorithm was only studied in theory but a practical implementation is still outstanding. Under the assumptions that speculative bindings are very infrequent it is questionable if this approach leads to an improved algorithm.

Situated simplification A formalization of the entailment and disentailment tests and proof of its correctness for rational tree constraints for deep guards is given in [77].

The authors define the situated simplification as an extension of the unification algorithm, which propagates bindings immediately. When the unification terminates the path consistency condition holds, which states that at most one binding for every variable on every path exists.

In the situated simplification and the beauty and the beast algorithm the equivalence sets of structures discovered during the unification are recorded to avoid their recomputation.

Comparison The script technique as implemented in the LVM is a simple binding method if the virtual machine has a single worker. It is also used in the sequential implementation of AKL [47]. AKL only supports Prolog structures and the extension of the script technique to records is defined in [98].

The reason for using the script technique for implementing multiple in LVM can be summarized as follows:

- The virtual machine has a single worker. This implies that at every moment only a single view on the bindings has to be efficiently supported.
- Spaces are used primarily for encapsulating computations for constraint programming and search, where the vast amount of time is spent in propagators and for cloning.
- The overhead for context switches for the worker, i.e. moving from one space to another, is small compared to the execution time within a context. The time slice for the execution of a thread is much longer then time needed to switch the context. In constraint solving problems many threads run in the same context.
- Only very few global variables are speculatively bound. The overwhelming majority of bindings are for local variables. The overhead for implementing a truly incremental algorithm is therefore not related to its benefits.

In our implementation a suspension is created for each deinstallation of the script. This can be optimized by creating a single thread per space which has the role of the wakeup thread and which is responsible for propagating bindings into this space.

The script technique in the LVM has quadratic complexity for examples with incremental bindings, because

- All bindings in the script are executed, even if a single binding must be propagated.
- Structure-structure bindings are not stored in the script and must be reexecuted for every installation of the script.

3.7 Other virtual machines

3.7.1 Prolog Abstract Machines

The design of LVM was influenced to a great extend by Warren's design of the abstract machine for Prolog, called WAM [110, 111, 1].

The LVM uses the basic techniques developed for the WAM to represent symbolic structures on a heap. This representation was adapted for records. The LVM uses the same optimized representation of lists as found in many Prolog implementations. The optimizations for compiling unification into low level instructions can be directly applied.

The LVM supports logic variables, but their representation could not be optimized into self-references, because variables are situated in spaces and need to represent their home pointer.

In the LVM variables are never allocated on the stack resp. in Y registers, but only on the heap. Therefore no concept of unsafe variables is needed.

L does not support backtracking as primitive search strategy, but first class spaces, which allow to efficiently program different search strategies [89]. The major difference for the virtual machine is that many environments resp. spaces are active simultaneously. Instead of generating choice points and trailing changes the virtual machine supports cloning, i.e. copying, of spaces.

Like the WAM the virtual machine of L has global resp. temporary registers X and local resp. permanent registers Y. Arguments are passed through X registers in both machines. To support first-class procedures the LVM has an additional register G for addressing the procedure environment.

The design of the LVM is targeted for an emulator-based approach and not for a high-performance native implementation. It is expected that the techniques developed for high-performance Prolog implementation, e.g. Van Roy's [105], or Taylor's [101], can be adapted.

3.7.2 The abstract machine of AKL

The Agents system [47] is the first implementation of AKL [27, 37] a deep guard programming language. Many implementation techniques for deep guards were pioneered in the Agents system.

L radically differs from AKL in its control strategy. Concurrency in AKL is fine grained as opposed to L, which supports course grained concurrency. The implementation concurrency for L requires to support preemption of computations to guarantee fairness among threads and reactivity. Exception handling is not supported in AKL, because it is only useful in the paradigm of threads.

Search in AKL is built-in and implicitly triggered. In L search is first-class [89].

The LVM supports a richer set of data types than AKL, e.g. names, records, and first class procedures are essential parts requiring new implementation techniques. The basic data type for stateful programming in AKL is a port. Compared to cells in L the implementation of ports is of a similar complexity.

3.7.3 LIFE

The tree data structures realized in the LVM are based on the foundational work on records [98] and features [4]. As part of the work on LIFE [2, 3, 78, 79] record-like structures were analysed and implemented in a concurrent constraint framework.

3.7.4 The Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) [60] is a machine designed for the implementation of Java. Java is an imperative concurrent programming language using the object-oriented imperative programming paradigm with automatic memory management.

The JVM is designed to support a wide range of platforms including embedded systems. The language requirements wrt. fairness and unsynchronized update and access in Java are very weak. This weakness simplifies the implementation of the JVM compared to the burden on the LVM to respect the interleaved semantics of L.

The JVM is a stack based machine, i.e. it has no general purpose machine registers, but operands and arguments are passed through a data stack. This approach compared to the register-based approach of LVM simplifies the compiler and the implementation of the virtual machines in some aspects, but many optimizations, e.g. using machine registers for passing arguments, requires non-trivial runtime optimizations [23, 24].

The design of the exception mechanism in the JVM is optimized for the case that exceptions are raised only in rare circumstances. An exception handler incurs no overhead at runtime if no exception is raised. When an exception is raised all stack frames are scanned to test if its PC refers to a region protected by an exception handler. Using exceptions for non-standard control primitives is not feasible with this approach.

The JVM does not support tail-call optimization, instead the usual loop constructs of imperative languages, e.g. while and for, are supported. The design of the JVM does not prevent tail-call optimization, but it seems that tail-call optimization has no priority for JVM developers.

3.7.5 Functional languages

Many ideas from the implementation of functional languages [7, 76] apply very well to the LVM. First-class procedures in L are very similar to first-class functions

and compilation techniques like closure conversion and continuation passing could be easily adapted to the LVM.

Closure conversion transform procedures such that its free variables become additional formal arguments. To such a converted procedure the values of the closure are passed as additional actual arguments. This technique makes the G addressing mode obsolete and is especially useful for native implementations and for elaborated compiler optimizations.

The continuation passing style is an alternative implementation to a stack based implementation of threads. In this approach every procedure is converted into a procedure with a continuation as additional argument. The continuation is tail-called at the end of the procedure instead of returning from the procedure. The continuation passed when calling a procedure is a closure representing the code which follows the procedure call.

Using continuation passing style for the a virtual machine simplifies the representation of threads, but requires to create a lot of closures. It pays off if the compiler aggressively optimizes the creation of closures.

The convention for returning values differs between L and functional languages. In L logic variables are used to pass values back to the caller of a procedure. In the LVM this is optimized for creating symbolic data structures. In the implementation of functional languages values are usually returned through registers. This typically avoids the overhead of creating and binding variables and often leads to a better register usage. On the other side it can hinder the tail recursion optimization, e.g. the L procedure for concatenation `app` is tail recursive, whereas the functional version is not.

Many functional languages have single argument functions. Multiple arguments are realized with pairing. To get the same efficiency as possible with multiple arguments a technique called deforestation [109] can be used. The basic idea of deforestation is to delay the pairing as far as possible. If a pair for example is passed as an argument to a procedure its fields are passed separately and they are never combined into the structure if the procedure directly decomposes its argument. In the LVM we use a similar technique for the implementation of first-class messages in the object system.

A major difference between L and many functional language is the type system. In L everything is dynamically typed, whereas functional languages, like Standard ML and Haskell, have a strong type system. The dynamic type system of L hinders many optimizations which take advantage of static type information, e.g. avoiding dereferencing and dynamic type tests and representing values as unbox and untagged data structures.

Reppy [81, 82, 83, 84] describes a concurrent extension (CML) of Standard ML. The communication primitive of CML is a first-class channel with two synchronous operations. `accept` reads from and `send` writes to a channel. Both

operations block until a pair of threads meet at a channel where one performs an `accept` and the other a `send` operation.

The implementation of CML is build on top of Standard ML using its primitives for first-class continuations and asynchronous signals to implement light-weight threads. When a signal occurs the current continuation is grabbed and passed to a signal handler. This allows to preempt a thread with its state captured in the current continuation.

3.7.6 Erlang's virtual machines (JAM, TEAM/BEAM)

Erlang [9] is a concurrent functional language designed for telephony applications. Two aspects of the language design are different compared to L: threads (which are called processes in Erlang) have no shared memory and the language does not support stateful data types.

As a functional language Erlang requires proper tail-call optimization. The communication is based on a message queue per process and a wait primitive to synchronize on messages in queue.

For Erlang two sequential virtual machines were designed: JAM [8] an emulator-based stack machine and TEAM/BEAM [41] a high-performance native implementation with a register based intermediate language.

Both implementation use separate stacks and heaps for every thread. The Erlang implementation is also influenced by the WAM, e.g. for the representation of data structures they use tagged pointers and pattern matching is implemented with indexing. Environments are allocated per pattern rule, which is similar to the WAM where the environment is allocated per clause.

Similar to the LVM Erlang has light-weight processes with a well-defined semantics. The implementation is a single threaded OS process with a round robin scheduler and possible preemption when executing calls.

The memory management of Erlang is based on a copying garbage collector. Garbage collection is performed on a per-thread basis, because every thread has its own heap. A nice property of Erlang is that no cyclic data structures can be created, which simplifies the garbage collection algorithm.

The overhead for garbage collection in Erlang is very high for examples with many threads and a lot of communication. The problem occurs because threads have no shared memory and the messages must be copied between the threads. The problem is further enlarged by the fact that object oriented programming is supported as active objects with a thread per object. Erlang has no stateful data structures and hence no possibility to express objects without thread.

As a summary the shared store for threads in the LVM has the advantage that no data structures must be copied during the communication among threads. On the other side the technique to allocate memory per thread in Erlang has the advantage that independent and concurrent garbage collection for each thread is possible. Furthermore the distribution of threads among many sites and the mapping of threads to multiple processors becomes simpler.

3.8 Summary of the design principles

Intermediate level of abstraction The virtual machine hides irrelevant details of concrete machines. It provides sufficient high-level abstractions to avoid unnecessarily complicated compilers. It provides enough low-level abstractions to allow the compiler to generate code which can be efficiently emulated.

A virtual machine is a good implementation compromise for a new programming language, which changes frequently and where experiments with new ideas are performed. A virtual machine is not as flexible, wrt. changes, as an interpreter, but its performance is much better.

Another advantage of a virtual machine is portability. The Mozart implementation, which is written in C/C++, has been adapted to many platforms.

Emulator-based implementation The machine language is designed for an emulator-based implementation. This means that the emulation overhead should be minimized. Therefor the instruction set is coarse-grained: many micro instructions are combined into one LVM instruction.

An intermediate language as target for native compilation has to be designed very differently. The work of Van Roy [105] and Taylor [101] on high-performance compilation of Prolog indicates that the intermediate language has to be at a very low-level and close to the hardware to reach the speed of C/C++. This is definitely not the case for the LVM, which has for example high-level graph rewriting and synchronization instructions.

Using a simple macro expansion of the LVM machine code to native code will surely give some speedup, but it is not the right track to reach a high-performance implementation.

Single worker The virtual machine is designed to run on single processor machines. A simple scheduler for concurrent threads is built into the virtual machine. The single processor model gives strong invariants for atomicity and simplifies the implementation of the interleaved semantics of L.

Multiple worker architectures for concurrent constraint languages are studied in [80] and [64]. In [80] the coarse-grained parallelism on the thread level of L is exploited. [64] exploits fine grained parallelism in the language AKL. Both projects show that a sequential virtual machine is a good starting point to explore parallelism.

Shared memory architecture The graph store of the LVM is shared between all threads. This differs from message passing architectures, where all threads have their own memory and communication between threads is done by message passing. The only means to communicate between threads in the LVM is through shared nodes. The LVM has no message passing primitives built-in, but they can be expressed efficiently using records, transients and cells.

A shared memory architecture has the advantage that data structures need not to be copied when communicating between threads. Only a reference to a node in the graph has to be actually sent from one thread to another.

For parallel and distributed implementations shared data structures require more effort in the synchronization code of the implementation, but for programmers shared data structures are very powerful.

Memory management in shared architectures is also more ambitious: to reclaim the memory of nodes potentially many threads are involved and have to be synchronized. In Erlang implementations [41, 8] every thread (called process there) has its own memory management. Non-shared memory architectures allow for a better real-time control, because threads are better decoupled. In the LVM (see chapter Section 4.6) we use a stop and copy collector, which stops the execution of all threads, reclaims the memory and after that restarts the execution of the threads.

Automatic memory management The LVM provides automatic memory management. Automatic memory management is well-understood and standard in modern high-level languages [114, 115].

The basic garbage collection rule for the LVM is that only the nodes reachable from the runnable threads and threads pending on I/O or the timer are live data. In Section 4.6 the implementation issues for the automatic memory management in the LVM is discussed.

First-class procedures The LVM has direct support for procedures with lexical scope and infinite extend, so called first-class procedures. Basically this means that the procedure application installs the environment captured at the procedure definition. The virtual machine has an additional addressing mode for

this environment. In an emulator-based implementation the support of first-class procedures comes almost for free.

An alternative approach to compile first-class procedures is closure conversion [7]. This technique converts first-class procedures into procedures with an additional argument containing the captured closure. The advantage of this technique is that no additional addressing mode is needed. A disadvantage of closure conversion is that it adds a level of indirection to address a node.

Tail call optimization The virtual machine has no loop constructs, but implements tail call optimization, i.e. if the last instruction of a procedure is an application, the stack frame of the caller is removed before the application. Tail call optimization allows to implement loop constructs efficiently. It has additional expressiveness, because any tail call is optimized and for example mutually tail recursive procedures don't need space on the stack of the thread.

In an emulator-based implementation tail calls can be implemented almost as efficiently as jumps. Therefore it is not necessary to complicate the compiler and engine with loop constructs.

Graph abstraction The graph abstraction is the canonical representation of data-structures in high-level languages with automatic memory management. A unit with links to other nodes is the single primitive abstraction for the representation of a value. The unit itself contains type specific scalar information and the links are directed and ordered connections to other nodes.

With this single concept all primitive language data types can be implemented efficiently. The graph abstraction maps very well to imperative data structures and automatic memory management is straightforward.

The store of the LVM is designed such that it provides for efficient representations of dynamically typed values for an emulator based VM. The underlying assumption is that the compiler does not compute static type information, e.g. an actual argument of a procedure (user-defined or built-in) can be of an arbitrary type and the VM has to handle it dynamically.

The store has to represent many different types of values. We use a layered approach. The core layer contains a few main data types, which are implemented highly efficient using tagged pointers. The basic layer, which contains the majority of types, is implemented with tagged objects. The extension layer, which opens the system to add new data types, uses objects with late binding.

The layered approach has the advantage that efficiency can be traded with simplicity, e.g. experimental data-types can be added easily and the essential data-types, e.g. integers, references, and transients, can be optimized.

Transient values are basically used for constraint programming and synchronization of threads. The store is designed such that transient values are almost gracefully degrading. If transient values do not occur in programs there should be no penalty. The major reason why this could be achieved in the LVM is, that all values are represented with dynamic type information and the test for determination can be integrated at no cost with the type test.

Another design goal is modularity and orthogonality of data structures. Data types are implemented in the LVM independently. The glue is the tagging scheme at the core layer, the tagged objects at the basic layer, and a virtual function interface at the extension layer.

I/O as orthogonal concept Input and output is not integrated into the virtual machine. I/O is modeled with ports as endpoints for communication with the outside world. A port [49] is an abstraction for many many-to-one communication with a stream for the reader and a send procedure for the writer. Ports can be easily expressed in L (see [96]).

Output is modeled as a port to which messages can be sent from MyOz and which have some impact on the outside. Input is modeled as a port on which messages arrive from the outside.

No limitations The virtual machine imposes no artificial limitations: the number of local registers Y is unlimited, arbitrary many threads can be created and scheduled, the graph store is unlimited, arities and the number of subtrees can be arbitrary large, an infinite number of new names can be generated, integers are not limited. These requirements simplify the compilation of the high-level language into the machine language, but they require some effort when implementing the virtual machine.

Control-stack and data heap The stack of tasks in threads is solely a control stack and the data structures of the language are stored on the heap. This setup clearly separates control from data. This separation guarantees for example that the tail-call optimization can be applied for every tail recursive application.

Built-in procedures Some of functionality of the LVM is implemented as built-in procedures, where performance is not an issue. This allows to keep the number of machine instructions small and focussed on the performance critical aspects.

Built-in procedures can also be used as a flexible mechanism to extend the virtual machine.

With respect to compiler optimizations built-in procedures can be handled like other machine instructions, e.g. an application of a built-in does not invalidate the contents of X registers.

Chapter 4

Implementation aspects

In this chapter some aspects of the implementation of the LVM in C++ are discussed. The main focus is on the representation of the data-structures in the store.

4.1 Storage representations

In the LVM the type of every unit is available at runtime and the implementation has to dynamically represent these types of units.

The main techniques for representing dynamic type information are tagged objects and tagged pointers. Typically an implementation has to find a compromise using a hybrid mix to trade the simplicity of tagged objects vs. the efficiency of tagged pointers.

The virtual machine supports more types than the language, because various subtypes have optimized representations, e.g. lists and tuples. The LVM tag scheme uses a representation, which allows for speed and memory optimizations of frequently used data types.

The operations on dynamically typed values are *type tests*, *boxing*, and *unboxing*. Types tests require the type of a unit and test if this unit is of a certain required type. Boxing creates a dynamically typed unit. Unboxing extracts the raw information from a dynamically typed unit.

In a language where virtually all units are dynamically typed, these operations are executed most frequently and therefore every optimization contributes significantly to the performance of the whole system.

4.1.1 Tagged objects

Tagged objects are simple data structures which have a type field and additional fields depending on the type. Tagged objects of a certain type can be implemented as subclasses of the class `TaggedObject`.

```
class TaggedObject {
protected:
  int type;
  TaggedObject(int t) : type(t) {}
public:
  int getType() { return type; }
}
```

A list element `Cons` for example can be implemented trivially as a tagged object with two additional fields for the head and the tail of the list.

```
class Cons : public TaggedObject {
protected:
  TaggedObject *head;
  TaggedObject *tail;
public:
  Cons(TaggedObject *h,TaggedObject *t)
    : TaggedObject(CONS), head(h), tail(t) {}

  TaggedObject *head() { return head; }
  TaggedObject *tail() { return tail; }
}
```

Similarly integers can be implemented as tagged object with an additional integer valued field.

```
class Int : public TaggedObject {
protected:
  int val;
public:
  Int(int v)
    : TaggedObject(INT), val(v) {}

  int getInt() { return val; }
}
```

The main advantage of the tagged object implementation is its simplicity and regularity, e.g. the memory management can use the invariant that all data structures on the heap start with the type field.

In a system using only tagged objects the machine registers and the fields of structures, e.g. the head and tail in the class `Cons`, contain pointers to tagged objects. This means that tagged objects are always referenced through an indirection.

The type test therefore requires not only a comparison but additionally a memory access for the indirect access to the tagged object. Boxing and unboxing are trivial casts with no runtime costs.

4.1.2 Tagged pointers

Tagged pointers are a data structure which fits into a word of the target architecture (typically 32 bits). The word is split into the tag (4 bits) and data field (28 bits). The tag contains the type informations. And the data field contains the value. If the value does not fit in the data, then additional storage is allocated and the data field contains a pointer to this additional storage.

Pointers Pointers are encoded into the 28 data bits of a tagged pointer combining two techniques. First, every heap node is aligned to word size. This ensures that the least significant two bits of a pointer are always zero, hence only 30 bits must be stored. Second, only 2^{30} bytes (1 GB) of the available virtual memory is used. These two techniques allow to represent a pointer in 28 bits. The overhead for tagging and untagging pointers is significant.

The class `Tagged` shown below is an implementation of tagged pointers. It has initialization (`Tagged` constructors), update (`set`), and access (`get`) methods.

```
class Tagged
{
private:
  static const mask=15;
  static const bits=4;
  unsigned int tagged;
  void checkTag(int tag) {
    Assert(tag >= 0 && tag <= mask);
  }
  void checkVal(int val) {
    Assert((val & (mask<<(32-bits))) == 0);
  }
  void checkPtr(void* ptr) {
    unsigned int val=(unsigned int) ptr;
    Assert((val&3)==0);
    Assert((val&(3<<30))==mallocBase);
  }
public:
  Tagged()                          { tagged = 0; }
  Tagged(void* ptr,int tag)         { set(ptr,tag); }
  Tagged(unsigned int val,int tag) { set(val,tag); }
```

```
  void set(void* ptr,int tag) {
    checkPtr(ptr);
    checkTag(tag);
    tagged = (((unsigned int)ptr)<<(bits-2)) | tag;
  }
  void set(unsigned int val,int tag) {
    checkTag(tag);
    checkVal(val);
    tagged = (val<<bits) | tag;
  }

  unsigned int* getRef() { return &tagged; }
  int           tag()    { return (tagged&mask); }
  unsigned int  getData(){ return tagged>>bits; }
  void*         getPtr() {
    return (void*)(mallocBase|((tagged>>(bits-2))&~3));
  }
};
```

Boxing The `set` methods implement boxing. They need one shift and one logical OR operation. For the zero tag[1] boxing reduces to a single shift. This optimization comes for free, when using an optimizing C++ compiler.

Unboxing The `get` methods implement unboxing. They need a single shift for non pointer values. Pointers require a shift by two and a AND operation to put zeros in the two least significant bits. Unboxing pointers can be compiled into a single shift if the tag has the bit pattern `xx00`.

On some architectures, where the heap segment cannot be allocated at the bottom of the memory, i.e. the two most significant bits of pointers are not zero, an additional operation to add the segment start is required when unboxing a pointer.

Type tests Type tests are done by masking out the bits of the tag and comparing this tag with the required tag. The zero tag is optimized by the C++ compiler, because the result of mask operation is already the negated result of the type test: false (zero in C++), if the tagged pointer has the zero tag and true (non-zero in C++) otherwise. When the result is immediately used in a conditional the C++ compiler can remove the otherwise required negation and normalization.

[1] The zero tag is used for representing references in the LVM tag scheme.

The `check` methods show how we implement method contracts in C++ as a mixture of comments and runtime checks: the `Assert` macro expands to the empty statement in the production system and to an explicit test with an error message in the development system.

The following code shows examples of a tagged pointer representation of list elements and integers.

```
class ConsData {
friend class Cons;
  Tagged head;
  Tagged tail;
};

class Cons : public Tagged {
public:
  Cons(Tagged h, Tagged t) : Tagged() {
    set(CONS,new ConsData(h,t));
  }
  Tagged getHead() { return (*(ConsData *) getPtr()).head; }
  Tagged getTail() { return (*(ConsData *) getPtr()).tail; }
};
```

The list element does not fit into the tagged pointer and requires to allocate additional data **class** `ConsData` for the head and tail fields.

```
class SmallInt : public Tagged {
public:
  Int(int i) : Tagged(INT,i) { }
  int getInt() { return getData(); }
};
```

The integer type implementation is a straight-forward refinement of the `Tagged` class with the limitation that only 28 bit integers can be stored.

The advantage of the tagged pointer scheme is the smaller memory footprint and a better performance especially for type tests. Tagged pointers can be stored in the fields of structures and in machine registers. For some values,e.g. small integers, everything fits into the tagged pointer and does not need additional memory. Compared to the tagged objects the type tests for tagged pointers require no memory access, because the type information is stored directly in the pointer.

The main drawback of tagged pointers is that they impose several restrictions. Pointers must fit into the remaining bits of the data field. For integers the implementation limits their range to $[-2^{27}, +2^{27} - 1]$. The efficient implementation of the arithmetic operators requires additional efforts [87].

Num	Bits	Tag	Data	Explanation
0	00**00**	`REF`	`Tagged`*	reference
4	01**00**	`WREF`	`Tagged`*	read-write reference
8	10**00**	`REF3`		reserved
12	11**00**	`REF4`		reserved
1	00**0**1	`TRANS`	`TransBody`*	transient
5	01**0**1	`UVAR`	space	optimized variable
9	10**0**1	`FUT`	space	optimized future
13	11**0**1	`GC`	`Tagged`*	garbage collection
2	**00**10	`CONS`	`ConsData`*	list element
3	0011	`REC`	`Structure`*	record or tuple
6	0110	`INT`	direct value	small integer
7	0111	`EXT`	`Extension`*	extension
10	1010	`VEXT`	`ExtBody`*	generic/virtual extension
11	1011	`FLOAT`	`Float`*	floating point value
14	1110			unused
15	1111	`LIT`	`Literal`*	literal

Figure 4.1: The LVM tag scheme.

Furthermore the number of available tag bits limits the number of possible representations for data-types. Instead of a fixed number of tag bits an implementation with varying numbers bits is possible.

Another variant of tagged pointers used in the LVM is an encoding where two bits are used for tagging and 30 bits are available for data. This allows to represent arbitrary pointers to word aligned data. It is for example used for the secondary tag to distinguish records and tuples.

4.1.3 The LVM tag scheme

Figure 4.1 shows the tag scheme of the LVM. The different types are explained in the following sections. Pointer values are marked with a star, e.g. Tagged* is a pointer to a tagged pointer.

The LVM uses a hybrid scheme of tagged pointers and tagged objects: as much as needed is encoded as tagged pointer (see Figure 4.1). One tag `EXT` is reserved for tagged objects which have secondary tags as listed in Figure 4.2. Another tag `VEXT` is reserved for virtual objects, which uses late binding instead of an explicit tag. These virtual objects are explained in Section 4.5.

The tagged pointer with all bits zero, the TaggedNULL, is reserved for special proposes, e.g. for signalling errors and exceptions.

Tag	Explanation
PROC	user-defined procedure
BUILTIN	built-in procedure
CLASS	class
OBJECT	object
THREAD	first class thread
CELL	cell
SPACE	first class space
PORT	port
CHUNK	chunk
ARRAY	array
DICT	dictionary
LOCK	lock

Figure 4.2: Secondary tags.

Integers Integers in the interval $[-2^{27}+1, +2^{27}-1]$ are represented directly in the data part of the tagged pointer using the `INT` tag. Operations on these integers are optimized such that no unboxing is needed.

Integers outside this interval are represented as extension with a secondary tag (see Section 4.5). These integers use an external package, namely the GNU multi precision library, version 2 to implement big integers and their operations.

Floats Floats are represented as tagged pointers using the `FLOAT` tag. They refer to a heap node containing a IEEE floating point with double precision representation of float values. These heap nodes are aligned to double word boundaries on the heap, because floating point arithmetic requires it. It is then possible that for every float a word is vasted on the heap for alignment.

4.1.4 Discussion

Gudeman [32] gives a good overview of techniques to represent values in dynamically typed languages and defines basic notions.

The LVM tag scheme is a compromise which optimizes the case that dereferencing and test for determination must be done at runtime. As explained above the zero tag (`REF` in the LVM tag scheme) allows for optimized type tests, boxing and unboxing operations.

We have also analyzed an alternative tag scheme, where no boxing and unboxing is needed for the `REF` tag. In this scheme all tags (0,4,8, and 12) with the

two least significant bits of zero are used as `REF` tags. For small benchmarks (tak, nrev) boxing and unboxing of `REF` tags, especially in conjunction with the allocation of transients in structures (see Section 4.2), are done so frequently that this optimization implies a performance difference of approximately ten percent. In other applications, e.g. the compiler or the scheduler, the difference is not significant.

The encoding of transients is such that if a tagged pointer is known to be no reference then the test if it is a transient is very cheap: t is a transient if `t&2==0`, which is similar to the test for a reference.

Another optimization is the encoding of the `CONS` tag for the representation of lists. The `CONS` tag is especially optimized, because lists are a convenient method for representing dynamic data structures and list iteration occurs frequently in applications. If it is known that a tagged pointer is no reference and no transient then the test if t is `CONS` is a single AND operation `t&13==0`.

Using tagged pointers has a drawback with respect to moving and copying values. A tagged pointer representing transients can never be copied, because the identity of a transient is represented by its location in memory. Therefore transients stored in registers and record fields must be handled carefully.

In the LVM transients are never stored in registers. Registers can only contain references to transients on the heap. This allows to copy and move nodes between registers without danger of occasionally creating copies of transients by moving tagged pointers. Furthermore this restriction of the implementation avoids the problem of unsafe variables known from the WAM [1].

Oz has integers of infinite precision and in the implementation a subset called small integers are represented efficiently. Lisp [100] optimize integers even more. They use two tags: 0000 for positive and 1111 for negative values. Therefore no tagging and untagging is needed and the overflow test simply checks if the result of an operation has a valid integer tag.

The tag scheme of the LVM is optimized for a compiler which does no aggressive static analysis to deduce static type information. Other tag schemes are needed for a highly optimizing compiler. For example if it does static analysis to detect determined and dereferenced values [12], then the optimization for references and variables would loose their prominent role.

Other languages which have static type systems or where the compiler can extract static type information can often avoid using run-time tags. Untagged values can then for example be stored directly in registers. Dynamic types are still needed, e.g. for doing garbage collection [6], but there overhead during the execution can be often avoided. Possible type systems and type inference for Oz are analyzed in [67]

In the LVM procedures are represented as unboxed values when they are used in first-class procedure applications, i.e. at compile time it is known that a application is always applied to the same procedure. Another example of a unboxed representation is the reference to self during the execution of methods, which is stored as unboxed value in a LVM register.

4.2 Transients

4.2.1 References

A reference in the LVM is a tagged pointer with the `REF` tag and a pointer to a tagged pointer.

```
class Ref : public Tagged {
public:
  Ref(Tagged *vPtr) : Tagged() { set(REF,vPtr); }
  Tagged *getRef() { return (Tagged *)getPtr(); }
};
Bool isRef(Tagged v) { return v.tag()==REF; }
```

4.2.2 Representation of Transients

The LVM supports three levels of representations for transients. At the bottom layer a highly optimized representation for storing variables in the fields of structures is implemented. The medium layer with a secondary tag is used to implement the built-in transient types, i.e. free variables, futures, and kinded variables. The medium layer uses a secondary tag to distinguish the different types of transients. For experiments new transient types can be added (dynamically) using a virtual layer, which uses late binding of a small number of interface functions.

```
enum TransType {
  FREE,
  FUTURE,
  KINDED_FD,
  KINDED_FS,
  KINDED_OR,
  ...
};
class SuspList {
  Thread*    thread;
  SuspList*  next;
};
```

```
class TransBody {
  TransType  type;
  SuspList*  suspList;
  Space*     home;
  TransBody(TransType t,Space* s)
  : type(t), suspList(0), home(s) {}
};
class Trans : public Tagged {
  Trans(TransBody* tb) : Tagged(tb,TRANS) {}
  TransBody* getBody() { return (TransBody*) getPtr(); }
};
Bool isTrans(Tagged v) { return v.tag() & 2 == 0; }
```

The standard representation of transients is a tagged pointer with the tag TRANS and a pointer to a transient node. A transient node (`TransBody`) is a labelled heap node which is labelled with the type, e.g. free, future, or kinded variable, a space and a suspension list. The suspension list contains threads which are suspended until the transient is bound.

4.2.3 Variables

A new variable is created with `newVar()`. `newVar()` returns a reference to the variable.

```
class FreeBody : public TransBody {
  FreeBody(Space* s) : TransBody(FREE,s) {}
};
Tagged newVar(Space* s) {
  return Ref(new Trans(new FreeBody(s)));
}
Bool isFree(Tagged t) {
  return t.tag()==TRANS &&
         ((Trans)t).getBody()->type==FREE;
}
```

Note that the memory needed for a new variable is the memory for the body and the memory for the tagged pointer. The reference does not use heap memory, because the C++ compiler can store it in registers and fields.

4.2.4 Futures

A future is a read-only view on a transient. Futures are implemented as transient nodes where the assignment operation blocks and suspends its thread until the protected transient is bound. A future of a transient is created with `futureOf`.

```
class FutureBody : public TransBody {
  FutureBody(Space* s) : TransBody(FUTURE,s) {}
};
Tagged futureOf(Tagged v) {
  Tagged tmp=deref(v);
  if (!isTrans(tmp)) return tmp;
  Space* s=((Trans)tmp).getBody().home;
  TransBody* tb=new FutureBody(s);
  Trans *t = new Trans(tb);
  addPropagator(tmp,Ref(t));
  return Ref(t);
}
```

This function first tests if the argument is a transient. If it is not the argument is directly returned. If the argument is a transient a new future is created and a propagator is installed to propagate the binding of the transient to the future.

4.2.5 By-need Futures

By-need futures are a specialization of futures. Additionally to the read-only aspect, is has an associated function. The by-need future is implicitly assigned to the result of the concurrent execution of the function, when its value is requested. A by-need future is *requested* when a threads blocks on it.

The LVM supports an optimized by-need future for the case that the function is a simple field selection of a record. When the by-need future is requested this field selection is tried without spawning a concurrent thread. This optimization is needed for the lazy loading of modules in Oz [22].

4.2.6 Binding

When a transient is bound the threads stored in the suspensions must be resumed and then the transient node is destructively updated to the new value.

```
void bind(Tagged v1, Tagged v2)
{
  Tagged* vPtr=derefPtr(v1);
  TransBody *tb=((Trans)v1).getBody();
  wakeup(tb->suspList);
  *vPtr = v2;
  free tb;
}
```

The memory used for the transient body can be safely released, because after the binding no reference to it exists any more.

In the LVM binding is more complicated, because hooks for handling spaces are needed, e.g. bind has to decide if a transient is local or not and eventually trail the binding (script model) or store the binding in a local binding frame (situated model).

4.2.7 Suspensions

Operations expecting a determined value suspend if they are applied to transients. Suspending means that the thread executing the operation is stopped and a suspension is hooked to the transient. A thread hooked onto a transient is restarted when this transient is bound. More than one thread can suspend on a single transient, i.e. a transient can be hooked with many threads. The structure to store the threads is called suspension list.

The primitive operation to suspend on the determination of a single value is `void wait(Tagged)`. It simply tests if its argument is determined, if not it blocks and suspends the current thread. When the transient is bound the thread is resumed and the `wait` operation is restarted and checks again if the new value is now determined.

A thread can suspend on more than one transient. The primitive operation for this case is `void waitOr(Tagged,Tagged)`. It suspends if both arguments are transients. In this case the thread is added to the suspension list of both transients.

In the LVM threads are never removed from a suspension list. This can lead to spurious wakeup and memory leak. If a thread suspends on more than one variable after a wakeup it potentially remains in the suspensions of the other variable.

A spurious wakeup occurs for example in the following code

```
spawn fn () =>
  (waitOr (x, y);
   wait z)
```

The thread starts running and suspends on `x` and `y`. When `x` is bound and `y` is not bound `waitOr` succeeds and the thread suspends on `z`. If `y` is now bound the thread is woken up without need and retries `wait z`, which suspends again.

An example of a memory leak is shown in the following example

```
spawn fn () =>
  (waitOr (x, y);
   wait _)
```

When `x` is bound the thread cannot be garbage collected, because a reference to it remains in the suspension list of `y`.

Both problems can be solved using a shared suspension structure in the suspension lists. This suspension structure has a reference to the suspended thread and is stored in both suspension lists. After a wakeup it can be marked, such that further wakeups are inhibited [87].

4.2.8 Usage patterns

The major design goal for the implementation of transients is that they are gracefully degrading wrt. to determined values. Every operation has to be prepared to handle transients, but if no transients are used no performance penalty should be payed. This is only possible in the current design of the LVM because all operations have to test the type of the node dynamically and transients are of a distinguished type.

Therefore special attention has been payed to an optimized implementation of references (`REF`) and transients (`UVAR`, `FUT`, `TRANS`). Every operation has to tests its arguments at least for the following cases:

reference If the argument is a reference it has to be dereferenced.

transient If the argument is a transient the operation has to suspend until the transient is bound.

Several variants of the dereference operation are useful. The simple `deref` function follows the reference chain until the end.

```
Tagged deref(Tagged v) {
  while (isRef(v)) {
    v = *((Ref)v).getRef();
  }
  return v;
}
```

This function is considered dangerous. Several hard to track bugs occured during the implementation of Mozart. The problem is that this function makes it easy to duplicate a transient by mistake. When the node returned by deref is a transient and it is stored into a register or field the transient is duplicated (see Figure 4.3).

A variant of this function is `safeDeref` which guarantees that a register node is returned, i.e. no transient is ever returned by `safeDeref`. The result of `safeDeref` can be stored safely into registers and fields.

```
Tagged safeDeref(Tagged v) {
   while (isRef(v)) {
     Tagged tmp = *((Ref)v).getRef();
     if (!isRef(tmp) && isTrans(tmp)) {
```

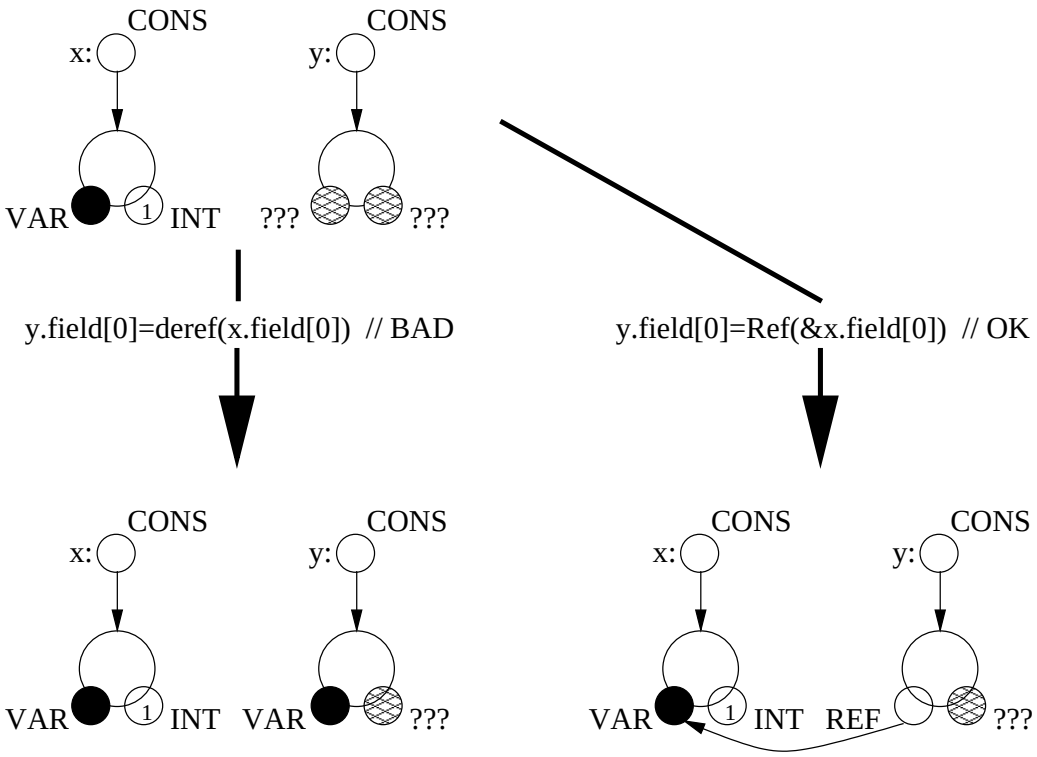


Figure 4.3: A possible dereference bug.

```
        return v;
      }
      v = tmp;
    }
    return v;
}
```

The last variant is `derefPtr`, which returns a pointer to the last tagged pointer, if the input is a reference. Furthermore it side-effects its call by reference argument and leaves the dereferenced value there.

```
Tagged *derefPtr(Tagged &v) {
  Tagged *ptr=0;
  while (isRef(v)) {
    ptr = ((Ref)v).getRef();
    v = *ptr;
  }
  return ptr;
}
```

In the following we present some implementation patterns for handling dynamically typed values and discuss their usage.

Optimistic pattern The optimistic pattern first tests if value is of the required type. Only if it is not dereferencing and the transient case are handled.

```
if !is<T>(v)
  v=safeDeref(v);
  if      isRef(v) suspend;
  else if !is<T>(v) error;
doit;
```

This pattern is very good if transients and reference chains occur infrequently. The LVM is optimized towards this case, because in the concurrent functional programming style transients and references occur only for the synchronization of concurrent activities.

Deref pattern The deref pattern ensures that the value is dereferenced before any type tests are performed.

```
v=safeDeref(v)
if      is<T>(v) doit
else if isRef(v) suspend
else             error
```

This pattern was used in the LVM before we had the insight that L can be seen as a functional language with extensions from logic programming rather than the

other way round. In the relational style of logic programming many references occur only, because return values are passed as references to variables used as containers for return values.

Optimized deref pattern The optimized deref pattern allows to slightly optimize the deref pattern such that the transient case is more efficient. An invariant of the LVM is that transients are never accessed directly but always through the indirection of a reference. This can be used to test the transient case only when dereferencing is needed.

```
if (isRef(v))
  v=safeDeref(v);
  if isRef(v) suspend;
if is<T>(v)   doit
else          error
```

Caller responsible pattern The caller responsible pattern only tests if the argument is of required type. No dereferencing and transient test is done. If the required type is no supplied an error is signalled to the caller. The caller is responsible for dereferencing and suspending in the case of a transient. The caller can ensure that dereferencing and the transient tests are performed before the application or it can do it lazily, i.e. after the operation has signalled an error.

Non-monotonic pattern The non-monotonic pattern is used for non-monotonic operations on transients, e.g. binding.

```
Tagged *vPtr=deref(v)
if isTrans(v) {
   *vPtr = ...
else error
```

When the dereferenced node is a transient the pointer `vPtr` refers to the transient which can be bound to a new value.

4.2.9 Unification

The store abstractions allow to implement the WAM-like instructions for an optimized unification. As an example we show the compilation of the following program

```
let Y=lvar() in
   unif (X, {a=Y, b=a});
   unif (Z, {a=Y});
   ...
end
```

into the following WAM-like code

```
get_record [a b], X
unify_variable Y
unify_constant a
get_record [a], Z
unify_value Y
```

The implementation of the instructions can be outlined as follows:

```
get_record(ar,R) {
  if (isTrans(R)) {
    mode = WRITE;
    node n = newRecord(ar);
    status = bind(R,n);
  } else {
    mode = READ;
    if (arity(R) != ar) status = FAIL;
  }
  sPointer = getArgRef(R);
}

unify_variable(R) {
  if (mode == READ) {
     R = *sPointer++;
  } else {
     R = newVar(currentSpace);
     *sPointer++ = R;
  }
}

unify_value(R) {
  if (mode == READ) {
     status = unify(R,*sPointer++);
  } else {
     *sPointer++ = R;
  }
}

unify_constant(c) {
  if (mode == READ) {
     node n = *sPointer++;
     if (isTrans(n)) {
       status = bind(n,c);
     } else if (n != c) {
       status = FAIL;
     }
```

```
  } else {
     *sPointer++ = c;
  }
}
```

Note, that the compiler knows the mapping of the arity from the features to the index and schedules the `unify` instructions for reading resp. writing the arguments in the correct order.

The `get_record` implementation shows that record constraints can be implemented as efficiently as prolog structures, if the arity is known at compile time.

4.2.10 Extending transients

In this section we explain a minimal and convenient interface to add new transient types to the LVM.

The interface for adding new transient types is defined by the class `ExtBody`.

```
class ExtBody : public TransBody {
public:
  ExtBody(Space *s) : TransBody(EXTVAR,s) {}

  virtual int           getIdV();

  virtual TransBody*    gcV();
  virtual void          gcRecurseV();
  virtual void          disposeV();

  virtual ReturnCode    bindV(Tagged*, Tagged);
  virtual ReturnCode    unifyV(Tagged*, Tagged*);
  virtual ReturnCode    addSuspV(Tagged*, Thread*);

  virtual Tagged        statusV();
};
```

Every transient has a type. The type is encoded as a unique id returned by the method `getIdV`. A new unique id may be obtained from a generator.

The methods `gcV`, `gcRecurseV`, and `disposeV` are used for memory management. In the stop and copy garbage collector `gcV` is used to copy a variable and `gcRecurseV` is used to update the reference after copying. These two methods are separated to allow the garbage collector to detect and break cycles.

The method `bindV` is called when the LVM wants to bind a transient to a value. This method succeeds if the binding is possible, fails if it is impossible, or suspends if the binding is neither possible nor impossible.

The LVM calls the method `unifyV` if the value is a transient. The method `unifyV` calls its own `bindV` method or the `bindV` method of the argument.

With this technique it is possible to incrementally add new types, where only the newer types need to know how to unify themselves with the transients of older types. The unification of two variables `unifyV(x, y)` calls `bindV(x, y)` if `x` "knows" how to unify with `y` else it calls `bindV(x, y)`.

When a thread needs to suspend until a transient is bound the method `addSuspV` is called to hook the thread to the function. For experimental purposes this function can fail, e.g. to enforce a programming style where only suspensions on futures are allowed.

The last function `statusV()` allows to distinguish the status of a transient. The status distinguishes variables, constraint variables, and futures.

As an example we show the by-need future implemented with the extension interface.

```
class ByNeed: public ExtBody {
private:
  Tagged fun;
public:
  ByNeed(Space* s,Tagged fun) : ExtBody(s), fun(fun) {}

  virtual int getIdV() { return TRANS_BY_NEED; }

  // memory management
  ExtBody* gcV()      { return new ByNeed(*this); }
  void gcRecurseV()   { if (fun) collect(fun,fun); }
  void disposeV(void) { delete this; }

  // always suspend binding
  ReturnCode bindV(Tagged* vPtr,Tagged t) {
     return SUSPEND
  };

  // allow unification with variables, otherwise suspend
  ReturnCode unifyV(Tagged* vPtr,Tagged* tPtr) {
    if (isFree(*tPtr))
      return ((Trans)*tPtr).getBody()->bindV(Ref(vPtr));
    return bindV(Ref(tPtr));
  }

  // kick the by need computation once
  ReturnCode addSuspV(Tagged*, Thread t) {
    if (fun) kick(fun);
    fun=0;
```

```
    suspList=new SuspList(t,suspList);
    return SUSPEND;
  }

  // a by need transient is a future
  Tagged statusV() { return atom("future"); }
};
Tagged byNeed(Tagged fun) {
  return Ref(new Trans(new ByNeed(current_space,fun)));
}
Bool isByNeed(Tagged t) {
   return t.tag()==TRANS &&
          ((Trans)t).getBody()->type==EXTVAR &&
          ((ExtBody*)((Trans)t).getBody())
           ->getIdV==TRANS_BY_NEED;
}
```

4.3 Records

In this section we explain the implementation of literals, records, and arities.

4.3.1 Literals

Literals are tagged pointers using the `LIT` tag. The pointer refers to a node with a secondary tag of an atom `ATOM` or a name `NAME`.

Atoms are allocated and stored in the atom table. The atom table[2] is another memory area beside registers and the heap. Atoms have fields for the ATOM tag, a print name, and the length of the print name.

New strings are internalized to atoms using hashing. The function `newAtom(char*)` finds or allocates an entry in the atom table, by calculating a hash value over all characters in the argument string. For every atom in a L source program this is done at compile or load time.

The hash value of atoms for selecting fields in records (see section 4.3.2) is done by the function `hashAtom(Tagged)`. It uses the fixed memory address of the atom in the atom table to efficiently generate a hash value.

Names are represented in the LVM either as named names or as free names. Named names can be created statically. The compiler can optimize the usage of named names similar to atoms.

[2] In other systems the atom table is also called symbol table.

Named name are further classified into unique names, copyable names, and optimized names. Unique names are **`true`**, **`false`** and **`()`** which are unique in every VM[3]. Optimized names are all other statically created names.

Named names are allocated and stored in the name table. The name table is similar to the atom table. Named name are labelled with a hash value, a print name, and their type, i.e. unique, copyable, or optimized.

Free names are dynamically created heap nodes which are labelled with a hash value and a space. The hash value is needed for the efficient representation of record arities (see Section 4.3.2) and can be extracted with the function **`int`** `hashName(Tagged)`.

Names are situated in spaces to be consistent with procedures and objects which must be situated[4]. The representation of a name thus contains a pointer to its space.

The implementation of free names needs two words. The first word represents the kind of literal, and the hash value. The second word contains the space.

The function `Tagged newName()` creates a new free name. It chooses a new hash value by incrementing a global counter, allocates an object of **`class`** `Name` on the heap, initializes it, and creates a tagged pointer to this object with the tag `LIT`.

A basic property distinguishing atoms and names is scalability. The number of atoms is fixed at compile time[5]. In contrast free names are created at run time and the number of names is unlimited. Therefore names are allocated from the heap and they are subject to garbage collection.

Hashing

For implementing record arities (see below) a hash function mapping a feature to a positive integer must be implemented for all types of features. For hashing on small integers their absolute value is used. Big integers are not hashed in the current implementation: all big integers are mapped to the same value.

The hash value of atoms is the unique address of their entry in the atom table. It is very efficient to use the address because it avoids the calculation of a hash value depending on the characters in the string.

A random hash value for names is computed when a name is created and it is stored in the data structure representing the name.

[3] Unique names are needed for serialization and pickling.

[4] Situated names are required to simplify the decision if procedures and objects must be copied when a space is cloned.

[5] Creating atoms dynamically is possible in full Oz, but it is depreciated. Strings or virtual strings can be used instead of dynamically created atoms.

Another implementation techniques for getting a hash value for names is the allocation of names in a separate memory area, where their address is fixed. Using this fixed address as hash value reduces the memory consumption of names dramatically: for a name generated in the top level space only one bit is needed. The garbage collector has to be adapted to use a non-copying, e.g. mark and sweep, collector for the new memory area instead of the implemented copying collector for the heap.

To efficiently implement the arity table it should be possible to order names. Using the fixed memory address a total order on names is trivially induced. Otherwise the random hash values must be all distinct. In the current implementation the distinctness is guaranteed by using a counter instead of a random number generator[6].

4.3.2 Record representations

The LVM uses four different representations for records with varying efficiency: list elements, tuples, simple records, and open records. The representation of a record is always normalized to its canonical representation. A record with the features `Head` and `Tail` is represented as list element. Records with consecutive numeric features from 1 to n are represented as tuples. Other determined records are represented as standard records. We use the name record also for the standard representation if it is clear from the context what we mean.

Tuples and (standard) records The representation of tuples and (standard) records are tagged pointers with tag `REC` where the pointer refers to a labelled heap node. The label of the heap nodes contains a secondary tag for distinguishing tuples and records. Furthermore the heap node is labelled with the tuple width resp. the record arity (see below). The heap node has a field for every feature.

```
class Structure {
  ArityOrWidth arity;
  Tagged field[n];
}
```

The field `ArityOrWidth` is a tagged pointer with an `RECORD` resp. `TUPLE` tag and the arity of the record resp. the width of the tuple.

The only reason to support an optimized representation for tuples in the LVM is that the dynamic creation of tuples is significantly (approximately a factor of five) faster than the dynamic creation of records, because no lookup in the arity table is needed.

[6]The implementation uses a 32 bit counter and effects related to counter overflow are not handled.

List elements List elements are represented as tagged pointers (`CONS`) where the pointer refers to an unlabelled heap node with one field for the head and one for the tail of the list.

The operations on list elements are the creation of new lists and the field selection. The class `Cons` implementing list elements was already shown in Section 4.1.

This representation saves a word, i.e. fifty percent, per list element, because the arity required for records is represented in the tag. The test for a non-empty list is more efficient then the test for a record with a certain label and arity, because only the tag bits must be checked and additionally the `CONS` tag is chosen such that the test for `CONS` needs only two native machine instructions (see Section 4.1.3).

A small issue with the list optimization is that the forwarding pointer has to be stored in the field of the head or the tail, which are also subject to garbage collection (see Section 4.6).

4.3.3 Arity

Efficient lookup The representation of an arity contains the hash table and the hash mask.

```
class KeyAndIndex {
  Tagged key;
  int index;
}

class Arity {
  Tagged featureList;
  int width;
  int hashMask;
  KeyAndIndex table[hashMask+1];
}
```

For an efficient access to the record width and the list of features both are stored in the arity. An alternate design would be to compute them from the content of the hash table on demand.

The size of the hash table is the next power of two which is greater than 1.5 times the width of the arity. The hash mask is the size of the table minus one. The size and hash mask are chosen such that the calculation of an index modulo the table size is a bitwise AND with the hash mask.

The table is created as hash table with the open addressing scheme from Knuth [52]. The table contains pairs of features and indices (KeyAndIndex). The features are stored as tagged pointer (Tagged key) and the indices (int index) are unique indices of the field at the corresponding feature. The lookup function returns a field index or -1 if the feature is not in the arity.

```
int Arity::lookup(Tagged feature)
{
  int i    = featureHash(feature) & hashMask;
  int step = (i%7)*2+1;
  while (1) {
    if (table[i].key == feature)
       return table[i].index;
    if (!table[i].key)
       return -1;
    i = (i + step) & hashMask;
  }
}
```

The size of the table is at least 1.5 times the width to have enough empty entries to make the member test also efficient for the case that a feature is not found.

The function `featureHash` computes a hash value for a literal or an integer.

An implementation of arities using bucket lists instead of the open addressing scheme would have fewer collisions, but the size required per arity would be larger. For buckets $3+3\times width+size$ words are needed in a linked list implementation. This is larger than $3+2\times size$ words for the open addressing scheme if we assume that the size is between $1.5\times width$ and $2\times width$.

Furthermore the access of the key resp. value of an entry requires one more memory access if the bucket list is represented as a linked list.

The arity table The arity table is a hash table using hashing with bucket lists to store all arities. The key used to access the arity table is the list of features of an arity. The hash value of a feature list is the sum of the hash values of its elements.

To compare a feature list with an entry of the arity table in linear time the list of features should be sorted. A problem when sorting a list of features are names, because they are not ordered in the Oz programming model. In the LVM names are ordered using the hash value. The order of names is not a total order, because the hash value is derived from a counter modulo the C++ word size. If consecutive names in the feature list have the same counter value it is therefore necessary to compare all there permutations[7].

A better heuristics to compute a hash value for a list of features could compute a hash value based only on the first k features. This optimization has no practical relevance, because dynamic record and hence arity creation occurs too infrequently. Furthermore dynamically created arities are in many case new, i.e. not

[7] With the technique of allocating names in a separate memory area the ordering problem of names disappears.

yet in the arity table, and the cost of their creation dominates the computation of the hash value.

4.3.4 The record interface

The basic operations on records are the creation of new records, the selection of fields, and pattern matching. The functions for creating and accessing tuples and records are summarized in the following table

`Structure* newTuple(int)`	allocate a new tuple
`Structure* newRecord(Arity*)`	allocate a new record
`Tagged Structure::setArg(int,Tagged)`	initialize a field
`Tagged Structure::makeRecord()`	create a tagged pointer
`Arity *Structure::getArity()`	access the arity
`int width(Tagged)`	access the width
`Arity* arity(Tagged)`	access the arity
`Tagged arg(Tagged,int)`	select a field by index
`Tagged field(Tagged,Tagged)`	select a field by feature
`Bool isTuple(Tagged)`	test if is tuple or record

Record creation Two kinds of record creations are distinguished static and dynamic creation.

Static record creation is used when the arity is known at compile time. In this case the arity is looked up and added to the arity table when the machine code is loaded. This is similar to internalizing string into the atom table.

Static record creation allocates the memory for the record structure on the heap, writes the label and arity into the record structure. The field values are written into the field array without hashing. This can be done because the arity is known at compile time and hence the index is also known at compile time. Of course the compiler and LVM must agree on mapping of feature to index.

We present one example of a dynamic record creation which allows to adjoin one feature and its field value to an existing record. This adjoin operation creates a new record, which has the same fields and field values as the original record, except that the new feature is added or that is value replaces an existing field.

- The new record has the same arity if the adjoined feature is already in its arity.
- If the adjoined feature is not yet in the record arity then a lookup in the arity table is performed with the new feature inserted into the feature list.

```
Tagged adjoinAt(Tagged rec, Tagged fea, Tagged val)
{
  // find arity
  Arity *newArity;
  Arity *oldArity = arity(rec);
  if (oldArity->lookup(fea)) {
    newArity = oldArity;
  } else {
    Tagged newList = insert(fea, oldArity->featureList);
    newArity = arityTable.find(newList);
  }

  // create record
  Structure *newRecord = newRecord(newArity);

  // copy fields
  Tagged l=oldArity->featureList;
  while (isCons(l)) {
    f = head(l);
    newRecord->setArg(newArity->lookup(f),field(rec,f));
    l = tail(l);
  }

  // new field
  newRecord->setArg(newArity->lookup(fea),val);
  return newRecord->normalize();
}
```

The cost of this adjoin operation has two parts: the test if the feature is already in the arity and eventually the dynamic lookup of the new arity. The first part is very efficient, because it uses the arity lookup function. The second parts requires hashing a feature list in the arity table and eventually creation of a new arity.

Additional optimized adjoin functions are provided by the implementation to adjoin more than one new feature at once and to create a new record from a list of features.

Field selection Selecting the field at feature of a record first calls the lookup function of the arity and if this is successful reads the corresponding entry of the field array.

```
Tagged arg(Tagged rec, int i) {
  Structure *str = getStructure(rec);
  Tagged val = str->field[i];
  return isDirectVariable(val) ? makeRef(&str->field[i]) : val;
}
```

```
Tagged field(Tagged rec, TaggedRef fea) {
  int i = arity(record)->lookup(fea);
  if (i<0) return 0;  // not found
  return arg(rec,i);
}
```

The value stored in the field array cannot be used unconditionally. The problem is the memory efficient representation of variables (see chapter Section 4.2). If a variable is allocated directly in the array and not on the heap a reference to this variable has to be returned by the field selection function. This means that an additional test is required for every field access.

To optimize the field selection inline caching [20, 108, 87] is used. The instruction `fieldCached` caches the triple of last feature, arity, and index. If the same feature is selected using the same arity then the index is directly taken from the cache.

```
Tagged fieldCached(Tagged rec, Tagged fea,
                   Arity *&cachedArity, Tagged &cachedFea,
                   int &cachedIndex)
{
  int i;
  if (arity(rec) == cachedArity && fea == cachedFeature) {
    i = cachedIndex;
  } else {
    i = arity(rec)->lookup(feature);
    cachedArity = arity(rec);
    cachedFeature = fea;
    cachedIndex = i;
  }
  if (i<0) return 0;  // not found
  return arg(rec,i);
}
```

Pattern matching: tests and indexing Pattern matching deals with efficiently decomposing records. The main techniques used to implement pattern matching are tests and indexing. A test compares a record with one pattern and indexing selects a matching pattern from a set of patterns.

The efficient compilation is based on the fact that the arity of the patterns are known at compile time. When the test resp. indexing code discovers that a pattern matches then the fields can be selected without hashing, because the compiler can precompute the lookup of the index.

The function `testRecord` checks if a node is a record with a given arity. Beside the type test the `testRecord` function reduces to exactly one comparison for the

arity. This is exactly the same number of comparisons as required for tuples. For tuples only the width is compared instead of the arity.

```
ReturnCode testRecord(Tagged rec, Arity *ar)
{
  Assert(ar != ArityEmpty);

loop:
  if (isRecord(rec)) {
    return arity(rec) == ar ? PROCEED : FAILED;
  }
  if (isCons(rec)) {
    return ArityCons == ar ? PROCEED : FAILED;
  }
  // deref and test for variable
  if (isRef(rec)) {
    rec = deref(rec);
    if (isTrans(rec)) return SUSPEND;
    goto loop;
  }

  return FAILED;
}
```

Indexing consists of two parts. The arity is hashed into the indexing table (using open addressing). The entries of the bucket list are then compared using the same comparison technique as `testRecord`.

4.3.5 Discussion

Flexible field selection Subtrees of records can be selected with decreasing efficiency numerically by an index if the feature and arity are statically known, with a statically known feature, or with a built-in procedure.

Selection by index is supported well on standard hardware and is therefore fast. The virtual machine has no instruction to support this select method for records, because in an emulator-based approach the performance difference to the selection by a statically known feature with inlining caching is negligible. The selection by index is useful in optimized built-in procedures, e.g. for selecting fields of records with known arities like tuples.

Arity The arity abstraction allows to separate the issues of field selection and of mapping of features to the fields. This provides a uniform model of the graph and an efficient mapping of the graph to standard hardware.

To support the equality test of arities an arity table is maintained in the runtime system which guarantees that every arity is represented exactly once.

Features The feature abstraction encapsulates two efficiency problems: the equality test of two features and the mapping of features to indices through arities.

Equality of features is implemented by the identity of nodes. Strings of characters are made unique with an atom table, which guarantees that two equal strings are mapped to the same atom. For names the runtime system maintains the invariant that they are never duplicated, which makes the equality test trivial.

The efficient mapping of features to indices is done through hashing. Useful hash functions are discussed in Section 4.3.2.

Representations Supporting three representations for determined records requires in the implementation additional code, because code dealing with records must be written such that all the representations are correctly handled. In cases where efficiency is not the major concern it is possible circumvent this problem by converting any record into the standard record representation and operate only on this representation.

Furthermore the dynamic creation of records has the overhead that the representation must be normalized. This basically means that list elements must be detected and turned into their optimized representation.

4.4 Feature constraints

Representation Feature constraints are implemented as transients with a field for the width attribute and a hash table for the fields attribute, which contains pairs of features and field values.

```
class OFVar : public TransBody {
private:
  int width;
  DT *dt;
public:
  OFVar(Space *s,int n)
  : TransBody(OFVAR,s) {
     width=-1;
     dt = DT::allocate(n);
  }
  ...
};
```

```
Tagged newOF(int n) {
  TransBody* tb=new OFVar(space);
  Trans *t = new Trans(tb);
  return Ref(t);
}
```

The hash table **class** DT, called dynamic table, contains an array of pairs DTE table[], the size of this array **int** size, and the number of elements in the array **int** num. The size of the array is a power of two to simplify open hashing. When the array is filled up to 75 percent the array size is doubled.

```
// dynamic table entry
class DTE {
   Tagged ident;
   Tagged value;
};
// dynamic table
class DT {
static:
   DT *allocate(int n);
private:
   int num;
   int size;
   DTE table[N];
};
```

Feature constraints The feature constraint is implemented such that first a test if the feature is already in the hash table is performed. If this is the case the old and new feature are unified.

```
ReturnCode OFVar::featureC
  (Tagged *vPtr, Tagged fea, Tagged val)
{
  Tagged oldVal=dt->get(fea);
  if (oldVal) return unify(oldVal,val);

  if (dt->isFull()) dt=DT::resize(dt);
  dt->add(fea,val);
  if (width==dt->num) return this->toRecord(vPtr);
  return PROCEED;
}
```

If a new feature is added then a test is performed if the table is up to 75 percent filled and must be resized. Then the feature with the corresponding value is added to the hash table.

Finally a test has to be performed if the number of elements is equal to the width attribute. In this case the open record is closed as shown in the method `toRecord`.

```
Return OFVar::toRecord(Tagged *vPtr)
{
  Tagged alist=dt->getArityList();
  Arity *arity=aritytable.find(alist);
  Structure *newrec = newRecord(arity);
  newrec->initArgs();
  return this->bindRecord(vPtr,newrec);
}
```

Closing an open record means to dynamically lookup resp. create an arity in the arity table. To simplify the implementation the fields of the new record are initialized with variables and the generic function to bind an open record to a closed record is called.

```
Return OFVar::bindRecord(Tagged *vPtr, Structure *str)
{
  PairList* pairs = dt->check(str);
  if (!pairs) return FAILED;

  Tagged saved=*vPtr;
  *vPtr = str->normalize());

  Return ret = unifyPairs(pairs);

  if (ret == PROCEED) {
    this->checkSuspension();
  } else {
    *vPtr = saved;
  }

  return ret;
}
```

Binding an open record to a closed record first checks if every feature of the open record is in the closed record. The `check` method returns the matching pairs of field values in the open and closed record.

```
Pairs *DT::check(Structure *str)
{
  Pairs *pairs=new PairList();

  for (int i=size; i--; ) {
    if (table[i].value) {
      Tagged val=str->field(table[i].ident);
```

```
      if (!val) {
        pairs->free();
        return 0;
      }
      pairs->addpair(val, table[i].value);
    }
  }
  return pairs;
}
```

If `check` was successful the second step in `bindRecord` is to bind the open record transient to the new record. This is necessary at this point to break a possible cycle when unifying the fields. Then corresponding fields in the pair list are unified. If all pairs are unified successful the suspensions are woken up.

Unification The merge method is the main part of the unification of two open records. It merges the features of one dynamic table into the other table.

```
PairList *DT::merge(DT* &dt)
{
  PairList *pairs=new PairList();

  for (int i=0; i<size; i++) {
    if (table[i].value) {
      Tagged val = dt->get(table[i].ident);
      if (val) {
        pairs->addpair(val, table[i].value);
      } else {
        if (dt->isFull()) dt=DT::resize(dt);
        dt->add(table[i].ident, table[i].value);
      }
    }
  }
  return pairs;
}
```

The `merge` method merges the features of the current table into its argument. Merging means that a feature is added if it is not yet in the table. The field values of features which are already in both tables are collected in a pair list for later unification with the `unifyPairs` function.

```
Return unifyPairs(PairList *pairs)
{
  PairList* p = pairs;
  TaggedRef t1, t2;
  Return ret = PROCEED;
  while (p->getpair(t1, t2)) {
```

```
    Assert(!p->isempty());
    ret = oz_unify(t1, t2);
    if (ret != PROCEED) break;
    p->nextpair();
  }
  pairs->free();
  return ret;
}
```

During the unification of two open records the following cases must be distinguished

- If both are local the largest dynamic table is used to merge in the smaller one.
- If one is local and the other is global, then the local variable is bound to the global one and the table of the global one is merged into the table of the local variable.
- If both are global then a copy of the largest table is created and the other table is merged into the copy.

4.5 Extensions

In this section we describe two methods to add new datatypes to the LVM. Both techniques use more memory for representing the type information and are slower for type tests. They are used for datatypes which need more memory anyway, e.g. arrays or for datatypes which are not frequently used.

4.5.1 Standard extensions

Standard extension nodes, have the head tag `EXT` and a secondary tag. Figure 4.4 lists the secondary tags.

The additional costs for these extension types are moderate. The type test has to test the primary tag, unbox the extension and then test the secondary tag. In the case of a successful test the already unboxed extension can be simply casted to the proper type for applying an operation. The costs of successful type tests is therefore amortized by the following operation on the datatype.

To create a new node a small overhead occurs only for storing the secondary tag.

The LVM knows all these types and can do some optimizations, e.g. inlining the methods of the corresponding implementation classes. For gaining efficiency this

BigInt	big integers
UserProc	user-defined procedures
Builtin	built-in procedures
Cell	cells
Space	first class reference to a spaces
Object	user-defined objects
Port	ports
Array	multiple cells
Dictionary	hash table of cells
Lock	lock
Class	user-defined class
Chunk	non-mutable object

Figure 4.4: Secondary tags.

is needed, but from a design point it would be nicer to have a small interface, as provided by the virtual extension explained below. For every data-type a performance analysis can be made and a design decision can be made on which level to support it.

Procedures and objects are further optimized in the LVM. In the bytecode top level procedures are represented directly. During a method application the unboxed representation of the current object, called self, is stored in a explicit machine registers the LVM for immediate access.

In the following we describe some of the extensions.

Big integers The tag `BigInt` allows to represent integers, which do not fit into the small integer representation describe in Section 4.1.2. In the LVM the GNU Multiple Precision Arithmetic Library (GMP) is used. The representation and the operations are taken from the library. Only the memory management is hooked to allocate big integers on the heap of the LVM using the free list technique (see Section 4.6).

Procedures Procedures are represented as built-in procedures or user defined procedures. Built-in procedures are native procedures typically written in C or C++. User defined procedures are written in L and compiled into LVM bytecode.

Objects and classes Objects and classes allow for an efficient representation of the object-oriented extension of Oz.

Spaces Spaces allow for first class references to computation spaces. First class space nodes are labelled with a reference to the internal representation of a computation space.

Cells Cells have a modifiable field for the content of the cell. To allow for the modification of the field only register nodes can be stored in it. The register node restriction guarantees that there are no reference from other nodes directly to the field. Cells are heap nodes which are labelled with their space.

A cell needs in addition to the secondary tag two words on heap. They are not optimized, because they are rarely used. Their primary usage is to serve as a theoretical foundation for objects. Objects are built into the LVM as optimized datatypes. The representation of objects is converted to a cell based representation to simplify the distribution protocols.

Ports Ports are represented in the same way as cells. The only difference is that the cell is not directly accessible. The update of the content is restricted to the port send operation, which creates puts another element on a stream associated with the port [49].

Locks Locks are another variant of cells, with a protocol to implement mutual exclusion.

Arrays Arrays are a straightforward extension of cells to multiple cells indexed by integers.

Dictionaries Dictionaries are more elaborate extension of multiple cells using a hash table mapping features (integers and literals) to cells. The hash table implementation of dictionaries is shared with the dynamic tables of the open records implementation.

4.5.2 Virtual extensions

The major difference between the standard extensions described before and virtual extensions is the usage of late binding for virtual extensions.

A virtual function interface defines all the hooks needed in the LVM to add new data-types. It allows to add arbitrary many new built-in data types in a modular way. It defines a small and simple interface for adding new types.

A drawback of virtual extensions is that a performance penalty has to be payed. Late binding implies that no inlining optimizations can be performed, i.e. calling a virtual method always needs a table access and a function call, which cannot be inlined.

```
class VExtension {
public:
  virtual ~VExtension();
  VExtension() {}
  virtual int          getIdV();
  virtual VExtension*  gcV();
  virtual void         gcRecurseV() {}
  virtual Tagged       printV(int = 10);
  virtual Tagged       typeV();
  virtual Bool         isChunkV()
  virtual Tagged       accessV(Tagged);
  virtual ReturnCode   eqV(Tagged);
  virtual Bool         marshalV(void *);
  Bool isLocal();
};
```

The virtual extension has virtual methods for typing (`getIdV`), garbage collection (`gcV` and `gcRecurseV`), inspecting (`printV, typeV`), field selection (`accessV`), equality test (`eqV`), and marshaling (`marshalV`). The minimal effort to add an extension is to implement the garbage collection and the `getIdV()` method.

The virtual methods are called by hooks in the memory management, printing, unification resp. equality test, the `select` operator, and the marshaling and unmarshaling routines of the LVM.

Two kinds of virtual extension are possible: situated and non-situated. Situated extensions are labelled with a space. They are handled correctly when spaces are cloned, by calling the garbage collection methods if needed. Non-situated extensions are never copied when spaces are cloned. New extensions need only to specify if they are situated or not.

To differentiate extensions a unique id is used. Every type of extension chooses a different id. New ids can be generated using a built-in id generator or ids can be pre-registered in LVM.

The type test for a virtual extension involves the following steps: test the primary tag, unbox the virtual extension, call the virtual function to get the id and compare it with the required id.

For operations on virtual extension the same argument as above holds: after type test the unboxed value can be casted to the required type without additional costs.

To create a new node the storage must be allocated, the method table must be initialized, if needed additional fields and labels must be initialized and finally

the extension must be boxed. The only difference between virtual extensions and standard extensions is the initialization of the method table instead of storing the secondary tag.

4.6 Memory Management

As usual for high-level languages L requires automatic memory management. The mapping of the language graph to the memory is done transparently, with no explicit requests to free or allocate memory at the language level.

4.6.1 Principles

The design goals of the memory management are similar to the design goals for most other parts of LVM: simplicity, flexibility, extensibility, and efficiency.

Simplicity Simplicity is required because the resources for our research project are limited and discovering elaborated memory management techniques was not in the focus of our research. The system should be stable and practically useful without too much effort for maintenance.

Flexibility For an explorative development, where new techniques and concepts are tried out and often replaced by new and better ideas the primitives have to be designed such that its easy to adapt them.

Extensibility The integration of new data-types must be simple.

Efficiency The performance of the system should, of course, not be degrading because of a bad memory management.

Generic principles of automatic memory management are

- Find garbage as soon as possible and make it available for reuse. The LVM supports free lists for data-structures which can be reused, e.g. the body of transients can be reused when transients are bound.
- Follow the principle of locality of memory access. The memory hierarchies of modern processors really pay off if the working set of the memory is not scattered all over the available memory. In the LVM we use therefor stack disciplines wherever possible.
- If none of the previous principles apply the graph representing the store has to be scanned and partitioned into the used and unused nodes. The unused nodes must then be made available.

In the LVM a stop and copy collector is used. All concurrent activities are first stopped, such that the memory management has exclusive access to the memory. The living parts of the graph are traversed and copied into new segments of the memory. Finally the old segments are released for future use.

A stop and copy collector has the advantage that it is simple because it has exclusive access to the memory. It behaves very well if the amount of living memory is small compared to the garbage. The memory is compacted automatically, which provides better locality. The node representation in the store can be very irregular, because their structure must be only known when a link is followed, e.g. no run-time type information is needed if a link is statically typed.

A stop and copy collector has the disadvantage that it is not concurrent and arbitrary delays of concurrent activities can occur during the execution of the collector. The collector needs (temporary) much more memory as required for the representation of the living graph.

4.6.2 Primitives

In this section we describe the primitives supplied in the LVM for maintaining memory. C++ supports to overwrite the memory management functions per class. In the LVM we use this to replace the operators **`new`** and **`delete`** with implementation to use heap resp. free list memory.

Heap memory

The heap memory is allocated from the operating system in chunks of memory called segments. The LVM maintains a chain of allocated segments. When a segment is full a new segment is allocated. The size of the segments is configurable.

When the garbage collection starts a new chain of segments is allocated and the living nodes are copied into the new chain. When the garbage collection is finished the old chain is released to the operating system.

The memory in a segment is allocated in a stack fashion starting from the topmost address down to the bottom. The LVM has two pointers for maintaining the available memory in a segment: the segments current top and the segment bottom. When new memory is requested the segments current top is decremented until the segment bottom is reached. When it is reached a new segment is allocated from the operating system.

Free list memory

A frequent case is that memory allocated for a structure can be released after a certain operation was performed, but that some of these structures can be

released too because they are not longer reachable in the graph. For this case the LVM supports free lists on top of the heap memory.

A typical case where a free list is useful are the body of transient values. When a transient is bound the body can be safely released. Using only this condition to release this memory is not sufficient: in the case of an unreachable transient in the graph its body should be released too. Therefore it is essential to combine the free lists with garbage collection.

Whenever a structure which was allocated from the heap can be safely released it is put into a free list. A request for a new structure then checks if memory is available from the free list. New memory is allocated from the heap when no memory is available from the free list.

Technically it is a useful optimization to have different free lists for different sizes of memory. This avoid problems with fragmentation and the release and allocation can be done efficiently in constant time.

Stack memory

Stack memory is used for maintaining the tasks on threads. The problem which arises here is that multiple concurrent threads exists and therefore multiple stacks must be maintained. Another complication is that the size of these stacks should be dynamically adaptable. Furthermore the conditions for deallocating the stack depends on the reachability of transient nodes in the graph.

All these problems are solved by allocating the stacks of threads on the heap. When a stack overflows a new stack is allocated and the old one it is copied to the new one and released to the free list.

4.6.3 The implementation of the garbage collector

The garbage collector of the LVM starts traversing the graph of the store from the roots. The roots for garbage collection are the threads in the runnable queue and some global data-structures, e.g. global properties, the default exception handler, etc.

For every living reference to a node the garbage collector performs the following steps

copy The node is copied to the new chain and the reference to the node is updated.

mark The original node is marked and a forward pointer is stored there. When the node is visited again this is detected and the forward pointer is used to update the reference to the new location.

collect The additional entry points reachable from the just collected node are collected after copying and marking. The order of the mark and collect steps is essential to avoid infinite recursion in the case of cyclic structures.

To avoid deep recursion on the runtime stack an explicit stack, the update stack, is used to maintain the not yet collected entry points. The update stack contains the type of the node and a pointer to the node. The LVM use the tagged pointer technique for the entries on the update stack.

The LVM does not use pointer reversal [25] and Cheney's breadth-first [17] techniques to make the update stack obsolete. These techniques can be adapted easily for the LVM.

Because many structures and nodes on the heap are implemented as C++ classes it is straight-forward to implement the collection algorithm with the following methods

```
class Node {
  Bool gcTestMark()      test if node is marked
  Node* gcGetForward()   get the forward pointer if node is marked
  void gcPutMark()       mark the node
  void gcPutForward()    put the forward pointer
  Node* gcCopy()         copy an unmarked node
  void gcCollect()       collect the entry points
... };
```

The implementation of the methods maintaining the mark and the forward pointer is trivial. E.g. for tagged nodes one tag is reserved as garbage collection mark and the data part is used as forward pointer.

The `gcCopy` method can simply use the C++ copy constructor, because the memory management constructor **`new`** is overwritten to use the heap of the LVM:

```
Node* Node::gcCopy() { return new Node(*this); }
```

For nodes with an explicit tag the copy constructor depends on the tag:

```
TransBody* TransBody::gcCopy() {
  switch (this.type) {
  case FREE:   return new FreeBody(*(FreeBody*)this);
  case FUTURE: return new FutureBody(*(FutureBody*)this);
  ...
  }
}
```

The mark, forward, and copy methods are usually combined into one method `Node *gc()`, which returns the forward pointer if the node is already collected, else the node is copied and the new node is pushed onto the update stack for the further collection of entry points.

The `gcCollect` method then simply updates its fields using the `gc` method.

```
void Node::gcCollect() {
  this.n1 = this.n1->gc();
  this.n2 = this.n2->gc();
  ...
}
```

The main gc procedure first copies the roots, and pushes additional entry points to the update stack. Then it loops until the update stack is empty to collect all entries.

```
void gcMain() {
  runnable=runnable->gc();
  ...

  while (!updateStack.isEmpty()) {
    GcNode n = updateStack.pop();
    switch (n.tag()) {
    case GC_TRANS: ((Trans*)n.getPtr())->gcCollect();
    case GC_THREAD: ((Thread*)n.getPtr())->gcCollect();
    ...
    }
  }
}
```

4.6.4 Optimized transients

To avoid that optimized transients allocated in fields of records are copied into the heap, the collection of references to optimized transients is delayed until the end of the garbage collection. When an optimized transient is found during the collection of a record it is directly copied with this record. The collection of transients found through a reference node is delayed because it may be that this transient is allocated in a field of some record reached later in the garbage collection.

The delayed updates are pushed onto an additional stack, called the var fix stack. When the regular update stack is empty the var fix stack is processed. If the reference is found to be marked as already collected, then the variable was in a field and the forward pointer is used for the update. If the reference is not marked the variable is copied to the new chain.

4.6.5 Liveness analysis

The X registers are allocated per thread, but in the implementation only one shared register array is used. When a thread is preempted or suspended the living

X registers are saved in the thread and when the thread is scheduled again they are restored. The number of X registers saved and restored is only an approximation of the exact number of living X registers, i.e. the compiler calculates the maximal number of registers used per procedure.

During the garbage collection an exact analysis of the liveness of the X registers is performed to avoid that unreachable data in X registers is collected.

The base of the liveness analysis is the control flow graph of the bytecode. The control flow graph of a code segment has a node for every instruction in the code segment. The graph has a directed link from node A to node B if it is possible that the instruction B is executed directly after the instruction A. The control flow graph has no cycles.

The liveness analysis scans the control flow graph starting from the instruction which is executed when the thread is rescheduled. It finds out which X registers are never used. The algorithm works such that all possible paths in the control flow graph are examined.

For every path in the data flow graph the liveness maintains a map of the current register usage. The status of a register can be

written The first usage of the register in the path was an assignment operation. In this case the register can be assumed to be dead.

read The first usage of the register was an access operation. In this case the register must be saved.

unknown The register is neither assigned nor accessed. This is the initial status of every register.

When two paths join at an instruction the maps of these paths have to be joined. For every register the state of the two maps are compared and the result status is computed as follows

- If both stati are the same the result status is also the same.
- If one status is unknown the other status is the result status.
- If one status is written and the other status is read the result status is read.

Two invariants of the LVM bytecode allow for an efficient implementation of liveness:

- Branches are always forward branches to higher addresses. No backward branches are allowed. This makes it easy to ensure that no instruction is scanned more than once.

- For two paths starting at the same instruction no register is marked as written on one path and marked as read on the other path. This allows to maintain one status map for the whole liveness analysis, because two paths never disagree on the status of a register.

Besides a register usage map the algorithm maintains an ordered list of addresses, the todo list, and the address of the currently scanned instruction. The todo list contains a list of increasing addresses.

For the current instruction one or more of the following actions are performed:

write If the instruction writes into a register and its status is unknown, the status is changed to written.

read If the instruction reads a register and its status is unknown, the status is changed to read.

branch If the instruction can branch the target address of the branch is inserted into the todo list.

The main procedure for the liveness analysis has two loops: the outer loop iterates over the ordered todo list and the inner loop iterates over a sequence of instructions until a break point is reached. Break points are instructions after which no assumption about the liveness of X registers can be made, e.g. the `return` instruction at end of a procedure or a non-inlined application.

Addresses on the todo list are skipped if they are less or equal than the current address, because its guaranteed that the instruction at this address was already scanned.

```
RegMap liveness(ByteCode *startAddr)
{
  RegMap    regMap[] = UNKNOWN;
  Todo      todo     = nil;
  ByteCode* PC       = 0;

  todo.add(startAddr);

outerloop:
  while (!todo.isEmpty()) {
    ByteCode *newPC = todo.pop();
    // already scanned?
    if (newPC <= PC) goto outerloop;
    PC=newPC;

  innerloop:
```

```
    while (true) {
      switch (getOP(PC)) {
        case MOVEXX(i,j):
          if (regMap[i] == UNKNOWN)
             regMap[i] = READ;
          if (regMap[j] == UNKNOWN)
             regMap[i] = WRITE;
          break;
        case TEST*(...,addr1,addr2):
          ...
          todo.add(addr1);
          todo.add(addr2);
          break;
        case RETURN:
          goto outerloop;
        ...
      }
      PC=PC+1;
      goto innerloop;
    }
  }
  return regMap;
}
```

Y registers The liveness analysis is only performed for X registers, because one array of X registers is saved per thread. This means for every thread found during a garbage collection the liveness analysis has to be performed once.

No liveness analysis is performed for Y registers, because Y registers are usually allocated per procedure application, i.e. per task. A liveness analysis for Y registers would be too expensive, because the number of tasks is under the assumption that in the average ten tasks per thread are active an order of magnitude larger than the number of threads. Furthermore non-inlined procedure applications are no longer break-points for stopping the liveness analysis of Y registers.

4.6.6 Lists

Lists are frequently used data structures. With the generic collection algorithm outlined above an entry is pushed onto the update stack and popped immediately afterwards for every list element.

The solution is to use an iterative algorithm for collecting list elements. During the collection phase the head is copied and eventually pushed onto the update

stack as usual, but the collection directly continues with the copying and collection of the tail while it is a list element.

The memory efficient representation of list elements has the consequence that the forward pointer for the list element and its first element are shared. Coincidentially this does no harm, because both forward pointers are equal.

Chapter 5

Conclusion

5.1 Summary

We have presented an efficient mapping of a concurrent functional programming language L with logic variables, futures, record constraints, and deep guards to an imperative virtual machine LVM.

The virtual machine is constructed using a modular and open design. The modules correspond closely to the language primitives and can be to a large extend developed and explained independently. The open design allows for simplified modifications and for an easy integration of extensions into the LVM.

The implementation of data structures uses a layered architecture with a highly optimized tagged pointer scheme at the bottom, a medium level tagged object scheme for many datatypes, and an extensible and open layer based on late binding for experiments and easy integration of new data types.

We have shown that many well known ideas from different research communities can be integrated into a single system. For example first-class procedures, logic variables, deep guards, concurrency, records and feature constraints, and statefull programming could be smoothly combined in the LVM.

Personal remarks Many parts of the implementation have an extremely minor impact on the performance of the systems. If these parts can be integrated in an orthogonal manner, then the lesson learned is: don't invest too much time in clever algorithms and design, but simply do it in the naive way quickly.

A lot of time during the work on the LVM went into the engineering of a stable and useful system for users. Typically bugs found by users were corrected in less than a day. New features could often be implemented before their specification was finished due to the flexible design.

The development of the LVM was highly explorative. New ideas for language primitives came up frequently. As implementors we are eager to incorporate them quickly to find out if they can be efficiently implemented and what are their costs.

After some time of programming experience these ideas were typically refined and sometimes replaced by more powerful concepts. One example of such a development are threads. At the beginning we started with fine-grained concurrency and we tuned and optimized the LVM to support them very well. Then we saw that this fine-grained concurrency is not really wanted and needed. After an intermezzo based on jobs, which allowed for a semi-grained concurrency, we arrived at the thread model.

Performing all these frequent changes throughout the LVM was a challenging task. A major effort was to identify orthogonal pieces and to design interfaces between them, such that further changes only effect small parts of the whole system.

5.2 Engineering considerations

In this section we summarize some of our engineering experiences with respect to the implementation language and hardware platforms.

5.2.1 C++ vs. C as implementation language

At the beginning of the project C++ was chosen as implementation language. The main reason was that C++ has a lot of features which simplified the first implementation and it allowed us to make many experiments.

Encapsulation of data structures using classes and methods was useful because the implementation could be changed frequently, without too much influence on the rest of the system.

During the development performance became an issue and it turned out that because of the number of features supported by C++ it was difficult to predict the performance directly from the source code.

One useful feature for high-performance implementations are inlined functions. The compiler usually replaces the call of such a function during compile time by its definition. This optimization avoids a function call and typically creates larger basic blocks for other optimizations. The drawback of inlining is that it is not a language requirement of C++ and the compiler can also decide not to do it. This means that as an implementor one has to check what the compiler has done.

C does not support inline functions and the basic concept for achieving a similar result is to use macros. Macros are not as safe as inline functions, e.g. the compiler does not check the types of arguments, but there expansion is predictable and does not depend on the compiler. A major trap of macros is that the programmer must be careful to that arguments are not evaluated twice.

Another source of optimization are virtual functions: in our implementation we avoid virtual functions in many classes and implemented the dispatch to different implementation in subclasses using explicit tags. Together with inlining this was faster and less memory was needed per object. Only a small number of bits are required for the tag bits to distinguish different subtypes and the memory for the pointer to a virtual function table is saved.

In the current implementation we only use a small amount of features which are not available in C. For optimizing the emulator it would be helpful to rewrite it in C, because the optimizations done by the C compiler are better predictable. The shear amount of features in C++ makes it extremely difficult to predict if the compiler can optimize them. An example to illustrate this: recently we found out that GCC 2.7.2 cannot optimize conditions if the second condition of a conjunction (`&&`) contains a call to an inline function.

Another problem which occurs with C++ is that the size of header files is huge and the dependency among them becomes quite complex for such a large project as the LVM.

5.2.2 The role of the target platform

Implementing the emulator in C++ makes it easy to port it to different platforms, because compilers for C++ are available on every platform and compilers are almost compatible. The main effort when porting the Mozart system to a new platform are the operating system dependent functions.

Porting the OS specific parts is not the only problem. A second problem is that different hardware architectures require different kinds of optimizations at the level of the C++ source code to gain maximal efficiency. This problem is not specific to the implementation of virtual machines, but the performance of the LVM depends to a large extended on the exact understanding of the mapping to real hardware.

Dispatch One performance bottleneck is the threaded-code interpretation, which needs to dispatch to the next instruction. RISC architectures have one or more delay slots which can be executed in parallel with a jump. To use this slot the increment of the program counter must be decoupled from the jump:

```
#define DISPATCH_OPT(n)
   void *lbl = *(PC+n);
   PC += n;
   goto *lbl;
```

This code allows the compiler to increment the PC in parallel to the jump, by using the delay slot for the increment.

The following naive dispatch

```
#define DISPATCH_NAIVE(n)
   PC += n;
   goto *PC;
```

will stall the jump until the PC is incremented and if no other instructions could be scheduled the delay slot remains empty.

When the emulator was ported to the INTEL x86 architectures we noticed that the DISPATCH_OPT did not generate optimal code. For this architecture the naive DISPATCH was better, because x86 processors have fewer registers and it has an indirect jump instruction which can read an address directly from memory.

Machine registers A typical difference between CISC and RISC architectures is the number of available assembler registers and the addressing modes. RISC have many general purpose registers and CISC have few and some special purpose registers. RISC only supports a limited number of addressing modes which are typically based on registers. CISC supports a rich number of addressing modes.

As an example we analyzed the usage of the X registers in the LVM. The LVM has a single set of the global X registers at a fixed address in memory.

For RISC architectures it is good to load this address into a local machine register of the workers main procedure, because this address is frequently used and RISC processors need two instructions to load an address.

CISC architectures support the direct addressing of every memory location and it is better to use this direct addressing mode instead of storing the address in one of the few available registers.

As an example accessing X[i] needs two RISC instructions if X is not in a register compared to one if it is. On CISC processors the situation is swapped. CISC needs two instructions if X is in a temporary variable and only one if the fixed address is used.

5.3 Future work

5.3.1 Improve compilation

A disadvantage wrt. a high-performance implementation is that every data structure is dynamically typed and type tests, unbox, and box operations have to be performed frequently at runtime. If more type information would be available at compile-time a better interface between the compiler and the LVM allows to use unboxed representations for values, e.g. storing floating point values in float registers for numeric calculations.

Another aspect of this problem are references and transients. It would be useful if the compiler could derive information about reference chains and determination of values. Implicit dereference operations and synchronization code all over the LVM could then be replaced by explicit bytecode instructions.

5.3.2 Reuse existing technology

The Mozart system is self contained, which means that it has only few dependencies on third-party tools and software. The development model was flexible, because only few people had to coordinate their changes and no legacy problems occur. A disadvantage of such a model is that new techniques, libraries, and tools developed in other projects could not be easily reused.

Often it is possible to design and implement interfaces to third-party software, e.g. for GUI programming we use an interface to Tcl/Tk. Typically such an interface is not trivial and requires a lot of effort. Sometimes the wheel has to be invented again for designing useful libraries, e.g. for OS services like sockets, pipes, and files, database interfaces, etc.

For the future of Oz/Mozart I think it would be useful to investigate the possibility to add our ideas to existing systems and to reuse their technology and infrastructure.

One promising candidate is Java and the Java Virtual Machine as a platform for compiling Oz programs. The JVM is nowadays available on virtually all platforms, including coffee machines and libraries and API for all kinds of applications exist. It would be necessary to analyze the limitations of the JVM and how resp. if it can be a target language for Oz.

Another option is to incorporate the Oz ideas into functional languages, like Objective Caml, Standard ML, and Scheme. These languages are closer to the language model of Oz than the imperative language Java. These languages have not the commercial impact and the library base of Java, but they are well-known in the academic community. Another advantage of this direction is that software

developed in academic institutions is typically available freely and can thus be adapted to the specific needs.

5.3.3 Functional core

The original design of the LVM was based on the relational model inherited from logic programming. In many parts the current design described in this thesis is based on the functional programming model. In the design some parts are left over from the relational model. The LVM has only procedures and return parameters are passed using logic variables as call-by-reference parameters. In this design logic variables are at the core of the system.

An alternate design could be a VM based on functions, where logic variables and other transient types can be introduced as fully orthogonal primitives.

5.3.4 Distribution

The LVM is implemented as a single-threaded operating system process with a single worker for the execution of threads. It is useful to investigate how to take advantage of the emerging multi-processor technology.

The directions currently investigated are parallelism and distribution. Parallelism [80] starts with the idea of a single LVM and investigates which synchronization is needed to allow for multiple workers in a single address space. Distribution [39] starts with multiple LVMs and analyzes how to give the illusion of a transparent distributed store, based on distributed access structures to nodes in the store and protocols to implement graph rewriting steps.

It seems that the distributed approach dominates parallelism. Distribution allows also to take advantage of multiple processors by starting two LVM on one computer and it allows also to explore the computation power of computer clusters. It seems that the amount of communication necessary for interesting parallel applications is small compared to the amount of computation. In this case a parallel implementation has no advantage over a distributed implementation.

Bibliography

[1] Hassan Aït-Kaci. *Warren's Abstract Machine - A Tutorial Reconstruction.* The MIT Press, 1991.

[2] Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[3] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3 and 4):195–234, August 1993.

[4] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, January 1994.

[5] Andrew W. Appel. A runtime system. Technical Report CS-TR-220-89, Princeton University, May 1989.

[6] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 19(7):703–705, July 1989.

[7] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[8] Joe L. Armstrong, Bjarne O. Däcker, Robert Virding, and Mike C. Williams. Implementing a functional language for highly parallel real time applications. In *Software Engineering for Telecommunication Systems and Services*, March 1992.

[9] Joe L. Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Progamming in ERLANG (2nd Edition).* Prentice Hall, 1996.

[10] Rolf Backofen and Ralf Treinen. How to win a game with features. *Information and Computation*, 142(1):76–101, April 1998.

[11] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[12] Peter A. Bigot and Saumya K. Debray. A simple approach to supporting untagged objects in dynamically typed languages. *The Journal of Logic Programming*, 32(1):25–47, July 1997.

[13] Peter A. Bigot and Saumya K. Debray. Return value placement and tail call optimization in high level languages. *The Journal of Logic Programming*, 38(1):1–29, January 1999.

[14] Per Brand. A decision graph algorithm for ccp languages. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming*, pages 433–447, Tokyo, Japan, June 1995. The MIT Press.

[15] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[16] *International Standard ISO/IEC 14882:1998, Programming Language - C++*, 1998.

[17] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[18] Alain Colmerauer. Prolog and infinite trees. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 231–251. Academic Press, New York, 1982.

[19] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, pages 70–90, July 1990.

[20] L. Peter Deutsch and Alan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In 11^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–302. ACM Press, January 1984.

[21] Robert B. K. Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, June 1975.

[22] Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Programming Systems Lab, DFKI and Universität des Saarlandes, 1998. DRAFT.

[23] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[24] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, Austria, 1996.

[25] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *CACM*, 12:611–612, 1969.

[26] Alessandro Forin. Futures. In Lee [58], chapter 9.

[27] Torkel Franzén, Seif Haridi, and Sverker Janson. An overview of the Andorra Kernel Language. In *Proceedings of the 2nd Workshop on Extensions to Logic Programming*. Springer-Verlag, 1992.

[28] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[29] John B. Goodenough. Structured exception handling. In *2nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 204–224, Palo Alto, California, January 1975.

[30] James Gosling, Bill Joy, and Guy Steele. *The Java language specification*. Addison-Wesley, 1997.

[31] Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1997.

[32] David Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, October 1993.

[33] Gopal Gupta and Bharat Jayaraman. Analysis of Or-parallel execution models. *ACM Transactions on Programming Languages and Systems*, 15(4):659–680, 1993.

[34] Robert H. Halstaed. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[35] Seif Haridi. A tutorial of Oz 2.0, 1997. Available from the web at `http://www.sics.se/~seif/oz.html`.

[36] Seif Haridi. Tutorial of Oz, 1999. `http://www.mozart-oz.org/`.

[37] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In David H. D. Warren and Peter Szeredi, editors, *Logic Programming, Proceedings of the 7th International Conference*, pages 31–48, Cambridge, MA, June 1990. The MIT Press.

[38] Seif Haridi and Dan Sahlin. Efficient implementation of unification of cyclic structures. In J. A. Campbell, editor, *Implementations of Prolog*, pages 234–249. John Wiley & Sons, Ltd., 1984.

[39] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 1998.

[40] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[41] Bogumil Hausman. Turbo Erlang: Approaching the speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[42] Martin Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *the Kluwer international series in engineering and computer science*. Kluwer Academic Press, October 1997.

[43] IEEE. *9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition] Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application: Program Interface (API)*, 1996.

[44] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. Technical report, Department of Computer Science, Monash University, Australia, June 1986.

[45] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In 14^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–119. ACM Press, 1987.

[46] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–582, May-July 1994.

[47] Sverker Janson. *AKL — A Multiparadigm Programming Language*. PhD thesis, Computer Science Department, Uppsala University, Sweden, 1994.

[48] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In Saraswat and Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*. The MIT Press, 1991. Available as SICS RR R91:08.

[49] Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects in concurrent logic programming. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.

[50] Guy L. Steele Jr. *Common Lisp: the language (2nd ed)*. Digital Press, 1990.

[51] Richard Kelsey, William Clinger, and Jonathan Rees. *Revised^5 Report on the Algorithmic Language Scheme*, 1998.

[52] Donald Knuth. *The Art of Computer Programming: Sorting and Searching (Vol 3, 2nd Ed)*. Addison-Wesley, 1998.

[53] Andrew R. Koenig and Bjarne Stroustrup. Exception handling for C++ (revised). In *Proc USENIX C++ Conference*, April 1990. Also in *The Evolution of C++: Language Design in the Marketplace of Ideas*, Journal of Object Oriented Programming, 3(2), July/Aug 1990.

[54] Peter M. Kogge. An architectural trail to threaded-code systems. *Computer*, pages 22–32, March 1982.

[55] Robert A. Kowalski. Predicate logic as a programming language. In *IFIP 74*, pages 569–574, October 1974.

[56] Robert A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[57] R. Greg Lavender and Dennis G. Kafum. A polymorphic future and first-class function type for concurrent object-oriented programming. The University of Texas at Austin, 1992.

[58] Peter Lee, editor. *Topics in advanced language implementation.* The MIT Press, 1991.

[59] Thomas Lindgren, Per Mildner, and Johan Bevemyr. On Taylor's scheme for unbound variables. Technical Report UPMAIL TR No. 116, Computing Science Department, Uppsala University, October 1995.

[60] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.

[61] Peter S. Magnusson, Fredrik Dahlgren, Hkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenstrm, and Bengt Werner. Simics/sun4m: A virtual workstation. In *Usenix Annual Technical Conference*, New Orleans, Lousiana, June 1998.

[62] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An Abstract Machine for Oz. Research Report RR-95-08, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D66123 Saarbrücken, Germany, June 1995. Also in: *Proceedings of PLILP'95*, Springer-Verlag, LNCS, Utrecht, The Netherlands.

[63] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[64] Johan Montelius. *Exploiting Fine-grain Parallism in Concurent Constraint Languages*. PhD thesis, Computer Science Department, Uppsala University, Sweden, 1997.

[65] Johan Montelius and Khayri A. M. Ali. An And/Or-parallel implementation of AKL. *New Generation Computing, Special issue on the Workshop on Parallel Logic Programming*, 14(1), 1996.

[66] The Mozart Programming System. `http://www.mozart-oz.org/`, 1998.

[67] Martin Müller. *Set-based Failure Diagnosis for Concurrent Constraint Programming*. Dissertation, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, January 1998.

[68] Tobias Müller. Solving set partitioning problems with constraint programming. In *Proceedings of the Sixth International Conference on the Practical Application of Prolog and the Forth International Conference on the Practical Application of Constraint Technology*, pages 313–332, London, UK, March 1998. The Practical Application Company Ltd.

[69] Tobias Müller and Martin Müller. Finite set constraints in Oz. In *13. Workshop Logische Programmierung*, Technische Universität München, 17–19 September 1997.

[70] Tobias Müller and Jörg Würtz. Extending a concurrent constraint language by propagators. In Jan Małuszyński, editor, *Proceedings of the International Logic Programming Symposium*, pages 149–163. The MIT Press, 1997.

[71] Richard A. O'Keefe. *The Craft of Prolog*. The MIT Press, 1990.

[72] The DFKI Oz Programming System. `http://www.ps.uni-sb.de/oz1/`, 1995.

[73] The DFKI Oz Programming System (version 2). Available from the web at `http://www.ps.uni-sb.de/oz2/`, 1997.

[74] Larry Paulson. *ML for the Working Programmer (Second Edition)*. Cambridge University Press, 1996.

[75] John Peterson and Kevin Hammond. *Report on the Programming Language Haskell, Version 1.4*, April 1997.

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

[77] Andreas Podelski and Gert Smolka. Situated simplification. *Theoretical Computer Science*, 173:209–233, February 1997.

[78] Andreas Podelski and Peter Van Roy. The beauty and the beast algorithm: Quasi-linear incremental tests of entailment and disentailment over trees. In *Proceedings of the International Logic Programming Symposium*, pages 359 – 374, Ithaca, New York, November 1994. The MIT Press.

[79] Andreas Podelski and Peter Van Roy. A detailed algorithm testing guards over feature trees. In Manfred Meyer, editor, *Constraint Processing, Selected Papers*, volume 923 of *Lecture Notes in Computer Science*, pages 11–38. Springer, 1995.

[80] Kostja Popov. A parallel abstract machine for the thread-based concurrent language Oz. In Inês de Castro Dutra, Vítor Santos Costa, Fernando Silva, Enrico Pontelli, and Gopal Gupta, editors, *Workshop on Parallism and Implementation Technology for (Constraint) Logic Programming Languages*, 1997.

[81] John H. Reppy. Asynchronous signals in Standard ML. Technical Report TR 90-1144, Department of Computer Science, Cornell University, Ithaca, NY 14853, August 1990.

[82] John H. Reppy. CML: A higher-order concurrent language. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1991. (revised 1993).

[83] John H. Reppy. *Higher-order Concurrency*. PhD thesis, Cornell University, 1992.

[84] John H. Reppy. *Concurrent Programming with Events - The Concurrent ML Manual*. Bell Labs, 1993.

[85] Clay Roach. Polymorphic futures in Java. The University of Texas at Austin, May 1998.

[86] Vijay A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards: Logic Programming. MIT Press, 1993.

[87] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. Dissertation, Technische Fakultät der Universität des Saarlandes, 1999. Submitted.

[88] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, Schloss Hagenberg, Linz, Austria, October 1997. Springer-Verlag.

[89] Christian Schulte. *Constraint Inference Engines*. Dissertation, Technische Fakultät der Universität des Saarlandes, 1999. To appear, preliminary title.

[90] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, New York, USA, November 1994. The MIT Press.

[91] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In Alan H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, May 1994. Springer-Verlag.

[92] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[93] Gert Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), February 1994.

[94] Gert Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, 7–9 September 1994. Springer-Verlag.

[95] Gert Smolka. The definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, Lecture Notes in Computer Science, vol. 910, pages 251–292. Springer-Verlag, 1995.

[96] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, 1995.

[97] Gert Smolka. Concurrent constraint programming based on functional programming. In Chris Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 1381, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.

[98] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[99] Richard M. Stallmann. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, MA, 1988–1998.

[100] Peter A. Steenkiste. The implementation of tags and run-time type checking. In Lee [58], chapter 1.

[101] Andrew Taylor. *High Performance Prolog Implementation.* PhD thesis, Basser Department of Computer Science, University of Sydney, June 1991.

[102] Ralf Treinen. Feature constraints with first-class features. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science*, Lecture Notes in Artificial Intelligence, vol. 711, pages 734–743, Gdańsk, Poland, 30 August–3 September 1993. Springer-Verlag.

[103] Ralf Treinen. Feature trees over arbitrary structures. In Patrick Blackburn and Maarten de Rijke, editors, *Specifying Syntactic Structures*, chapter 7, pages 185–211. CSLI Publications and FoLLI, 1997.

[104] Peter Van Roy. An Intermediate Language to Support Prolog's Unification. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1148–1164, Cleveland, Ohio, USA, 1989.

[105] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming.* PhD thesis, Computer Science Division (EECS), University of California, Berkeley, December 1990.

[106] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the aquarius prolog compiler. *COMPUTER*, January 1992.

[107] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[108] Peter Van Roy, Michael Mehl, and Ralf Scheidhauer. Integrating efficient records into concurrent constraint programming. In *International Symposium on Programming Languages, Implementations, Logics, and Programs*, Aachen, Germany, September 1996. Springer-Verlag.

[109] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231 – 248, 1990.

[110] David H. D. Warren. *Applied Logic – Its Use and Implementation as as Programming Tool.* PhD thesis, University of Edinburgh, 1977. Available as Technical Note 290, SRI International.

[111] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, Artifical Intelligence Center, SRI International, 1983.

[112] David H. D. Warren. The SRI model for Or-parallel execution of Prolog: Abstract design and implementation issues. In *Proceedings of the 1987 International Symposium on Logic Programming*, pages 92–102, 1987.

[113] David S. Warren. Efficient prolog memory management for flexible control strategies. *New Generation Computing*, 2(4):361–369, 1984.

[114] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, St. Malo, France, September 1992.

[115] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *In International Workshop on Memory Management*, Kinros, Scotland, UKs, September 1995.

[116] Jörg Würtz. Oz Scheduler: A workbench for scheduling problems. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, pages 132–139, Toulouse, France, November 1996. IEEE Computer Society Press.

[117] Jörg Würtz. Constraint-based scheduling in Oz. In *Symposium on Operations Research*, Braunschweig, Germany, 1997. Springer-Verlag.

[118] Jörg Würtz. *Lösen von kombinatorischen Problemen durch Constraintprogrammierung in Oz.* Dissertation, Universität des Saarlandes, Fachbereich Informatik, Postfach 1150, D-66041 Saarbrücken, Germany, 1998.

Index

This document was typeset with LaTeX at 12 point using the times font. The L and C++ listings where processed with the listings package from Carsten Heinz.