A Set-Based Calculus and its Implementation

vorgelegt von Diplom-Informatiker

Wolfgang Grieskamp

aus Essen

Am Fachbereich 13, Informatik,
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuß:

Vorsitzender	:	Prof. Dr. Günter Hommel
Berichter	:	Prof. Dr. Peter Pepper
Berichter	:	Prof. Dr. Stefan Jähnichen

Tag der wissenschaftlichen Aussprache: 26.11.1999

Berlin 1999

D 83

Meinen Eltern

Zusammenfassung

Diese Arbeit untersucht die *Fundierung* und *Implementierung* eines *Berechnungsmodells* für *mengenbasierte* Computersprachen. Es wird das μZ-Kalkül eingeführt, eine minimale mengenbasierte Ausdruckssprache, verwandt mit dem λ-Kalkül, das Anwendungssprachen wie die Spezifikationssprache Z oder funktional-logische Programmiersprachen komplett und natürlich einbetten kann. *Syntax*, *Semantik* und *Gleichungstheorie* von μZ werden definiert. Mehrere *Berechnungsmodelle* werden entwickelt. Durch gezielte Anwendung der Gleichungstheorie ergibt sich eine *Narrowingsemantik*, die durch lokale Modifikation der Redexauswahl nicht-strikt oder strikt ausfällt, und insbesondere interessant für interpretative "on-line" Auswertung ist, da sie auf symbolischer Termersetzung basiert. Das zentrale Berechnungsmodell aber basiert auf einer *natürlichen Semantik*, die durch ihren operationalen Charakter das Design kompilierter "off-line" Auswertung vorbereitet, und das *Berechnungsreferenzmodell* von μZ festlegt. Dieses Modell kombiniert funktionale Reduktion und nebenläufige Resolution, und definiert eine klare Trennung zwischen der (deterministischen, funktionalen) Konstruktion von Mengen und der (nichtdeterministischem, logischen) Dekonstruktion. Da Mengen sowohl extensional als auch intensional dargestellt werden können, ist das Modell von höherer Ordnung.

Die Implementierung erfolgt in Form einer abstrakten Maschine, der ZAM, welche auf der Technik nebenläufiger Resolutionsagenten basiert. Die Maschine realisiert eine effiziente Tiefensuche über die Verwaltung von Auswahlpunkten und Modifikationsprotokollen. Durch ein Prioritätenmodell in der nebenläufigen Ausführung werden nichtdeterministische Berechnungen dynamisch verzögert und deterministische propagiert. Die ZAM ist als Stackmaschine vollständig in Z spezifiert, und wird in der Implementierung in C++ zu einer Registermaschine verfeinert, welche *persistente* Register verwendet, um den Aufwand der Protokollierung von Modifikation zu minimieren.

Die Anwendung der vorgestellten Konzepte erfolgt im Rahmen des ZeTa Systems, im BMBF Projekt ESPRESS entwickelt, das verschiedenartige Notationen und Werkzeuge auf der semantischen Grundlage von Z integriert. Der ZaP Übersetzer, Bestandteil des ZeTa Systems, realisiert mit Hilfe des μZ-Kalküls die Ausführung von Z Modellen, und wurde beispielsweise in ESPRESS für die Automatisierung von Softwaretests eingesetzt.

Die Definition von μZ, der Berechnungsmodelle, der abstrakten Maschine und der Übersetzung von μZ in Maschinencode ist vollständig in der Spezifikationssprache Z gegeben (mit kleineren syntaktischen Erweiterungen), und daher syntax- und typgeprüft. Insofern ist diese Arbeit auch eine Fallstudie der Anwendbarkeit von Z für anspruchsvolle Probleme der Metamodellierung.

Preface

In response to those who thought that Cantor's theory of sets might be destroyed by the well-known paradoxes discovered at the beginning of this century, Hilbert once spoke about the "paradise created by Cantor from which nobody shall ever expel us" (quoted in Vaught [1995]). On the level of *non-constructive* calculi, the beauty of the set-based approach has been recognized in computer science in the formal notation Z [Spivey, 1992; Toyn, 1998], developed in the eighties and semantically based on Zermelo-Fraenkli set theory, which solves the paradoxes of classical set theory. In recent years, a growing number of authors have advocated the use of sets in declarative programming and program analysis.

In this thesis, I use the set-based approach as the basis for a novel, small intermediate calculus, which allows to naturally embed concepts from higher-order functional, logic and concurrent constraint programming, as well as set-based specification languages such as Z. I define the syntax and semantics of the calculus, analyze its equational theory, investigate several computation models, provide an *abstract machine* that implements a model, and define a compilation from the calculus to the machine. The power of the calculus and its implementation allows an unorthodox approach to the integration of higher-order functional and constraint logic programming.

The present thesis, written between January and September 1999, grew out of my activities in the application oriented research project ESPRESS (1995-1998), which was concerned with methods and tools for the specification of safety-critical embedded systems [Büssow et al., 1996; Büssow and Grieskamp, 1997; Büssow et al., 1997a; Grieskamp et al., 1998; Büssow et al., 1998; Büssow and Grieskamp, 1999]. One goal in the ESPRESS context was to provide a tool to execute Z specifications, for the purpose of evaluating test data and animating requirements specification of embedded systems based on Z. The development of a compiler for Z was influenced by my earlier experiences with the implementation of the algebraic-functional language Opal [Schulte and Grieskamp, 1992; Hartel et al., 1996; Didrich et al., 1994; Frauenstein et al., 1996a,b; Didrich et al., 1998], which strongly suggested basing such a compiler on a small intermediate language in the style of the λ-calculus. This led to a first design of the calculus presented in this thesis, called μZ because of its original purpose for abstracting from the diversity of the Z notation. This first version was used as an intermediate language in the experimental compiler ZaP, which is part of the Z-based tool-integration environment ZeTa [Büssow and Grieskamp, 1999], developed under the supervision of the author in ESPRESS. ZaP has proved to be a feasible tool, in particular for the evaluation of test data, and it is being used for this purpose in several projects conducted in cooperation with Daimler-Chrysler, FT3/SM, Berlin. The work presented here consolidates and refines the technology of ZaP on the basis of these experiences. On the one hand, it closes the gap between the exper-

imental prototype ZaP and its theoretical foundation, and on the other it extends ZaP's model by indeterministic computation, which proved to be desirable in the course of the case studies.

Acknowledgments

I would like to thank Prof. Dr. Peter Pepper for his continuous support on both a personal and professional level over the past years. Peter's tolerance combined with brilliance has been a major yardstick for me. I would also like to thank Prof. Dr. Stefan Jähnichen and Prof. Dr. Hartmut Ehrig for their encouragement and confidence in my ESPRESS work.

Numerous colleagues at Peters group, ÜBB, have indirectly contributed to this thesis, among them Klaus Didrich, Michael Cebulla and Christian Maeder, not at least by taking over some of my daily obligations during the time of the writing. In ESPRESS, the collaboration with Robert Büssow on the ZeTa system and other topics was very fruitful and pleasuring. The exchange of ideas on Z, among other things, with Andreas Fett, Thomas Santen and other colleagues of the ESPRESS context helped me to gain insights to aspects of this work. The discussions I had from time to time with Markus Lepper were highly stimulating. My special thanks go to Erich Mikk for his invaluable feedback on early prototypes of the ZaP compiler and to Jacob Wieland for the fruitful discussion of the μZ calculus and its use for program analysis in his diploma thesis [Wieland, 1999].

Contents

1 Introduction **17**

2 Meta Notation **23**
2.1 A Sketch of Z 25
2.1.1 Types and Sematic Universe 25
2.1.2 Schema Text 25
2.1.3 Genericity 26
2.1.4 Undefinedness 26
2.2 Notational Extensions 28
2.2.1 Where Notation 28
2.2.2 If-Exists Notation 28
2.2.3 Inference Rule Systems 28
2.2.4 Recursive Partial Functions 29
2.2.5 Local Variables in Schema Boxes 29
2.2.6 Case Distinction in Predicates 30
2.2.7 Selective Δ 30
2.2.8 Overloading 30
2.2.9 Further Extensions 31
2.3 Mathematical Tools 32
2.3.1 Sets and Relations 32
2.3.2 Sequences 32
2.3.3 Booleans 33
2.3.4 Order and Lattices 33

3 The μZ Calculus **37**
3.1 Introduction to μZ 38
3.1.1 Syntax and Informal Meaning 38
3.1.2 Encodings 39
3.2 Semantic Model 43
3.2.1 Set Representation 43
3.2.2 Types 44
3.2.3 Universe 45
3.2.4 Free Types 46
3.2.5 Standard Types and Constructors 47
3.2.6 Order and Fixed-Points 48
3.3 Expression Syntax and Meaning 50

3.3.1 Variable, Expression and Pattern Type . . . 50
3.3.2 Typing Relation . . . 50
3.3.3 Meaning Function . . . 51
3.3.4 Basic Forms: Typing and Meaning . . . 52
3.3.5 Expression Abbreviation Forms . . . 56
3.4 Equational Theory . . . 57
3.4.1 Syntactic Tools . . . 57
3.4.2 Equivalence Relation . . . 57
3.4.3 Boolean Laws . . . 58
3.4.4 Singleton Set Laws . . . 59
3.4.5 Translation Laws . . . 60
3.4.6 Schema Laws . . . 64
3.4.7 Fixed-Point Unrolling Law . . . 66
3.5 Discussion . . . 67

4 Computation in μZ 71
4.1 Preliminaries . . . 72
4.2 Normalization . . . 74
4.2.1 Property Normalization . . . 75
4.2.2 Normalization Rules . . . 76
4.2.3 Properties of Normalization . . . 79
4.3 Narrowing Semantics . . . 82
4.3.1 General Narrowing . . . 82
4.3.2 Outermost-in Narrowing . . . 84
4.3.3 Strict Narrowing . . . 85
4.4 The Reference Computation Model . . . 86
4.4.1 Values and Constraints . . . 87
4.4.2 Computation . . . 88
4.5 Discussion . . . 93

5 Abstract Machine 97
5.1 Basic Model of the ZAM . . . 98
5.1.1 Values . . . 98
5.1.2 Intensions . . . 99
5.1.3 Threads and Goals . . . 100
5.1.4 Configurations . . . 102
5.1.5 Instructions . . . 102
5.2 Specification . . . 105
5.2.1 Axiomatic Definitions . . . 105
5.2.2 Auxiliary Instructions . . . 107
5.2.3 Execution Step and Selection . . . 109
5.2.4 Instructions . . . 110
5.3 Compilation . . . 124
5.3.1 Environment . . . 124
5.3.2 Compilation Functions . . . 125
5.4 Implementation . . . 129

5.4.1 Data Model . . . 129
5.4.2 Instruction Set . . . 131
5.4.3 Memory Management . . . 133
5.4.4 Performance . . . 133
5.5 Discussion . . . 135

6 The ZeTa System 139
6.1 Three Dimensions of Integration . . . 139
6.1.1 Semantic Integration . . . 140
6.1.2 Document Integration . . . 141
6.1.3 Tool Integration . . . 141
6.1.4 Graphical User Interfaces . . . 143
6.2 The ZaP Compiler . . . 144
6.2.1 The Basic Specification of the Birthday Book . . . 146
6.2.2 Refining the Birthday Book for Execution . . . 147
6.2.3 Refining the Birthday Book for Testing . . . 150
6.3 Discussion . . . 154

7 Conclusion 157
7.1 Contributions . . . 157
7.2 Future Work . . . 159

List of Figures

3.1 Type Language 44
3.2 Type Language: Syntactic Tools 45
3.3 Semantic Universe 45
3.4 Free Types: Syntax 46
3.5 Free Types: Meaning 47
3.6 Free Type of Booleans 47
3.7 Free Type of Natural Numbers 47
3.8 The Order on Partial Sets 48
3.9 Expressions and Patterns 50
3.10 Typing Relation: Declarations 51
3.11 Meaning Function: Declarations 51
3.12 Value Matching 51
3.13 Syntax and Meaning: Constructor Application 52
3.14 Syntax and Meaning: Singleton Set Display and Selection 52
3.15 Syntax and Meaning: Set Algebra 53
3.16 Syntax and Meaning: Set Translation 54
3.17 Syntax and Meaning: Schema, Fixed-Point, and Variable 55
3.18 Expression Abbreviation Forms 56
3.19 Syntactic Tools 57
3.20 Equivalence Relation: Declaration 58
3.21 Boolean Laws 58
3.22 Excluded Middle Laws 59
3.23 Selection Elimination Law 59
3.24 Constructor Decomposition Laws 60
3.25 Translation Composition and Identity Laws 60
3.26 Translation Distributivity Laws 61
3.27 Translation Product Law 62
3.28 Translation Lifting Law 63
3.29 Translation Splitting Law 63
3.30 Translation Intersection Elimination Law 63
3.31 Schema Elimination and Distributivity Laws 64
3.32 Schema Complement Lifting Law 65
3.33 Schema Translation Elimination Laws 65
3.34 Schema Membership Lifting Law 65
3.35 Schema Property Substitution Law 66
3.36 Fixed-Point Unrolling Law 66

4.1 Extensional Equality and Unequality 72
4.2 Matching Contexts 73
4.3 Normalized Expressions 74
4.4 Singleton Set Normalization 76
4.5 Constructor Normalization 76
4.6 Union Normalization 77
4.7 Intersection Normalization 77
4.8 Complement Normalization 78
4.9 Translation Normalization 79
4.10 Schema Normalization 79
4.11 Narrowing Rules 82
4.12 Values and Constraints 87
4.13 Strict Context Reduction 89
4.14 Variable, Schema, and Fixed-Point Reduction 89
4.15 Translation and Complement Reduction 89
4.16 Set Value Reduction 90
4.17 Mu-Value Reduction 90
4.18 Compound Constraint Resolution 91
4.19 Expression Reduction in Constraints 91
4.20 Membership and Equality Resolution 92
4.21 Test-Emptiness Resolution 92

5.1 ZAM Values 98
5.2 ZAM Intensions 99
5.3 ZAM Threads and Goals 101
5.4 ZAM Configurations 102
5.5 ZAM Instructions 103
5.6 Shifting Values 106
5.7 Freezing Values 106
5.8 Getting the Variables of a Value 106
5.9 Equality 107
5.10 Unification 108
5.11 Merging and Joining Value Sequences 108
5.12 Adding Information to Choices 109
5.13 Ordering Index Sets 109
5.14 ZAM Auxiliary Instructions 109
5.15 ZAM Execution Step 110
5.16 ZAM Dispatching 111
5.17 The *LOAD* Instruction 111
5.18 The *WAIT* Instruction 112
5.19 The *LOADENV* Instruction 112
5.20 The *UNIFY* Instruction 113
5.21 The *STORE* Instruction 113
5.22 The *MEMBER* Instruction 114
5.23 The *TRYNEXT* Auxiliary Instruction: The Empty Case 114
5.24 The *TRYNEXT* Auxiliary Instruction: The Extensional Case 115

5.25 The *TRYNEXT* Auxiliary Instruction: The Intensional Case 115
5.26 The *MKEMPTY* and *MKSINGLE* Instructions 116
5.27 The *MKTERM* Instruction 116
5.28 The *MKINTEN* Instruction 116
5.29 The *UNION* Instruction 117
5.30 The *ISECT* Instruction 117
5.31 The *SUCCESS* Instruction 118
5.32 The *MU* Instruction 118
5.33 The *TEST* and *TESTNATIVE* Instructions 119
5.34 The *SEARCH* Auxiliary Instruction: Dispatching 120
5.35 The *SEARCH* Auxiliary Instruction: Select, More Intensions 120
5.36 The *SEARCH* Auxiliary Instruction: Select: No More Intensions 121
5.37 The *SEARCH* Auxiliary Instruction: Resume: Subgoal Success 122
5.38 The *SEARCH* Auxiliary Instruction: Resume: Subgoal Failure 123
5.39 The *SEARCH* Auxiliary Instruction: Backtracking 123
5.40 Compilation Environment 124
5.41 Declaration of Compilation Functions 125
5.42 Compiling General Expressions 125
5.43 Compiling Disjunctions and Conjunctions 126
5.44 Compiling Literals 126
5.45 Compiling Properties 127
5.46 Value Types of the ZAM in C++ 130
5.47 Control Types of the ZAM in C++ 130
5.48 Register Transfer Instruction Set of the ZAM 131

6.1 Semantic Integration in ZeTa 140
6.2 Document Integration in ZeTa 141
6.3 Chains of Content Queries 143
6.4 The XEmacs GUI 144
6.5 The Swing GUI 145

Ich ging im Walde
So für mich hin,
Und nichts zu suchen,
Das war mein Sinn.
(Goethe)
[1]

Chapter 1

Introduction

The automatic evaluation of *test data* for *safety-critical embedded systems* is an interesting application that can help to put declarative language implementation techniques into industrial practice. Some studies report that more than 50% of development costs in this application area go into testing. A setting for test-case evaluation that can improve this situation is as follows: given a requirements specification by an executable prototype, some input data describing a test case, and the output data from a run of the system's implementation on the given input, we check by executing the specification whether the implementation meets its requirements. At the first sight, this goal would seem to be simple, since input and output data are fixed. However, a real-world specification of a complex system may contain a lot of "hidden" data, which is used to describe the observable behavior. Thus, the problem scales up to finding the solution(s) to (a sequence of) partial data bindings.

In the application-oriented research project ESPRESS[2], which is concerned with methods and tools for the development of embedded systems, the set-based specification language Z [Spivey, 1992] is used for requirements specification in combination with other notations such as Statecharts which are incorporated by a shallow encoding in Z [Büssow and Grieskamp, 1999]. Performing test-case evaluation in this setting requires a *computation model* for Z.

In this thesis, the underlying *theory* and *implementation* of a computation model for Z are investigated. In fact, the work is not restricted to Z, since it is based on a small intermediate calculus, called μZ, that can embed Z as well as other declarative notations. The major contributions are the introduction of the novel calculus μZ, the definition of the calculus' computation model (operational semantics) and the definition of an abstract machine for implementing the calculus. In this initiatory Chapter, an outline of these topics is given.

[1] Im Goethejahr 1999 haben wir uns vorgenommen, (fast) alle Kapitelwahlsprüche beim großen deutschen Dichter auszuleihen.

[2] ESPRESS is a joint project involving industrial partners (Daimler-Benz AG and Robert Bosch AG) and academic research institutes (GMD-First, FhG-ISST, and TU-Berlin), funded by the German "Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie".

The μZ Calculus

The μZ calculus is a pure expression language in the spirit of the λ-calculus. Since it can embed the full Z language – the simple-typed λ-calculus, set calculus, predicate calculus, and schema calculus – computation in μZ can not be complete. However, a computation model and implementation by an abstract machine can be provided that is comparable with that of higher-order functional logic paradigms [Smolka, 1998; Hanus, 1999; Chakravarty et al., 1997].

The μZ calculus is based upon a small number of constructs, which are tentatively summarized below ($e \in EXP$ being expressions and $p \in PAT$ patterns):

$$
\begin{aligned}
e \in EXP &::= x \mid \rho(\overline{e}) \mid \{e\} \mid \mu\, e \mid \mathsf{0} \mid e \cap e' \mid e \cup e' \mid \sim e \mid \\
&\quad\;\; e[p_1 \mapsto p_2] \mid \{p \mid e\} \mid \mathbf{fix}\, p \lhd e \\
p \in PAT &\subseteq EXP
\end{aligned}
$$

The calculus employs constructor terms, $\rho(\overline{e})$, and the basic operations of set algebra, set union, intersection and complement. Patterns, $p \in PAT$, are the subset of expressions built from constructor terms and variables. In addition, the following expression forms are provided:

- The form $\mu\, e$ selects the element of a singleton set e.
- The form $e[p_1 \mapsto p_2]$ describes the *translation* of the set e. The result consists of all elements that match p_2, such that there exists an element in e that matches the pattern p_1 under the same substitution. For example, $(\{\rho(1,2)\} \cup \{\rho(1,3)\})[\rho(x,y) \mapsto x] \equiv \{1\}$ (where numbers are special 0-ary constructor symbols).
- The form $\{p \mid e\}$ describes a *schema* (set comprehension). The property e is a set expression of Boolean type – truth being the set containing a dedicated singleton element and falsity the empty set.
- The form $\mathbf{fix}\, p \lhd e$ describes a fixed-point of a set (or tuple of sets) e.

The calculus is able to represent all concepts of Z in a remarkably consistent way. For example, a Cartesian product $e_1 \times e_2$ is expressed as $e_1[x \mapsto (x, _)] \cap e_2[y \mapsto (_, y)]$, with $(_, _)$, the constructor symbol for pairs, used in mixfix notation, and $_$ a pattern "joker" (or anonymous variable). Relational application, $R\, e$, is encoded as $(R \cap \{e\})[_ \mapsto \diamond]$, where $\diamond$ is the 0-ary constructor for the unit in Boolean sets. Function application, $e_1\, e_2$, is given as $\mu((e_1 \cap \{e_2\}[x \mapsto (x, _)])[(_, y) \mapsto y])$. As these examples indicate, the heart of the μZ calculus is its novel construct for set translation, which provides a means for the constructive description of morphisms in the category of μZ sets.

In Chapter 3, we develop the exact syntax and semantics of the calculus and provide an equational theory for it. The semantic model of μZ is based on a hierarchical typed universe, where sets are represented as *partial characteristic (Boolean) functions* in a set-theoretical sense, which can be founded by Z itself or by ZF set theory. This representation of sets is a generalization of partial three-valued logics [Owe, 1997] to the case of set algebra. The equational theory of μZ is powerful enough to induce a *narrowing semantics* by a pure expression transformation, as will be shown in Chapter 4.

Computation in μZ

Several notions of computation are investigated in Chapter 4. We start with a *normalization* of expressions to a disjunctive normal form, which, combined with simplification, is a prerequisite for all further investigations (including compilation). From the normalization, we can easily deduce a general *narrowing semantics* which is suited for symbolic "on-line" computation. By restricting the redex selection order of narrowing, several submodels are derived.

The reference computation model of μZ, however, is given in the style of *natural semantics*, constituting a trade-off between computation power and efficient implementability, preparing a straight transition to the abstract machine given in Chapter 5 for "off-line" computation. This model draws a clear borderline between the *functional* (deterministic) construction of sets and the *logic* (indeterministic) deconstruction. For example, in an expression such as $\{x_1 \mid x_1 \in e_1\} \cap \{x_2 \mid x_2 \in e_2\}$ (where $_ \in _$ is syntactic sugar), the expressions e_1 and e_2 are concurrently reduced in a deterministic functional style. Once an expression e_i has reached a value form, a resolution step is performed, thereby binding the variable x_i.

The reference computation model of μZ is comparable with that of higher-order functional logic languages with concurrent constraint resolution, though it goes beyond this, supporting disjunction and negation. It supports the free combination of sets (relations and functions) by set union, intersection and complement, thereby abstracting from intensional and extensional representations. The widely used set-based Z specification style can thus be fully supported. We give some examples in Z. Given the function f, the execution of its domain restriction, $\{e_1, \ldots, e_n\} \lhd f$, causes no problems. We can also execute the properties $x \in \mathsf{dom} f$ or $x \in \mathsf{ran} f$. We may execute $f^{\sim}$, which might become a relation, if f is not injective. If g is a further function, then $f \cup g$ and $f \cap g$ are executable. Similar possibilities carry over to the schema calculus, which can be fully executed as well.

The limitations of the computation model are found in the executability of universal quantification and of complement. The universal quantification, $\forall x \bullet e$, is encoded in μZ as $\sim(\{x \mid \sim e\}[x \mapsto \diamond])$. The problem of executing this construct is conspicuously apparent in the equational theory of μZ: set complement cannot be distributed over hiding translation. Thus, we cannot expect complements to be pushed down completely to the expression leafs during the construction of the disjunctive normal form. We deal with this problem by computing independent "subgoals" for expressions nested inside of complements. These computations do not contribute to resolution in their enclosing context, and are bound to the context by residuation. For the universal quantifier, this entails that it can be computed provided the quantified range is finitely enumeratable.

The Abstract Machine ZAM

An abstract machine – called ZAM – which implements the reference computation model of μZ is developed in Chapter 5. In the ZAM, concurrent threads collaboratively work on a resolution task. The threads synchronize via variables: accessing an unbound variable suspends a thread and binding a variable resumes threads waiting for it (residuation). The machine maintains a stack of choice points which is shared by the threads jointly acting on a resolution. When a thread executes a property such as $x \in e_1 \cup e_2$, it continues

with $x \in e_1$ and creates a choice point for $x \in e_2$. A simple but powerful optimization is achieved by an instance of the "Andorra Principle" [Warren, 1990], dynamically lowering the priority of threads that try to create choice points, thereby giving deterministic computation preference.

The compilation of μZ to ZAM instructions is based on the disjunctive normal form defined in Chapter 4. The instruction set of the ZAM is at a higher level than, for example, of the Warren Abstract Machine [Warren, 1983], making compilation easy and retraceable.

In Chapter 5, we give a comprehensive specification of the ZAM in Z's sequential specification style. This specification is intended to be executable using the concepts presented in this thesis. A prototype of the ZAM has been implemented in C++, and will be also outlined.

The ZeTa System and Conclusion

In Chapter 6, we give a short overview of the design of the ZeTa system [Büssow et al., 1998; Büssow and Grieskamp, 1999], and the role of the ZAP compiler as integrated into ZeTa. The goal of the ZeTa system is to offer an *open* environment for editing, browsing and analyzing integrated specifications assembled from heterogeneous formalisms – such as Statecharts, Z, temporal logic, message sequence charts, and other. The formalisms are semantically integrated by a shallow encoding in Z, using the $\mu\mathcal{SZ}$ conventions [Büssow et al., 1997a]. The ZeTa system is designed to systematically support such encodings by three dimensions of integration: semantic integration, document integration and tool integration. The ZAP compiler, part of the ZeTa system, can thus be used not only to execute Z specifications, but also other formalisms that are mapped to Z and thereby to μZ.

Chapter 7 concludes with a summary of the contributions of this work and of future lines of research. A discussion of related work can be found at the end of each of the relevant chapters.

Reading Advices

Since this thesis aims at a comprehensive formalization of the concepts it introduces, it contains a lot of detailed formal material, most of which as been put into figures that are interleaved with the text. For layout technical reasons, figures occasionally occur before their contents is explained. It is recommended to read (at least) until the first reference to a figure in the text until its content is read.

Not all parts of this thesis need to be thoroughly read to get an idea of the main concepts. Chapter 2 may be skipped by readers which are firm to Z and which do not want to know how we explain certain Z extensions that have common roots in well-known notational concepts (such as inference rules). In Chapter 3, it might be sufficient to read the introduction to the μZ calculus, Section 3.1 (on page 38), then the equational theory, Section 3.4 (on page 57), skipping the proofs in there, and finally the discussion. In Chapter 4, the definition of the distributive normal form in the introduction of Section 4.2 (on page 74) and the entire Section 4.4 (on page 86) is essential, and, of course, the discussion. In Chapter 5, the reader might find it sufficient to read Section 5.1 (on page 98),

Section 5.4 (on page 129) and the discussion. Chapter 6 may be skipped by readers not interested in the application.

Seh' ich die Werke der Meister an,
So seh' ich das, was sie getan.
(Goethe)

Chapter 2

Meta Notation

The meta notation used throughout this work is strictly based on the *Z language* [Spivey, 1992]. A brief introduction to Z is given in Section 2.1 (on page 25). In general, the reader is expected to be familiar with Z.

One obvious advantage of using Z is that it is *standardized*[1]. Since Z is also a *formal language*, another important advantage is that it is amenable to machine support. Thus, most of the mathematical material in this thesis is *verified by a parser and type-checker*[2]. Exceptions to the use of type-checked Z may appear in mathematical phrases inlined in text and where template notations such as $f(x_1, \ldots, x_n)$ provide a better way of arguing than conventional Z. This is seldom the case, and is only applied when *properties* of objects are described, not for their declaration.

Using a formal notation such as Z does not necessarily imply using a formal method in the sense of formal reasoning and proof with machine support. The degree of detail of description required to this end would be enormous. A significant amount of description is therefore kept loose – informal assertions are made that are clear to a human reader but unamenable to a machine. An important example of such usage is when we informally select the "smallest" solution to some axioms. Though Z is powerful enough to express such constraints, it would be cumbersome to write them down, and would thereby distract from the real concern.

The Z language, though a general specification notation, is tailored to the description of stated-based sequential systems. A typical instance of this usage is the specification of the abstract machine, ZAM, developed in Chapter 5. However, in other parts of this thesis, description techniques such as inference rules and recursive partial functions over free term algebras (abstract syntax) are required, which are not directly supported by Z's syntax. Some extensions are therefore made to the language (Section 2.2 (on page 28)), which are explained as syntactic sugar, which is implemented using the macro-preprocessor of the *ESZ* type checker. The printed output of the extensions looks natural, and all are actually type-checked and have a well-defined meaning in Z.

[1] An ISO standardization of Z is currently in progress. We follow the the draft proposals for the Z standard, [Z-ISO]. A good text book on Z is written by Spivey [1992]; the innovations in the forthcoming standard relative to this source are described by Toyn [1998].

[2] The *ESZ* type-checker (written by the author), which is part of the ZeTa system, is used. See also Chapter 6.

Z provides a powerful mathematical toolkit for defining basic notions such as numbers, relations, and functions. It is assumed that the reader is familiar with Z's toolkit, which is a fairly standard account of basic set theory (see Spivey [1992] for a specification). Some extensions to the toolkit are nevertheless required: they are defined in Section 2.3 (on page 32).

2.1 A Sketch of Z

The basic model of Z is reviewed below, without the intention of providing a complete picture. For a comprehensive description of Z, see [Spivey, 1992; Toyn, 1998].

2.1.1 Types and Sematic Universe

The model of Z is based on *typed set theory*. Given a collection of so-called *given types*, Z types are freely constructed by *power set construction*, $\mathbb{P}\,A$, *cartesian product*, $A_1 \times \ldots \times A_n$, and *binding set construction*, $[x_1 : A_1; \ \ldots ; \ x_n : A_n]$. Binding sets denote products with named components. A semantic universe for Z is large enough to contain meanings for these types, where given types may have assigned arbitrary (not necessarily countable) domains. Because of the type hierarchy, Z avoids Russel's Antinomy: sets are well-founded, and no set contains itself.

A distinguishing feature of Z is that types are "first-order citizens" of the language. A type is represented by an expression which denotes a set. As a set, it just has the property of being the *largest* set containing elements of the same type.

Z has a static typing discipline which forbids terms which are not well-typed. A declaration such as $x : E$ has two effects in Z: the variable x ranges over the set denoted by the expression E ($x \in E$), *and* x is declared to be of some type, which is the largest superset of E. For example, the constant $\mathbb{Z}$ represents the given type of integer numbers, and the constant $\mathbb{N} \subset \mathbb{Z}$ a subset of it. Then, declaring $x : \mathbb{N}$ lets x range over $\mathbb{N}$ and assigns the type $\mathbb{Z}$ to it for type checking.

2.1.2 Schema Text

A central notion of Z is that of *schema text*, which denotes a set of bindings (tuples with named components). A schema text, $D \mid P$, is built from a set of declarations D and a set of properties P. The declarations spawn a binding set, which is constrained by P. The property is built of first-order predicative formulas over variables representing the binding components.

For instance, the expression $[x, y : \mathbb{N} \mid x \leq y]$ represents the set of bindings where both components x and y are constrained to be natural numbers by the declaration, and where x is constrained to be less than or equal to y by the property.

In Z, it is always possible to shift constraints induced by declarations to the property part. For example, the above schema can be transformed into $[x, y : \mathbb{Z} \mid x \in \mathbb{N} \land y \in \mathbb{N} \land x \leq y]$.

Schema text is used in manifold ways in Z:

- Schema text is used to directly denote sets of bindings in expressions, as in $[x, y : \mathbb{N} \mid x \leq y]$, which denotes the set of bindings where component x is less or equal to component y. (We already used this notation above).

- Schema text is used in quantifiers. For example, the universal quantifier is written as $\forall D \mid P \bullet Q$, where $D \mid P$ spawns the range of quantified values. This notation is equivalent to $\forall D \bullet P \Rightarrow Q$. Similarly, $\exists D \mid P \bullet Q$, which is equivalent to $\exists D \bullet P \land Q$.

- Schema text is used in function abstractions, $\lambda D \mid P \bullet E$, where $D \mid P$ denotes the domain of the function. The binding set $D \mid P$ is converted into a set of tuples by removing the names of the components. For instance, the domain of the abstraction $\lambda x : \mathbb{N};\ y : \mathbb{N} \mid y \neq 0 \bullet x/y$ is the set of pairs (x representing the first and y the second component) with the specified property ($y \neq 0$). The graph of the function described is a subset of $\mathbb{P}((\mathbb{N} \times \mathbb{N}) \times \mathbb{N})$.

- Schema text is used in set comprehension, $\{D \mid P\}$, where the tuples of $D \mid P$ are the elements of the comprehended set. A further variant of comprehension, $\{D \mid P \bullet E\}$, allows us to explicitly construct the result tuples, as e.g. in $\{x : \mathbb{N} \mid x \mathop{\mathsf{mod}} 2 = 0 \bullet (x, x+1)\}$, which describes the set of pairs where an even number is related with its direct successor.

- Schema text is also used for introducing global constants. Using the notation

  $$\begin{array}{|l} D \\ \hline P \end{array}$$

  the components of the bindings of $D \mid P$ are introduced as global constants. If $D \mid P$ contains more then one binding, the constants are loosely specified. If $D \mid P$ is empty, the specification is inconsistent.

2.1.3 Genericity

Z supports *genericity*. Global constants of a specification may be generic over a list of "formal" types. The meaning of a generic constant is a *total* function from an *instantiation* of the formals into a value. The instantiations are not just type "terms" but arbitrary set values. This distinguishes Z's polymorphism from the Hindley-Milner polymorphism found in functional programming languages and specification formalisms [Damas and Milner, 1982]. For example, if $\mathbb{Z}$ is the given type of integer numbers, then a generic constant such as $\mathrm{seq}\,\alpha$ may be instantiated as $\mathrm{seq}\,\mathbb{Z}$, $\mathrm{seq}\{x : \mathbb{Z} \mid x \geq 0\}$, $\mathrm{seq}\{1, 2\}$, and so on.

2.1.4 Undefinedness

The semantics of Standard Z does not prescribe a particular view of *undefinedness*. The treatment of undefinedness is left to conventions introduced by specifiers and/or tools. Throughout this thesis, the following conventions are adopted, which are oriented to desirable properties of a partial logic, as described, e.g., by Owe [1997]:

- There is a special predicate for testing the definedness of an expression: **dfd** e iff e is defined.

- The undefined value can be denoted by the expression $\perp$, which is an abbreviation for $\mu\,\varnothing$.

- Undefinedness strictly propagates in relational application, $e\ e'$, and in relational test, $e\ \underline{R}\ e'$.

- A set may have "unknowns", that is the proposition $x \in X$ may be undefined for certain x's. For schema text, $x : X \mid P\,x$, if the proposition $P\,c$ is undefined for some c, then $c \in \{x : X \mid P\,x\}$ will be as well. At the same time, for some other c' such that $P\,c'$ is defined, $c' \in \{x : X \mid P\,x\}$ is defined.

- Intersection and union of sets in set algebra or the schema calculus can deal with unknowns. Briefly, if $x \in X_1$ is unknown and $x \in X_2$ is false, then $x \in X_1 \cap X_2$ is false. This behavior is commutative and associative. Symmetrically, if $x \in X_1$ is unknown and $x \in X_2$ is true, then $x \in X_1 \cup X_2$ is true.

- Logical connectives can be seen as a special case of set-union and set-intersection, and treat undefinedness in a similar way. Moreover, if P is undefined, then $\neg\ P$ is as well. The conditional expression, $\textbf{if}\ \ p\ \ \textbf{then}\ \ e\ \ \textbf{else}\ \ e'$, is treated as an abbreviation for $\mu\,x : \alpha \mid p \Rightarrow x = e \wedge \neg\ p \Rightarrow x = e'$, and is therefore non-strict.

- Quantifiers are treated as the usual expansion to *and*-chains and *or*-chains regarding undefinedness.

Note that the above treatment still is loose as regards the representation of undefinedness in a *model*: we have merely stated some properties. The μZ calculus is more explicit, adding a $\perp$ element to the carriers, which can be seen as a refinement of the above.

In Section 2.2.4 (on page 29), syntactic sugar will be introduced which allows a convenient notation for recursive partial functions making use of these conventions – in particular of the $\textbf{dfd}\,e$ predicate.

2.2 Notational Extensions

Throughout this thesis, a few purely syntactical extensions of Z will be used, which embed well-known mathematical notation techniques in the Z framework.

2.2.1 Where Notation

In Z, axioms of the kind $\forall D \mid P \bullet Q$ are commonly used at a top-level position of a specification, where no outer scope exists. For this kind of "shallow" quantification, the $\forall$-prefix notation distracts from the actual specification goal, the predicate Q, which contains free variables bound by $D \mid P$. To enhance readability, we allow the alternative notation Q **where** $D \mid P$. The binding priorities for Q are lowered to the level of schema properties, such that newline and "; " can be used instead of logical conjunction, "$\land$". Thus the notation

$$
\begin{array}{l}
Q_1;\ Q_2 \\
Q_3;\ Q_4 \\
\quad \mathbf{where}\ D \mid P
\end{array}
$$

is syntactic sugar for $\forall D \mid P \bullet Q_1 \land Q_2 \land Q_3 \land Q_4$.

2.2.2 If-Exists Notation

An often encountered situation in specification is the desire to formulate a "conditional let": if a unique binding for some schema text exists, then an expression should be evaluated under this binding, otherwise an alternative expression should be evaluated. In plain Z, one has to write **if** $\exists_1 D \mid P$ **then** $\mu D \mid P \bullet E_1$ **else** E_2 to achieve this effect. We allow the notation $\mathbf{if}_{\exists_1}\ D \mid P$ **then** E_1 **else** E_2 as syntactic sugar for the above phrase.

2.2.3 Inference Rule Systems

In computer science, a widely used notation for axioms of the kind $\forall D \mid P \bullet Q$ are so-called *inference rules*:

$$\frac{P}{Q}\ {\scriptstyle D}$$

We allow this notation as syntactic sugar for the universal quantification.

Systems of inference rules are often used in this thesis to inductively describe relations on syntax. When defining a relation with an inference rule system, in addition to the axioms described by the rules, a principle of *generation* is assumed. We say that a relation is the *smallest* one which satisfies the rules. Formally, let $R : T$ be the relation defined (with T its type) and $I_1[R]$ upto $I_n[R]$ the rules:

$$R = \bigcap\{R' : T \mid I_1[R'];\ \ldots;\ I_n[R']\}$$

In this thesis, a formalization of generation is not usually given. However, it will be sufficiently treated in the explanatory text.

2.2.4 Recursive Partial Functions

A commonly used technique for describing recursive partial functions is to use equations of the form $f\ E \Leftarrow E'$. Here, the value E is in f's domain iff E' is defined. If so, the mapping $E \mapsto E'$ is in the functions graph. For example, an interpretation function which maps terms to semantic values is defined iff the interpretation of subterms is defined. This technique is indispensable for readability in those situations where the description of the definedness and mapping cannot be adequately partitioned.

Throughout this thesis, we use the notation

$$\begin{array}{l} E_0\ \ E_1 \Leftarrow E'_1 \\ \quad \mid \ldots \Leftarrow \ldots \\ \quad \mid E_n \Leftarrow E'_n \\ \quad \mathbf{where}\ D \mid P \end{array}$$

as a syntactic abbreviation for the Z form:

$$\begin{array}{l} \forall D \mid P \bullet (E_1 \in \mathsf{dom}\, E_0 \Leftrightarrow \mathbf{dfd}\, E'_1) \wedge (E_1 \in \mathsf{dom}\, E_0 \Rightarrow E_0\, E_1 = E'_1) \\ \qquad\qquad \wedge \ldots \wedge \\ \qquad\qquad (E_n \in \mathsf{dom}\, E_0 \Leftrightarrow \mathbf{dfd}\, E'_n) \wedge (E_n \in \mathsf{dom}\, E_0 \Rightarrow E_0\, E_n = E'_n) \end{array}$$

In general, with this style of definition, the left-hand sides E_i may be not exhaustive. Moreover, we may spread equations for one function E_0 over several paragraphs. We normally choose the *smallest* solution for a function defined by a system of recursive partial definitions, given in one or several paragraphs. It is constructed similarly as for relations (see Section 2.2.3 (on the facing page)).

If patterns of definitions overlap, we treat them to be priorized by textual order. For example, the following recursive function definition removes each second element in a sequence:

$$\begin{array}{|l} [A] \\ \hline \mathit{remove}2 : \mathrm{seq}\, A \longrightarrow \mathrm{seq}\, A \\ \hline \mathit{remove}2\ (\langle x, y \rangle \frown s) \Leftarrow \langle x \rangle \frown \mathit{remove}2(s) \\ \quad \mid \quad (s) \qquad \Leftarrow s \\ \quad \mathbf{where}\ x, y : A;\ s : \mathrm{seq}\, A \\ \hline \end{array}$$

The second definition case captures any sequence, but has less priority than the first case, such that sequences whose length is greater then one are handled by the first case.

2.2.5 Local Variables in Schema Boxes

When specifying sequential systems using schemas in vertical box notation, one often encounters the necessity of introducing local variables. In Z, this is has to be notated as shown in the left-hand side below:

S	S
$\Delta State$	$\Delta State$
$\exists\, Locals \bullet$ $P_1 \land$ $\ldots \land$ P_n	$Locals$ .... P_1 $\ldots$ P_n

One obvious problem of this notation is that it becomes necessary to separate the properties P_i in the context of the existential quantifier by "$\land$" instead of line break or ";" which would be more readable. We therefore introduce the notation on the right-hand side as a shortcut for the left-hand side.

2.2.6 Case Distinction in Predicates

Case distinction in predicates must be denoted in Z as on the left-hand side below. We introduce the notation on the right-hand side as a much more instructive variant:

S	S
$\Delta State$	$\Delta State$
$x > 0 \Rightarrow x' = x - 1$ $\neg\, x > 0 \Rightarrow x' = x + 1$	$\mathbf{if}\ x > 0$ $\mathbf{then}\ x' = x - 1$ $\mathbf{else}\ \ x' = x + 1$

2.2.7 Selective Δ

When specifying state transition operations in Z's sequential specification style, one uses the ΔS operator to create pre and post state versions of the fields of the schema S. If certain fields are not going to be changed, equations such as $x = x'$ have to be added to the operation schema. Apart of notational noise, this diminishes localizability of changes to S.

We therefore use the notation $\Delta[x_1, \ldots, x_n]S$ to express that, from the schema S, *only* the variables x_i are changed, and all other keep their value. This is syntactic sugar for

$$[\Delta S \mid \theta(S \setminus (x_1, \ldots, x_n)) = \theta(S' \setminus (x'_1, \ldots, x'_n))]$$

2.2.8 Overloading

In a large Z specification that introduces hundreds of new names, keeping all these names disjoint decreases the readibility when similar things need to carry different names. For example, suppose a function which delivers the set of free variables of different types of terms: naming this function $\mathsf{vars}_{\mathsf{EXPR}}$, $\mathsf{vars}_{\mathsf{CONSTR}}$ and so on, does not contributes to readibility.

We therefore allow the selective use of *ad-hoc overloading*. Actually, this overloading is implemented by using different LaTeX markup names for Z symbols which expand to the same representation.

2.2.9 Further Extensions

The *ESZ* type checker, developed in ESPRESS and used for processing the mathematics used in this thesis, adds some further extensions to the Z notation, which have been proposed to the Z ISO Panel:

- The order in which paragraphs appear in a specification is arbitrary. The only requirement is that the definition-use relation of paragraphs is acyclic. (A paragraph is in definition-use relation to another paragraph if it introduces a name that is referred to by the other paragraph.)

- Δ and Ξ are introduced as expression operators.

- Mutually recursive free types are allowed. This is (roughly) explained as follows. A free type definition, $T ::= \mathit{variants}$, introduces two paragraphs: one containing the definition of the given type $[T]$, and a second containing the free type axioms derived from the *variants*. Given the freedom of order of paragraphs, this allows arbitrary recursive dependencies between free type names and other definitions.

- Global constants can be declared multiple times, provided all declarations are type-compatible.

A detailed justification, including the semantic foundation, is given in [Fett and Grieskamp, 1998].

2.3 Mathematical Tools

Standard Z provides a rich set of "mathematical tools" for working with sets, relations, and functions. It is assumed that the reader is familiar with Z's mathematical toolkit (a specification can be found in [Spivey, 1992]). Some further tools are required in this thesis, and will be defined below.

2.3.1 Sets and Relations

The Z toolkit is fairly complete as regards basic functions on sets and relations. For our present purposes, it lacks only one concept: the *reduction* of a set of values by a binary function.

$\sum_e^f$ denotes a function that takes a set and reduces its elements using the function f and the starting element e. f must be right-commutative w.r.t. the argument set ($f(x_1, f(x_2, y)) = f(x_2, f(x_1, y))$) since the order in which reduction is performed is not determined:

$$[X, Y]$$
$$\sum_{_}^{_} : (X \times Y \rightarrow\!\!\!\!\!+ Y) \times Y \longrightarrow \mathbb{P}\, X \rightarrow\!\!\!\!\!+ Y$$
$$\sum_e^f s \Leftarrow (\mu\, y : Y \mid s = \varnothing \wedge y = e \vee (\forall\, x : s \bullet y = f(x, \sum_e^f (s \setminus \{x\}))))$$
$$\textbf{where } f : X \times Y \rightarrow\!\!\!\!\!+ Y;\ e : Y;\ s : \mathbb{P}\, X$$

This definition makes use of the syntactic extension for defining recursive partial functions (Section 2.2.4 (on page 29)). The domain of the function $\sum_e^f$ is induced by its defining expression, which is (for non-trivial s) only defined if f is right-commutative and defined for all elements.

2.3.2 Sequences

The Z toolkit lacks some typical functions known from functional languages for working with sequences, which are defined as follows. With $x \cdot\!: s$, the element x is prepended to the sequence s, with $s :\!\cdot\, x$ appended. By $\mathsf{zip}\,(s_1, s_2)$, the sequence is delivered where the elements are pairwise combined. The phrase $(f, e) \,/\, s$ denotes the reduction of the function f from right to left on the sequence s, with starting value e; the phrase $(f, e) \setminus s$ denotes the reduction from left to right.

$$[X]$$
$$_ \cdot\!: _ == \lambda\, x : X;\ s : \mathrm{seq}\, X \bullet \langle x \rangle \frown s$$
$$_ :\!\cdot\, _ == \lambda\, s : \mathrm{seq}\, X;\ x : X \bullet s \frown \langle x \rangle$$

$$[X, Y]$$
$$\mathsf{zip} : \mathrm{seq}\, X \times \mathrm{seq}\, Y \rightarrow\!\!\!\!\!+ \mathrm{seq}(X \times Y)$$
$$\mathsf{zip}\ (\langle\rangle, \langle\rangle) \Leftarrow \langle\rangle$$
$$\mid (x \cdot\!: s, y \cdot\!: t) \Leftarrow (x, y) \cdot\!: \mathsf{zip}\,(s, t)$$
$$\textbf{where } x : X;\ s : \mathrm{seq}\, X;\ y : Y;\ t : \mathrm{seq}\, Y$$

$$
\begin{array}{l}
[X, Y] \\
_ / _ : ((X \times Y \rightarrowtail\!\!\!\!\!\!\rightarrow Y) \times Y) \times \mathrm{seq}\, X \rightarrowtail\!\!\!\!\!\!\rightarrow Y \\
\hline
(_ / _)\ ((f, e), \langle\rangle) \quad \Leftarrow e \\
\quad \mid \quad ((f, e), x \cdot\!\!: s) \Leftarrow f(x, (f, e) / s) \\
\quad \mathbf{where}\ f : X \times Y \rightarrowtail\!\!\!\!\!\!\rightarrow Y;\ e : Y;\ x : X;\ s : \mathrm{seq}\, X
\end{array}
$$

$$
\begin{array}{l}
[X, Y] \\
_ \backslash _ : ((X \times Y \rightarrowtail\!\!\!\!\!\!\rightarrow Y) \times Y) \times \mathrm{seq}\, X \rightarrowtail\!\!\!\!\!\!\rightarrow Y \\
\hline
(_ \backslash _)\ ((f, e), \langle\rangle) \quad \Leftarrow e \\
\quad \mid \quad ((f, e), s :\!\cdot\, x) \Leftarrow f(x, (f, e) \backslash s) \\
\quad \mathbf{where}\ f : X \times Y \rightarrowtail\!\!\!\!\!\!\rightarrow Y;\ e : Y;\ x : X;\ s : \mathrm{seq}\, X
\end{array}
$$

With $f * (e, s)$, the "context mapping" of the function f to the elements in s from left to right is denoted. e is an environment or "state" parameter, which is threaded through the mapping:

$$
\begin{array}{l}
[X, Y, Z] \\
_ * _ : (X \times Y \rightarrowtail\!\!\!\!\!\!\rightarrow X \times Z) \times (X \times \mathrm{seq}\, Y) \rightarrowtail\!\!\!\!\!\!\rightarrow X \times \mathrm{seq}\, Z \\
\hline
(_ * _)\ (f, (e, \langle\rangle)) \quad \Leftarrow (e, \langle\rangle) \\
\quad \mid \quad (f, (e, x \cdot\!\!: s)) \Leftarrow \mathbf{let}\ e', e'' : X;\ x' : Z;\ s' : \mathrm{seq}\, Z \mid \\
\qquad\qquad\qquad\qquad (e', x') = f(e, x);\ (e'', s') = f * (e', s) \\
\qquad\qquad\qquad \bullet (e'', x' \cdot\!\!: s') \\
\quad \mathbf{where}\ f : X \times Y \rightarrowtail\!\!\!\!\!\!\rightarrow X \times Z;\ e : X;\ x : Y;\ s : \mathrm{seq}\, Y
\end{array}
$$

2.3.3 Booleans

In the forthcoming Z ISO Standard, schema expressions and value expressions have been unified. Moreover, schemas with an empty signature are allowed (for example, $[\mid P]$). This opens up the possibility of defining the type of Boolean values as the set of schemas to the empty signature, and use schema reference in a predicate to refer to a Boolean value. The type $\mathbb{B}$ is defined below following this approach:

$$\mathbb{B} == \mathbb{P}[]$$

$$
\begin{array}{|l}
\mathsf{true} == [\mid \mathit{true}] \\
\mathsf{false} == [\mid \mathit{false}]
\end{array}
$$

$$\mid \bigvee == \lambda\, s : \mathbb{P}\,\mathbb{B} \bullet [\mid \exists\, x : s \bullet x]$$

$$\mid \bigwedge == \lambda\, s : \mathbb{P}\,\mathbb{B} \bullet [\mid \forall\, x : s \bullet x]$$

With $\bigvee s$, the members of the set of booleans s are disjuncted, with $\bigwedge s$, they are conjuncted. Note that Booleans are truly two-valued, in contrast to truth values as discussed in Section 2.1.4 (on page 26).

2.3.4 Order and Lattices

Ordered sets and lattices provide a fundamental theory for computer science and discrete mathematics. Below, some basic notions that are required later on in this thesis are for-

malized in Z. The definitions and theorems are mostly taken from [Davey and Priestly, 1990].

An *ordered set* is a set equipped with a (partial) order, $_ \sqsubseteq _$, which is reflexive, antisymmetric and transitive:

$$
\begin{array}{l}
[X] \\
_ \sqsubseteq _ : X \leftrightarrow X \\
\hline
x \sqsubseteq x;\ x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y;\ x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq y \\
\quad \mathbf{where}\ x, y, z : X
\end{array}
$$

The axiom $(_ \sqsubseteq _)[A] = E$ is used throughout this thesis to state that "the" partial order of a given set A regarding lattice properties is equal to the expression E of type $A \leftrightarrow A$. For example, the following axiom states that the partial order of sets of some given element type *FOO* is set inclusion:

$$[FOO] \qquad (_ \sqsubseteq _)[\mathbb{P}\, FOO] = (_ \subseteq _)$$

The technique used for this kind of "overloaded" definition of $_ \sqsubseteq _$, where meanings are assigned in dependency of generic instances, is comparable to Haskell's type classes [Wadler and Blott, 1989; Hudak et al., 1992][3].

From the order of a set X, its *dual order* is derived by relational inversion:

$$
\begin{array}{l}
[X] \\
_ \sqsupseteq _ == (_ \sqsubseteq _)[X]^{\sim}
\end{array}
$$

Given an ordered set $S \subseteq X$, the *supremum* (least upper bound) and the *infimum* (greatest lower bound), if existent, are denoted by $\bigsqcup S$ ($\bigsqcap S$):

$$
\begin{array}{l}
[X] \\
\bigsqcup, \bigsqcap : \mathbb{P}\, X \rightarrowtail X \\
\hline
\bigsqcup\ S \Leftarrow \mathbf{let}\ U == \{x : X \mid \forall x' : S \bullet x \sqsupseteq x'\} \bullet (\mu\, x : U \mid \forall x' : U \bullet x \sqsubseteq x') \\
\quad \mathbf{where}\ S : \mathbb{P}\, X \\
\bigsqcap\ S \Leftarrow \mathbf{let}\ L == \{x : X \mid \forall x' : S \bullet x \sqsubseteq x'\} \bullet (\mu\, x : L \mid \forall x' : L \bullet x \sqsupseteq x') \\
\quad \mathbf{where}\ S : \mathbb{P}\, X
\end{array}
$$

To build the supremum (infimum) of two-element sets $\{x, x'\} \subseteq X$, the notation $x \sqcup x'$ ($x \sqcap x'$) is used. The former is called the *join*, the latter the *meet*:

[3] Standard Z, [Toyn, 1998], relaxes restrictions found in the Z of [Spivey, 1992] and permits such use of genericity. Semantically, specifying $_ \sqsubseteq _$ generically for all possible instantiations does not cause a problem: for any set A, there exists at least the empty order, $(_ \sqsubseteq _)[A] = \varnothing$.

$$[X]$$
$$_\sqcup_, _\sqcap_ : X \times X \nrightarrow X$$
$$(_\sqcup_)\ (x, x') \Leftarrow \sqcup\{x, x'\} \quad \textbf{where } x, x' : X$$
$$(_\sqcap_)\ (x, x') \Leftarrow \sqcap\{x, x'\} \quad \textbf{where } x, x' : X$$

A *lattice* is a set $L \subseteq X$, where for all pairs $x, x' \in L$, $x \sqcup x'$ and $x \sqcap x'$ are defined. The set of all subsets L of X which are a lattice is described by $\mathsf{lat}\,X$. In a *complete lattice*, for each subset $L' \subseteq L$ the supremum (infimum) exists. $\sqcup L$ ($\sqcap L$) are called the *top* (*bottom*) element of a complete lattice L:

$$[X]$$
$$\mathsf{lat}\,_ == \{L : \mathbb{P}\,X \mid \forall x, x' : L \bullet (x, x') \in \mathsf{dom}(_\sqcup_) \land (x, x') \in \mathsf{dom}(_\sqcap_)\}$$

$$[X]$$
$$\mathsf{lat_C}\,_ == \{L : \mathsf{lat}\,X \mid \forall L' : \mathbb{P}\,L \bullet L' \in \mathsf{dom}\,\sqcup \land L' \in \mathsf{dom}\,\sqcap\}$$

$$[X]$$
$$\top_{_} == \lambda L : \mathsf{lat_C}\,X \bullet \sqcup L$$
$$\bot_{_} == \lambda L : \mathsf{lat_C}\,X \bullet \sqcap L$$

A well-known structure related to a complete lattice is a *cpo* (complete partial order). It has a bottom element and behaves like a complete lattice as regards the infimum. For the dual, it is only demanded that for every directed subset of S, the supremum exists. A *directed set* is one where for any finite subset the upper bound is in the set. Cpos are commonly used for modeling the semantics of programming languages; however, for our set-based calculus, the more general notion of lattices is better suited.

For complete lattices (and cpos), theorems exist that can characterize *computability*. These will be briefly reviewed below. Let $f : L \longrightarrow L$ be a function on a complete lattice L. The set of *fixed-points* of f is defined as the set $\{x : L \mid x = f\,x\}$. The *least fixed-point* exists if the infimum of this set exists (the set is not empty) and is a fixed-point (is in the set). It is known that if f is *order-preserving* (monotonic), then the least fixed-point exists for complete lattices. If f is also *continuous*, then the least fixed-point can be expressed by the image of f under the supremum of the set of *finite elements* in the domain of f. Continuity in this context means that for any subset $S \subseteq L$, it holds $\sqcup(f(\!|S|\!)) = f(\sqcup S)$. We denote the set of continuous functions over a base type X as $X \mathrel{\ominus\!\!\rightarrow} X$:

$$[X, Y]$$
$$_ \mathrel{\ominus\!\!\rightarrow} _ == \{f : X \nrightarrow Y \mid \mathsf{dom}\,f \in \mathsf{lat_C}\,X \land \mathsf{ran}\,f \in \mathsf{lat_C}\,Y$$
$$(\forall S : \mathbb{P}(\mathsf{dom}\,f) \bullet \sqcup(f(\!|S|\!)) = f(\sqcup S))\}$$

A *finite element* of a complete lattice is one where there exists only a finite number of

elements in a chain starting at $\perp$ and reaching the element. Let $F \subseteq \mathsf{dom} f$ be the non-empty set of all finite elements in the domain of f. Then, the central fixed-point theorem for continuous functions and complete lattices states:

$$f \in X \twoheadrightarrow X \Rightarrow \sqcap \{x : \mathsf{dom} f \mid x = f\,x\} = f(\sqcup F)$$

Thus, putting the supremum of maximal finite information into f yields the (possibly, infinite) least fixed-point. Since (by continuity) f is order preserving, this (indeed "ideal") maximal finite information can be incrementally computed by successive application of f starting at $\perp_L$. These results are used in Chapter 3 to justify the fixed-point law of μZ.

So gestaltend, umgestaltend –
Zum Erstaunen bin ich da.
(Goethe)

Chapter 3

The μZ Calculus

The μZ calculus is a set-based expression language which can embed set-oriented notations such as Z as well as functional and logical programming languages. The calculus is designed for the automatic analysis and transformation, where it is desirable working with a *small* set of language constructs. However, μZ is not "as small as possible"[1]. The calculus has redundancy in that it allows *extensional* as well as *intensional* description of the same meaning. For example, the set containing the numbers 1 and 2 can be denoted in μZ as $\{1\} \cup \{2\}$ or by the schema $\{x \mid x = 1 \cup x = 2\}$. The support of both intensional and extensional description allows the definition of computation as a source-level transformation from intensional towards extensional representation, as shown in the next chapter.

The language constructs of μZ are essentially those of set algebra (set intersection, union and complement), set comprehension and a novel notation for *set translation*. Moreover, singleton sets can be constructed and deconstructed. The calculus entails a notion of free constructors, which are used to model tuples, Z-style bindings and free types. As will be seen, the λ-calculus, predicate calculus and schema calculus can be derived from this basic set of operators.

In this chapter, μZ will be introduced and formally defined. A semantic model is developed, and the language constructs will be described one-by-one, giving their typing rules and semantic meanings. An equational theory for μZ is developed. The chapter is concluded with a discussion of some aspects of the calculus and of related work.

[1]Higher-order logic [Gordon and Melham, 1993], for instance, has a smaller number of language constructs.

3.1 Introduction to μZ

We give an introduction to the μZ calculus and discuss the encoding of Z as well as logic and functional logic programming languages in μZ.

3.1.1 Syntax and Informal Meaning

Let $x \in \mathit{VAR}$ be variable symbols, and let $\rho \in \mathit{CONS}$ be (term) constructor symbols. The μZ calculus provides the syntactic categories of *expressions*, and a subset of them, *patterns*:

$$\begin{array}{lll} e \in \mathit{EXP} & ::= & x \mid \rho(\overline{e}) \mid \{e\} \mid \mu\, e \mid \mathsf{0} \mid e \cap e' \mid e \cup e' \mid \sim e \mid \\ & & e[p_1 \mapsto p_2] \mid \{p \mid e\} \mid \mathbf{fix}\, p \lhd e \\ p \in \mathit{PAT} & \subseteq & \mathit{EXP} \end{array}$$

The form $\rho(\overline{e})$ describes a constructor application, where $\overline{e}$ is a sequence of expressions. (We generally use the convention of denoting sequences of terms t by $\overline{t}$.) The set of patterns $\mathit{PAT} \subseteq \mathit{EXP}$ are those expressions that are solely built from constructor application to patterns, $\rho(\overline{p})$, and variables, x.

The form $\{e\}$ denotes a singleton set. The form $\mu\, e$ is the singleton set selection that returns the element of a singleton set, and is undefined for each other set. The usual forms for set algebra are provided: the empty set is denoted by $\mathsf{0}$, set intersection by $e \cap e'$, set union by $e \cup e'$ and set complement by $\sim e$.

The form $e[p_1 \mapsto p_2]$ describes the *translation* of the set e. The result of the translation is the set of all elements that match p_2 such that there exists an element in e which matches the pattern p_1 under the same substitution. Thereby, both patterns may contain unbound variables, which may be denoted by wildcards ($_$). For example. For example, $(\{\rho(1,2)\} \cup \{\rho(1,3)\})[\rho(x,_) \mapsto x]$ (where numbers are special 0-ary constructor symbols) equals to $\{1\}$. The cartesian product of two sets, $e_1 \times e_2$, can be expressed in μZ as $e_1[x \mapsto (x,_)] \cap e_2[y \mapsto (_,y)]$ ($(_,_)$ being the constructor symbol for pairs used in mixfix notation). Conversely, the range of a binary relation e (a set of pairs) is denoted as $e[(_,y) \mapsto y]$. We call a translation with $\mathsf{vars}\, p_1 \subseteq \mathsf{vars}\, p_2$ an *embedding* and with $\mathsf{vars}\, p_1 \supset \mathsf{vars}\, p_2$ a *hiding*. Note that there might exist translations which are neither embeddings nor hidings. In general, translations are capable of expressing all kinds of specification morphisms induced by signature morphisms as used in algebraic specification [Ehrig and Mahr, 1985; Wirsing, 1990][2], and thus can be seen as a means for constructive description of (some) morphisms in the category of μZ sets.

The form $\{p \mid e\}$ describes a set comprehension, or a *schema* in Z terminology[3]. It denotes the set of values that match the pattern p such that the Boolean expression e is true under the binding of variables resulting from the match. Thereby, a Boolean expression is just a set expression over the element type of units (a type containing just one element): let $\diamond$ be the 0-ary constructor for this element: then truth is the set $\{\diamond\}$, falsity is the

[2] It is easy to see that sets of tuples or bindings (tuples with named components) correlate to sets of models/algebras.

[3] We prefer the notion "schema" since the structure of the generating pattern, p, is preserved, whereas in Z's set comprehension p is reduced to its characteristic tuple.

empty set, conjunction is intersection, disjunction union, and negation complement. For example, the (sugared) schema $\{(x,y,z) \mid x < y \cap y < z\}$ describes the set of triples where each member is larger than the previous one. The Boolean expression $x < y$ is syntactic sugar for the form $((_ < _) \cap \{(x,y)\})[_ \mapsto \diamond]$ (where $_ < _$ is a binary relation, a set of pairs). The conclusive translation can be read as: we are not interested in the actual result of the intersection, we merely want to know if it is empty or not. The entire schema literally reads as

$$\{(x,y,z) \mid ((_ < _) \cap \{(x,y)\})[_ \mapsto \diamond] \;\cap\; ((_ < _) \cap \{(y,z)\})[_ \mapsto \diamond]\}.$$

The form $\mathbf{fix}\, p \lhd e$ denotes the smallest member of the set $\{p \mid p = e\}$ (where the equality abbreviates $(\{p\} \cap \{e\})[_ \mapsto \diamond]$). The order needed here is discussed in Section 3.2.6 (on page 48). If a smallest member does not exist, the form is undefined.

The μZ calculus uses a shallow polymorphic type system. To this end, constructors ρ have associated a (generic) type, and expressions are assumed to have a principal type. A further condition for well-formedness is that patterns in translations and schemas are used linearly (no variable is allowed to appear twice).

For commonly used forms of expressions, syntactic sugar is used, the most important of which is the following:

$$
\begin{array}{lcl}
\mathsf{1} & \leadsto & \sim \mathsf{0} \\
?_1 e & \leadsto & e[x \mapsto \diamond] \\
e \in e' & \leadsto & ?_1(\{e\} \cap e') \\
e = e' & \leadsto & e \in \{e'\}
\end{array}
$$

$\mathsf{1}$ is the “universal set”, the largest set of its according type domain. $?_1 e$ is an abbreviation for the Boolean expression which is true iff the set e is nonempty. The two other forms represent membership and equality.

3.1.2 Encodings

Constructs such as function application, λ-abstraction, membership-test, equality, existential and universal quantification can be naturally encoded in the μZ calculus. We demonstrate this by sketching the encoding of Z in μZ. A brief comparison with logic programming languages (horn-clause systems) and with the functional logic language Curry [Hanus, 1999] is also given, anticipating some basic ideas underlying the computation model of μZ.

Encoding Z

Lambda Calculus. The operations of the simple-typed λ-calculus are encoded as follows:

$$
\begin{array}{lll}
\lambda p \bullet e & \leadsto \; \{(p,y) \mid y = e\} & \qquad (y \notin vars\, e) \\
e\, e' & \leadsto \; \mu((e \cap \{e'\}[x \mapsto (x, _)])[(_, y) \mapsto y]) &
\end{array}
$$

This treats functions as *sets of pairs*. The encoding of the application does in fact also capture relational application as used in Z: it is sufficient for e to be a binary relation that is right-unique and defined at the point e'.

Predicate Calculus. We have already noted that formulas are a special case of set expressions over the element type of units, and we have defined abbreviations for the basic predicates of membership-test and equality. To complete the picture, we give the encoding of quantifiers:

$$\exists x \bullet e \leadsto ?_1\{x \mid e\}$$
$$\forall x \bullet e \leadsto \sim(?_1\{x \mid \sim e\})$$

The resulting semantics of the encoding of predicate calculus is a partial three-valued logic (see Section 3.2.1 (on page 43)).

Schema Calculus. In Z, a schema denotes a set of *bindings*, bindings being tuples (records) with named components. The set is constrained by the declarations of a schema as well as by the property. For example, in Z, the schema $[a : \mathbb{N};\ b : \mathbb{Z} \mid a < b]$ denotes the set of bindings $\langle\!| a == x, b == y |\!\rangle$, where $x \in \mathbb{N}$, $y \in \mathbb{Z}$, and $x < y$.

Assuming μZ constructors for bindings, $\langle\!| a_1 == _, \ldots, a_n == _ |\!\rangle$, where the a_i's are interpreted as part of the constructor name, the above Z schema can be encoded in μZ by

$$\{\langle\!| a == x, b == y |\!\rangle \mid x \in \mathbb{N} \cap x < y\}$$

The constraint $y \in \mathbb{Z}$ vanishes since $\mathbb{Z}$ is a given type. The property $x < y$ is further translated as described in the previous section. Other possible encodings exist, for example

$$\mathbb{N}[x \mapsto \langle\!| a == x, b == _ |\!\rangle] \cap \{\langle\!| a == x, b == y |\!\rangle \mid x < y\}$$

The connectivities of the schema calculus directly map to μZ. Schema conjunction in Z, $S_1 \wedge S_2$, is the set of bindings with the joined signature of S_1 and S_2 where components with the same name are identified. This is expressed in μZ as

$$S_1[\Sigma(S_1) \mapsto \Sigma] \cap S_2[\Sigma(S_2) \mapsto \Sigma],$$

where Σ is the pattern for the joined signature. For example, the Z schema conjunction $[a : A;\ b : B \mid P[a, b]] \wedge [a : A;\ c : C \mid Q[a, c]]$ (where A, B and C are given types) maps to

$$\{\langle\!| a == x, b == y |\!\rangle \mid P[x, y]\}[\langle\!| a == x, b == y |\!\rangle \mapsto \langle\!| a == x, b == y, c == _ |\!\rangle] \cap$$
$$\{\langle\!| a == x, c == y |\!\rangle \mid Q[x, y]\}[\langle\!| a == x, c == y |\!\rangle \mapsto \langle\!| a == x, b == _, c == y |\!\rangle]$$

Existential schema quantification, $\exists S \bullet S'$, is encoded as

$$(S[\Sigma(S) \mapsto \Sigma(S')] \cap S')[\Sigma(S') \mapsto \Sigma(S') \setminus \Sigma(S)]$$

where $\Sigma(S') \setminus \Sigma(S)$ denotes the removal of the components from S in S'. Similarly, the remaining connectivities of the schema calculus are encoded.

Note that μZ's support of the schema calculus clearly shows that μZ is capable to express relations between entire specification units, *modules*, etc.: Z schemas can be thought of as an algebra on the calculus level.

Encoding Logic Languages

A logic program in a language such as Prolog consists of a set of Horn clauses $A \leftarrow B_1, \ldots, B_n$, where A is called the *conclusion* and the B_i are the *premises* of the clause. The A and B_i are so-called atoms of the form $R(\bar{t})$, where $\bar{t}$ is a sequence of terms over constructor symbols and variables, and R is a n-ary relation. We can map Horn clause systems to μZ as follows. Let $A = R(\bar{t})$ and $B_i = Q_i(\overline{s_i})$; then the "body" of the clause is represented as $\{(\bar{t}) \mid (\overline{s_1}) \in Q_1 \cap \ldots \cap (\overline{s_n}) \in Q_n\}$. To define the meaning of the relation R, we collect the bodies of all clauses in which R is the head of the premise, as follows:

$$
\begin{aligned}
R = {} & \{(\overline{t_1}) \mid (\overline{s_{1_1}}) \in Q_{1_1} \cap \ldots \cap (\overline{s_{n_1}}) \in Q_{n_1}\} \\
& \cup \\
& \ldots \\
& \cup \\
& \{(\overline{t_m}) \mid (\overline{s_{1_m}}) \in Q_{1_m} \cap \ldots \cap (\overline{s_{n_m}}) \in Q_{n_m}\}
\end{aligned}
$$

We proceed similarly with the clauses for Q_{i_j} in the system. If R and Q_{i_j} are recursively dependend, the fixed-point operator will be used. Answering a query $R(\overline{q})$ then merely means computing the intersection $R \cap \{(\overline{q})\}$.

Encoding Functional Logic Languages

Languages such as Curry [Hanus, 1999] or Goffin [Chakravarty et al., 1997] can, in principle, be encoded in μZ. The exact operational semantics will not be the same, since a strict reduction order for μZ is preferred by design, whereas both Curry and Goffin, which are extensions of Haskell, use lazy reduction. An illustration of such an encoding is instructive, anyway.

A function definition in Curry is mapped as follows (f being a function symbol, p a pattern, c a constraint, and e an expression; the fresh variables y_i may not appear in the patterns p_i):

$$
\begin{aligned}
& f\, p_1 \mid c_1 \;= e_1;\; \ldots\; ;\, f\, p_n \mid c_n \;= e_n \;\rightsquigarrow \\
& \qquad \mathbf{fix}\, f \triangleleft \{(p_1, y_1) \mid c_1 \cap y_1 = e_1\} \cup \ldots \cup \{(p_n, y_n) \mid c_1 \cap y_n = e_n\}
\end{aligned}
$$

Thus, a constraint c of Curry directly maps to a corresponding constraint in μZ. Conjunctions of constraints are expressed by intersection. Calling a function that returns a constraint with a free variable as the argument, such as $C\, x$, amounts to membership test, $x \in C$. In fact, μZ allows a richer constraint language than Curry (in particular, disjunction and negation), which, however, will not always be executable.

The computation model of μZ, described later on, executes constraints concurrently, using both residuation [Ait-Kaci et al., 1987] and backtracking and enumeration to deal with free variables. An access to a free variable is suspended in a nested context. This is, for example, the case for arguments of the μ -operator (and therefore for function application $e\, e'$ which is defined using μ) or for negative constraints. Backtracking takes place for constraints which map to the membership test of a pattern in a set. For example, consider the definition of an "append" relation on lists (below, an abbreviation for introducing local variables is used: $\{p \setminus q \mid e\}$ is equivalent to $\{(p, q) \mid e\}[(p, q) \mapsto p]$):

$$\begin{aligned}\textbf{fix}\ app \triangleleft\ & \{(\langle\rangle, ys, zs) \mid zs = ys\} \cup \\ & \{(x :: xs, ys, zs) \setminus t \mid (xs, ys, t) \in app \cap zs = x :: t\}\end{aligned}$$

Applying this relation to unbound *xs* and *ys* and bound *zs* will compute the possible partitions of the list *zs*, as usual in logic languages.

μZ is higher-order because sets can be stored in constructed values. Similar as Curry or Goffin, the computation model does not employ higher-order unification. This restriction is, though, not tied to the notion of a type of a value (whether it is a set or not) but to the representation: sets that are *extensionally* represented are incorporated by a simple equality test in unification. An extensional representation is given by applying constructors, singleton sets and set union to extensional values.

3.2 Semantic Model

The model of μZ is constructed by an interpretation function over terms of a type language. We will first discuss the concepts underlying the set representation in the model and then define the type interpretation.

3.2.1 Set Representation

The model of μZ is based on a hierarchical typed universe, where sets are represented as *partial characteristic (Boolean) functions* in a set-theoretical meaning, which can be founded by Z itself or by Zermelo-Fraenklin set theory. We will use Z to this end.

Let U be the domain of some type. The representation of an element of the powerset of this type is a partial function $f \in U \pfun \mathbb{B}$. The function f characterizes the knowledge about the elements in the set in the following way. A value $u \in U$ is

1. a member of the set, if $u \in \mathsf{dom} f \wedge f\, u$;
2. not a member of the set, if $u \in \mathsf{dom} f \wedge \neg f\, u$;
3. unknown to be a member or not, if $u \notin \mathsf{dom} f$.

As an example, consider the (sugared) μZ schema $\{x \mid 1 \;\mathsf{div}\; x = 1\}$. For this set, 1 is known to be member, and all natural numbers greater then 1 are known *not* to be members; however, for 0, no information is available, since the expression $1 \,\mathsf{div}\, 0$ is undefined. Thus this set is represented as the Boolean function

$$\lambda\, x : \mathbb{N} \setminus \{0\} \bullet [\!| \; x = 1].$$

The given representation of sets allows for a natural definition of the set-algebraic operations. Let f_1, f_2 be the representations of two sets. The intersection is defined as

$$f_1 \cap f_2 \equiv \lambda\, x : (\mathsf{dom} f_1 \cap \mathsf{dom} f_2) \cup (f_1^{\sim}) (\!| \{\mathsf{false}\} |\!) \cup (f_2^{\sim}) (\!| \{\mathsf{false}\} |\!) \bullet [\!| \; f_1\, x \wedge f_2\, x],$$

and the union is defined as

$$f_1 \cup f_2 \equiv \lambda\, x : (\mathsf{dom} f_1 \cap \mathsf{dom} f_2) \cup (f_1^{\sim}) (\!| \{\mathsf{true}\} |\!) \cup (f_2^{\sim}) (\!| \{\mathsf{true}\} |\!) \bullet [\!| \; f_1\, x \vee f_2\, x].$$

Thereby, the conjuction in our meta-language Z, $\wedge$, is defined if one of the operands is false, and disjunction, $\vee$, if one of the operands is true, independent of the definedness of the other operand. Accordingly, the domains of the resulting characteristic functions are constructed: we add all those elements to the domain for which the one or the other operand already determines the result.

As an example for the effect of the set operators, consider the expression $\{x \mid 1 \,\mathsf{div}\, x = 1\} \cap \{x \mid x \neq 0\}$. The representation of the left set operand was already defined above. The right set operand is given by the function $\lambda\, x : \mathbb{N} \bullet [\!| \; x \neq 0]$. The intersection of both representations according to the above definition yields the function

$$\lambda\, x : \mathbb{N} \bullet [\!| \; 1 \;\mathsf{div}\; x = 1 \wedge x \neq 0].$$

$$[TCONS, TVAR]$$

$$|\ \mathsf{arity} : TCONS \longrightarrow \mathbb{N}$$

$$\begin{array}{lll} TYPE ::= & (_[_]) & \langle\!\langle \{\rho : TCONS;\ \overline{\tau} : \mathrm{seq}\, TYPE \mid \mathsf{arity}\,\rho = \#\overline{\tau}\} \rangle\!\rangle \\ \mid & \mathbb{P} & \langle\!\langle TYPE \rangle\!\rangle \\ \mid & (_) & \langle\!\langle TVAR \rangle\!\rangle \end{array}$$

Figure 3.1: Type Language

Whereas the left-operand is not defined for 0, the result is (with 0 known not to be member of the resulting set).

An interesting aspect is the definition of set complement in our set representation. It is straight-forward:

$$\sim f \equiv \lambda\, x : \mathsf{dom} f \bullet \neg\, f\, x.$$

Thus the complement does not alter the domain of the characteristic function. What is unknown to be member of the set keeps to be unknown. As a consequence, the "principle of the excluded middle" will not hold in our framework: $f \cap \sim f \equiv \emptyset$ does *not* hold. However, $\sim\ \sim f \equiv f$ holds. The valid laws will be discussed in more detail in Section 3.4 (on page 57).

The representation of sets and the definition of the set-operations provides a generalization of what is called "three-valued" logic in the literature to the case of set algebra. We have already discussed that truth-vales are represented as sets over the "unit" type with one member, denoted by the constructor $\diamond$. The empty set, $\emptyset$, represents falsity, and the singleton set containing the unit, $\{\diamond\}$, truth. Intersection, union, and complement over sets of this type behave as conjunction, disjunction, and negation in the three-valued logic described by Owe [1997].

3.2.2 Types

The model of μZ is constructed by an interpretation function over terms of a *type language*. Terms of the type language, $\tau \in TYPE$, are built from type construction $\rho[\overline{\tau}]$ (where $\rho \in TCONS$ are type constructors with arity), from power-set types $\mathbb{P}\,\tau$, and from type variable application, α ($\alpha \in TVAR$). The definitions are given in Figure 3.1. Note that the Cartesian product is introduced by certain type constructors, which will be discussed in Section 3.2.4 (on page 46).

We suppose standard syntactic tools on type terms (Figure 3.2 (on the facing page)). The set of free type variables in a type is delivered by $\mathsf{vars}\,\tau$. The subset of types which are ground (contain no type variables) is denoted with $TYPE_{\emptyset}$. A type variable substitution is a partial function $\sigma \in TSUBS = TVAR \nrightarrow TYPE$, such that $\mathsf{dom}\,\sigma \cap \bigcup(\mathsf{vars}\,(\!|\mathsf{ran}\,\sigma|\!)) = \emptyset$. The application of a substitution to a type is denoted as $\mathsf{subs}\,\sigma\,\tau$. $TSUBS_{\emptyset}$ denotes the set of ground substitutions. The most general unifier of two types is described by the relation $\tau_1 =^{\sigma} \tau_2$. These standard concepts are not formalized.

$$TYPE_{\varnothing} : \mathbb{P}\, TYPE$$

$$TSUBS == TVAR \rightarrow\!\!\!\!\!\leftrightarrow TYPE$$
$$TSUBS_{\varnothing} == TVAR \rightarrow\!\!\!\!\!\leftrightarrow TYPE_{\varnothing}$$

$$\begin{array}{l} \mathsf{vars} : TYPE \longrightarrow \mathbb{P}\, TVAR \\ \mathsf{subs} : TSUBS \longrightarrow TYPE \longrightarrow TYPE \\ _ =^{-} _ : \mathbb{P}(TYPE \times TSUBS \times TYPE) \end{array}$$

Figure 3.2: Type Language: Syntactic Tools

$$[UNIV]$$

$$\begin{array}{l} \mathsf{undef} : UNIV \\ \mathsf{sem} : TYPE_{\varnothing} \longrightarrow \mathbb{P}\, UNIV \\ \hline \forall\, \tau : TYPE_{\varnothing} \bullet \mathsf{undef} \in \mathsf{sem}\, \tau \\ (\lambda\, \tau : TYPE_{\varnothing} \bullet \mathsf{sem}\, \tau \setminus \{\mathsf{undef}\})\ \mathsf{partition}\ UNIV \setminus \{\mathsf{undef}\} \end{array}$$

$$\begin{array}{l} \uparrow_{\mathcal{S}} : (UNIV \rightarrow\!\!\!\!\!\leftrightarrow \mathbb{B}) \rightarrowtail\!\!\!\!\!\leftrightarrow UNIV \\ \downarrow_{\mathcal{S}} : UNIV \rightarrow\!\!\!\!\!\leftrightarrow (UNIV \rightarrow\!\!\!\!\!\leftrightarrow \mathbb{B}) \\ \hline \forall\, \tau : TYPE_{\varnothing} \bullet \mathsf{sem}\ \tau \rightarrow\!\!\!\!\!\leftrightarrow \mathbb{B} \subseteq \mathsf{dom} \uparrow_{\mathcal{S}} \\ \downarrow_{\mathcal{S}} = \uparrow_{\mathcal{S}}{}^{\sim} \end{array}$$

$$\begin{array}{l} \mathsf{sem}\ (\mathbb{P}\ \tau) \Leftarrow \{r : \mathsf{sem}\, \tau \rightarrow\!\!\!\!\!\leftrightarrow \mathbb{B} \bullet \uparrow_{\mathcal{S}} r\} \\ \quad \textbf{where}\ \tau : TYPE_{\varnothing} \end{array}$$

$$\mathsf{undef} = \uparrow_{\mathcal{S}} \varnothing$$

Figure 3.3: Semantic Universe

3.2.3 Universe

The *semantic universe* is denoted by the Z type $UNIV$. It contains a special element, undef, representing undefinedness. The universe is partitioned by the *type interpretation* $\mathsf{sem}\ \tau$ over ground types, *modulo* the element for undefinedness, which is member of each interpretation. The definitions are given in Figure 3.3.

The following remarks on the semantic interpretation sem in Figure 3.3:

- For type construction, the partitioning axiom already states everything required: a type construction $\rho[\overline{\tau}]$ delivers a subset of the universe disjoint from any other type interpretation, except that the element for undefinedness, undef, is shared. More constraints on certain type constructors are added in Section 3.2.4 (on the next page), letting their interpretation being *generated* by value constructors.

- As discussed above, a powerset type, $\mathbb{P}\ \tau$, denotes all the partial Boolean functions over the domain of the meaning of τ, $\mathsf{sem}\ \tau \rightarrow\!\!\!\!\!\leftrightarrow \mathbb{B}$. To map this representation into $UNIV$, we define an encoding function $\uparrow_{\mathcal{S}}$ as a partial injection which is defined for values which can be generated from a ground type. The inductive definition of the domain of $\uparrow_{\mathcal{S}}$ ensures that an encoding exists, and contradictions such as Russel's Antinomy are avoided.

The same value for undefinedness is shared since by the interpretation of powerset

$$
\begin{array}{l}
[CONS] \\
\mid \mathsf{ftype} : TCONS \rightarrowtail\!\!\!\!\!\!+ \; \mathrm{seq}\, TVAR \times (CONS \rightarrowtail\!\!\!\!\!\!+ \; \mathrm{seq}\, TYPE) \\
\hline
\mathsf{ftype}\, \rho \in \{ \overline{\alpha} : \mathrm{seq}\, TVAR;\ \mathsf{c} : CONS \rightarrowtail\!\!\!\!\!\!+ \; \mathrm{seq}\, TYPE \mid \\
\qquad \#\overline{\alpha} = \mathsf{arity}\, \rho \wedge (\forall \rho : \mathrm{dom}\, \mathsf{c} \bullet \bigcup(\mathsf{vars}\, (\!| \mathrm{ran}(\mathsf{c}\, \rho) |\!)) \subseteq \mathrm{ran}\, \overline{\alpha})\} \\
\quad \textbf{where}\ \rho : \mathrm{dom}\, \mathsf{ftype} \\
(\lambda \rho : \mathrm{dom}\, \mathsf{ftype} \bullet \mathrm{dom}(\mathsf{ftype}\, \rho).2)\ \mathrm{partition}\ CONS \\
\mid \mathsf{type}, \mathsf{tinst} : CONS \longrightarrow \mathrm{seq}\, TYPE \times TYPE
\end{array}
$$

Figure 3.4: Free Types: Syntax

types the totally undefined Boolean function, $\varnothing = \varnothing \longrightarrow \mathbb{B}$, which is identified with undef in a power-set domain, cannot be distinguished. Reusing undef over different types causes no problems, since only typed expressions are considered in the μZ calculus.

3.2.4 Free Types

The type constructors represent by default abstract, "given" subsets of the universe. For some type constructors, this set may be generated by a collection of *value constructors*, a commonly so-called free-type construction. A value constructor is a total injective mapping from a (possibly empty) sequence of values into a given value. A system of value constructors effectively describes a many-sorted term algebra embedded into the model of μZ.

The function ftype (Figure 3.4) associates a free type definition with certain type constructors, which consists of a sequence of type variables matching the type constructor's arity, and a mapping from constructor symbols to argument types which are closed under the type variables. The given set of constructors, *CONS*, is partitioned by the constructors of the free types.

For convenience, we introduce an auxiliary function which delivers the type of a value constructor, $\mathsf{type}\, \rho = \overline{\tau} \mapsto \rho[\overline{\alpha}] \in \mathrm{seq}\, TYPE \times TYPE$. Furthermore, we assume there is a function $\mathsf{tinst}\, \rho$, which delivers the type of a constructor with "fresh" type variables.

The semantics of a value constructor is described by the function $\mathsf{sem}\, \sigma\, \rho \in \mathrm{seq}\, UNIV \rightarrowtail\!\!\!\!\!\!+ \; UNIV$, which delivers for a ground type substitution and constructor a total injection w.r.t. the meaning of the constructors argument types into the type constructor's domain. Given this function, the domain of a type constructor is partitioned by the range of the meaning of all its value constructors (Figure 3.5). We have formally stated freeness (constructors are injective) and generation (by the partitioning). It is further demanded that the generated type's domain, $\mathsf{sem}\, (\mathsf{subs}\, \sigma\, (\rho[\overline{\tau}]))$ is the *smallest* which satisfies the above property. A formalization is left open.

There may be free type configurations which are inconsistent: consider a constructor with type $\langle \mathbb{P}\, \rho \rangle \mapsto \rho$. There is no solution by the above axioms for such a constructor, and therefore it is not allowed. In Z one can further restrict the domain of constructor functions, such that (in the above notation) one gets e.g. a type such as $\langle \mathbb{F}\, \rho \rangle \mapsto \rho$ for a constructor, where $\mathbb{F}\, \rho$ is the set of *finite* subsets of ρ. Thus Z's solution to the problem is to use *partial* injective constructor functions regarding the basic type of the arguments

$$
\begin{array}{l}
\mathsf{sem} : \mathit{TSUBS}_{\varnothing} \longrightarrow \mathit{CONS} \nrightarrow \mathrm{seq}\,\mathit{UNIV} \rightarrowtail\!\!\!\!\rightarrow \mathit{UNIV} \\
\hline
(\forall \rho : \mathsf{dom}\,\mathsf{c} \bullet \\
\quad \rho \in \mathsf{dom}(\mathsf{sem}\,\sigma) \wedge \\
\quad \mathsf{sem}\,\sigma\,\rho \in \{\overline{u} : \mathrm{seq}\,\mathit{UNIV} \mid \#\overline{u} = \#(\mathsf{c}\,\rho) \wedge \\
\qquad\qquad (\forall i : \mathsf{dom}\,\overline{u} \bullet \overline{u}\,i \in \mathsf{sem}\,(\mathsf{subs}\,\sigma(\mathsf{c}\,\rho\,i)))\} \\
\qquad\qquad \rightarrowtail (\mathsf{sem}\,(\mathsf{subs}\,\sigma\,(\rho[\overline{\tau}]))) \setminus \{\mathsf{undef}\}) \\
(\lambda \rho : \mathsf{dom}\,\mathsf{c} \bullet \mathsf{ran}(\mathsf{sem}\,\sigma\rho))\ \mathsf{partition}\ \mathsf{sem}\,(\mathsf{subs}\,\sigma\,(\rho[\overline{\tau}])) \setminus \{\mathsf{undef}\} \\
\quad \mathbf{where}\ \rho : \mathsf{dom}\,\mathsf{ftype};\ \sigma : \mathit{TSUBS}_{\varnothing};\ \overline{\alpha} : \mathrm{seq}\,\mathit{TVAR} \\
\qquad \overline{\tau} : \mathrm{seq}\,\mathit{TYPE};\ \mathsf{c} : \mathit{CONS} \nrightarrow \mathrm{seq}\,\mathit{TYPE} \mid \\
\qquad\quad (\overline{\alpha}, \mathsf{c}) = \mathsf{ftype}\,\rho;\ \mathsf{ran}\,\overline{\alpha} \subseteq \mathsf{dom}\,\sigma;\ \overline{\tau} = (_) \circ \overline{\alpha}
\end{array}
$$

Figure 3.5: Free Types: Meaning

$$
\begin{array}{l}
\langle\rangle : \mathit{TCONS}
\end{array}
\qquad
\begin{array}{l}
\diamond : \mathit{CONS} \\
\hline
\mathsf{ftype}\,\langle\rangle = (\varnothing, \{\diamond \mapsto \langle\rangle\})
\end{array}
$$

$$
\mathbb{B} == \mathbb{P}\,\langle\rangle;\quad \mathsf{tt} == \lambda\, u : \mathsf{sem}\,\langle\rangle \bullet \mathsf{true};\quad \mathsf{ff} == \lambda\, u : \mathsf{sem}\,\langle\rangle \bullet \mathsf{false}
$$

Figure 3.6: Free Type of Booleans

(the basic type of $\mathbb{F}\,\rho$ is $\mathbb{P}\,\rho$). Since a clean equational theory for μZ heavily depends on the fact that value constructors are total, this approach is not feasible for μZ. Instead, a solution to the problem is to encode constraints on constructor arguments in special given types (thus $\mathbb{F}\,\rho$ would be a given type), and provide primitive (partial) encoding and decoding functions which interpret these types. This approach is, however, not detailed here.

3.2.5 Standard Types and Constructors

Some standard types and associated constructors with sugared notation will be defined below.

Units and Booleans. The unit type, $\langle\rangle$, contains exactly one element, denoted by the 0-ary constructor $\diamond$ (Figure 3.6). The type of Booleans is a powerset of the unit type, and is abbreviated as $\mathbb{B}$. The semantic values tt and ff are the characteristic functions mapping the unit value to true and false, respectively.

Natural Numbers. The type of natural numbers is available. It is generated by the countable set of 0-ary constructors $0, 1, \ldots$, denoted by the total injection $_$ (Figure 3.7).

$$
\begin{array}{l}
\mathbb{N} : \mathit{TCONS}
\end{array}
\qquad
\begin{array}{l}
_ : \mathbb{N} \rightarrowtail \mathit{CONS} \\
\hline
\mathsf{ftype}\,\mathbb{N} = (\varnothing, \lambda\,\rho : \mathsf{ran}(_) \bullet \langle\rangle)
\end{array}
$$

Figure 3.7: Free Type of Natural Numbers

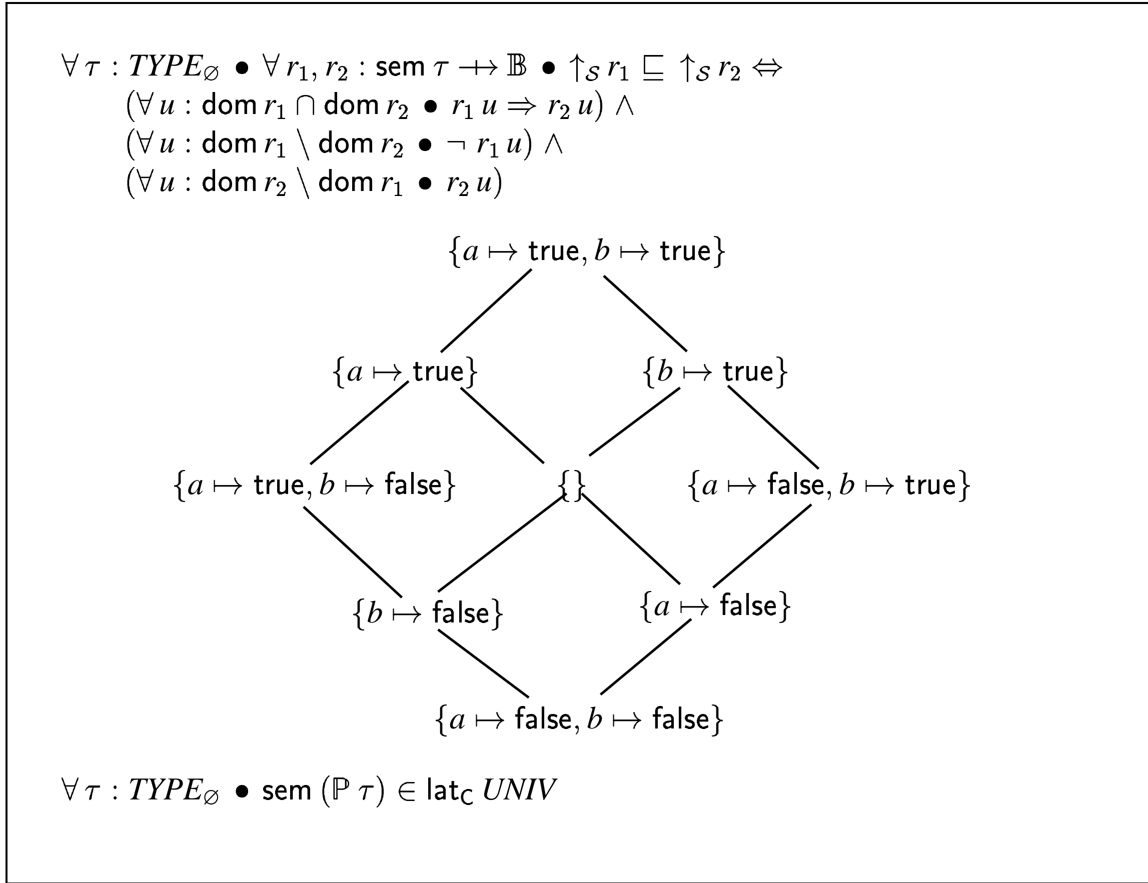


Figure 3.8: The Order on Partial Sets

This forces the interpretation of $\mathbb{N}$ in the semantic universe to be actually isomorphic to natural numbers. Requiring the existence of this type ensures that our semantic model is not trivial.

Bindings and Products. A family of type constructors and value constructors for bindings is available. Bindings are records with named components, called fields. A binding type constructor is denoted as $\langle f_1 : \tau_1; \ \ldots\ ; f_n : \tau_n \rangle$. This type is generated by exactly one constructor, $\langle\!| f_1 == _; \ \ldots\ ; f_n == _ |\!\rangle$. The formal definition is omitted here; it matches the notion of bindings in Z.

Cartesian products and their elements, tuples, are a special case of bindings, with "anonymous" field names, which will be denoted as **1**, **2**, and so on. A cartesian product type, $\tau_1 \times \ldots \times \tau_n$, abbreviates the binding type $\langle \mathbf{1} : \tau_1; \ \ldots\ ; \mathbf{n} : \tau_n \rangle$, and the tuple constructor $(_, \ldots, _)$ the binding constructor $\langle\!| \mathbf{1} == _; \ \ldots\ ; \mathbf{n} == _ |\!\rangle$.

3.2.6 Order and Fixed-Points

The fixed-point operator of μZ, $\mathbf{fix}\, p \lhd e$, selects the smallest element from the set $\{p \mid p = e\}$. To this end an order on values needs to be defined, and lattice properties need to be established (see Section 2.3.4 (on page 33)).

Our model of sets suggests to define the order such that $u \sqsubseteq v \Leftrightarrow u = u \cap v$, where $\cap$

is the definition of intersection on characteristic functions:

$$f_1 \cap f_2 \equiv \lambda\, x : (\mathsf{dom} f_1 \cap \mathsf{dom} f_2) \cup (f_1^{\sim})(\!|\{\mathsf{false}\}|\!) \cup (f_2^{\sim})(\!|\{\mathsf{false}\}|\!) \;\bullet\; [|\, f_1\, x \wedge f_2\, x]$$

The order confirming to this notion of intersection is defined in Figure 3.8 (on the facing page). The elements of a set-type can be refined from non-membership to unknown membership to membership. This follows from that non-membership intersected with unknown membership yields non-membership. For sets over the domain $\{a, b\}$ the order is illustrated in Figure 3.8 (on the preceding page). It is clear that with this order power-set domains are complete lattices: the lattices "join" equals to set intersection and the "meet" to union. The bottom element is the total function which delivers false for all members of the element type's domain, $\mathsf{sem}\ \tau$, and the top element the total function which delivers true. Note that the bottom element does *not* coincide with the completely undefined characteristic function, $\mathsf{undef} = \{\}$, as shown in Figure 3.8 (on the facing page).

We will not attempt to define ordering for the given values of the universe, though this might be in principle possible for constructed values, allowing to build "infinite terms" by fixed-points. For the fixed-point operator, $\mathbf{fix}\, p \triangleleft e$, it is demanded by the type conditions that the free variables in p over which a (simultaneous) fixed-point is constructed represent set-values, and that p is an exhaustive pattern (like a tuple) which matches each value of its type domain. The point-wise lifting of set-order to values matching such patterns is obvious and a formalization is left open.

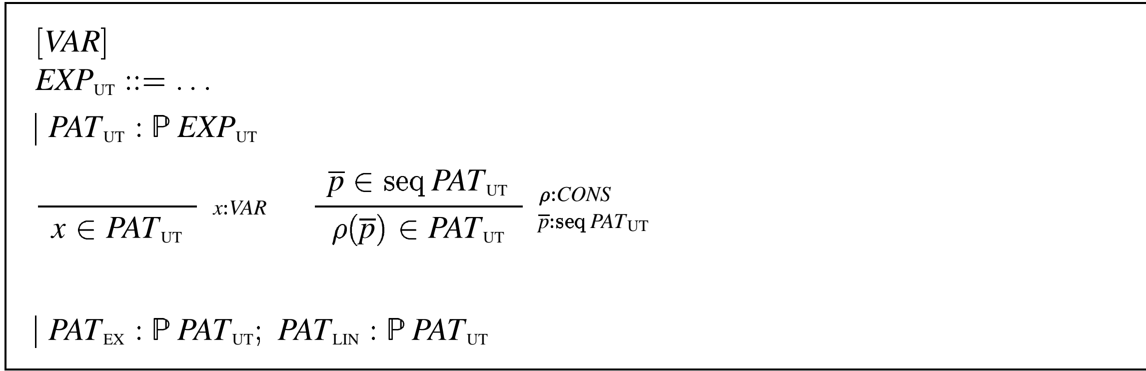


Figure 3.9: Expressions and Patterns

3.3 Expression Syntax and Meaning

This section provides the definitive reference for the exact typing conditions and meaning of μZ expressions. For an informal introduction and motivation, we refer to Section 3.1 (on page 38).

The forms of μZ expressions will be incrementally defined, each one together with its typing rule and meaning function. There are also a few syntactic abbreviations for common forms of expressions, which are introduced in Section 3.3.5 (on page 56).

3.3.1 Variable, Expression and Pattern Type

The syntax of (untyped) expressions is defined by the free type EXP_{UT} (Figure 3.9) which will be extended as we proceed. The set of variables, *VAR*, is pre-given. Patterns are the smallest subset of expressions which satisfies the rules in Figure 3.9. A pattern is called *exhaustive*, if it matches all values of its type. The property of being exhaustive can be syntactically checked by analyzing the free types the constructors of the pattern belong to. A pattern is called *linear*, if no variable in it appears twice. The set PAT_{EX} contains all exhaustive patterns, the set PAT_{LIN} all linear patterns; a formalization is left open.

3.3.2 Typing Relation

The well-typed expressions, $EXP \subseteq EXP_{UT}$, are those which are in the typing relation, $\Gamma \vdash e \cdot \tau$ (Figure 3.10 (on the next page)). The context for typing, $\Gamma \in TYPASS$, is a mapping from variables to types. Rules for the typing are supplied with each individual expression form in Section 3.3.4 (on page 52) below. The well-typed patterns are patterns which are well-typed expressions.

For well-typed expression it is assumed that there is a type-reconstruction function, $\mathsf{type}\ e$, which delivers the type of an expression in its context, as fixed by $_ \vdash _ \cdot _$. The function type is not strictly consistent with our formal model, since an expression does not contain information about its usage context. However, it is conceptually possible to model such a function formally by adding type annotations to expressions.

$$| \; TYPASS == VAR \rightarrow\!\!\!\!\!+ \; TYPE$$
$$| \; _ \vdash _ \cdot _ : \mathbb{P}(TYPASS \times EXP_{\mathrm{UT}} \times TYPE)$$
$$| \; EXP == \{e : EXP_{\mathrm{UT}} \mid \exists \Gamma : TYPASS;\ \tau : TYPE \bullet \Gamma \vdash e \cdot \tau\}$$
$$| \; PAT == PAT_{\mathrm{UT}} \cap EXP$$
$$| \; \mathsf{type} : EXP \longrightarrow TYPE$$

Figure 3.10: Typing Relation: Declarations

$$| \; SEMCTX == TSUBS_{\varnothing} \times UNIASS$$
$$| \; \mathsf{sem} : SEMCTX \longrightarrow EXP \rightarrow\!\!\!\!\!+ \; UNIV$$
$$\begin{aligned} \mathsf{dom}(\mathsf{sem}\ \Omega) = \{e : EXP \mid \exists \Gamma : TYPASS;\ \tau : TYPE \bullet \\ \Gamma \vdash e \cdot \tau \wedge \bigcup(\mathsf{vars}\ (\!| \mathsf{ran}\ \Gamma |\!)) \cup \mathsf{vars}\ \tau \subseteq \mathsf{dom}\ \Omega.1 \wedge \\ \mathsf{dom}\ \Gamma = \mathsf{dom}\ \Omega.2 \wedge \\ (\forall x : \mathsf{dom}\ \Gamma \bullet \Omega.2\, x \in \mathsf{sem}\ (\mathsf{subs}\ \Omega.1(\Gamma\ x)))\} \end{aligned}$$
$$\textbf{where}\ \Omega : SEMCTX$$

Figure 3.11: Meaning Function: Declarations

3.3.3 Meaning Function

With each expression form, the meaning is supplied in subsequent sections, defining a case for the recursive partial function, $\mathsf{sem}\ \Omega$, that is declared in Figure 3.11. The context, $\Omega \in SEMCTX$, consists of a ground type substitution and a value assignment for variables.

The domain of $\mathsf{sem}\ \Omega$ is fixed to those expressions which fit to the context. An expression fits if its typing, $\Gamma \vdash e \cdot \tau$, confirms to Ω, which means the following: all types in Γ, as well as the expression's result type τ, are ground under the substitution $\Omega.1$[4], and all values in the value assignment are in their type's domain. In a similar way we could state that every value delivered by the meaning function is in the according result type's domain, which is, however, omitted.

For the definition of the meaning function, we need the notions of *value matching and substitution*. A pattern p can be matched against a value u under a value assignment $\iota \in UNIASS = VAR \rightarrow\!\!\!\!\!+ \; UNIV$, written as $p \rhd^{\iota} u$. The inverse of value matching is substitution, written as $\mathsf{subs}\ \iota\ p$ (Figure 3.12). These fairly standard concepts are not formalized.

[4] The Z phrase $e.n$ selects the n-th element of a tuple.

$$| \; UNIASS == VAR \rightarrow\!\!\!\!\!+ \; UNIV$$

$$| \; _ \rhd^{-} _ : \mathbb{P}(PAT \times UNIASS \times UNIV)$$
$$| \; \mathsf{subs} : UNIASS \longrightarrow PAT \rightarrow\!\!\!\!\!+ \; UNIV$$

Figure 3.12: Value Matching

$$EXP_{\text{UT}} ::= \ldots \mid (_(_))\langle\langle CONS \times \text{seq}\, EXP_{\text{UT}} \rangle\rangle$$

$$\frac{\begin{array}{c}\mathsf{tinst}\ \rho = \overline{\tau} \mapsto \tau;\ \mathsf{dom}\,\overline{\tau} = \mathsf{dom}\,\overline{\tau}' = \mathsf{dom}\,\overline{e} \\ \forall\, i : \mathsf{dom}\,\overline{\tau} \bullet \Gamma \vdash \overline{e}\, i \cdot \overline{\tau}'\, i \wedge \overline{\tau}'\, i =^{\sigma} \overline{\tau}\, i\end{array}}{\Gamma \vdash \rho(\overline{e}) \cdot \mathsf{subs}\, \sigma\, \tau} \quad \begin{array}{l}\Gamma : TYPASS;\ \rho : CONS;\ \overline{e} : \text{seq}\, EXP_{\text{UT}} \\ \tau : TYPE;\ \overline{\tau}, \overline{\tau}' : \text{seq}\, TYPE;\ \sigma : TSUBS\end{array}$$

$$\begin{array}{l}\mathsf{sem}\,\Omega\,(\rho(\overline{e})) \Leftarrow \mathsf{sem}\,\Omega.1\,\rho\,((\mathsf{sem}\,\Omega) \circ \overline{e}) \\ \quad \textbf{where}\ \Omega : SEMCTX;\ \rho : CONS;\ \overline{e} : \text{seq}\, EXP\end{array}$$

Figure 3.13: Syntax and Meaning: Constructor Application

$$EXP_{\text{UT}} ::= \ldots \mid (\{_\})\langle\langle EXP_{\text{UT}} \rangle\rangle \mid \mu\, \langle\langle EXP_{\text{UT}} \rangle\rangle$$

$$\frac{\Gamma \vdash e \cdot \tau}{\Gamma \vdash \{e\} \cdot \mathbb{P}\, \tau} \quad \begin{array}{l}\Gamma : TYPASS \\ e : EXP_{\text{UT}};\ \tau : TYPE\end{array} \qquad \frac{\Gamma \vdash e \cdot \mathbb{P}\, \tau}{\Gamma \vdash \mu\, e \cdot \tau} \quad \begin{array}{l}\Gamma : TYPASS \\ e : EXP_{\text{UT}};\ \tau : TYPE\end{array}$$

$$\begin{array}{l}\mathsf{sem}\,\Omega\,(\{e\}) \Leftarrow \uparrow_{\mathcal{S}} (\lambda\, u : \mathsf{sem}\,(\mathsf{subs}\,\Omega.1\,(\mathsf{type}\, e)) \bullet [\mid u = \mathsf{sem}\,\Omega\ e]) \\ \mid \quad (\mu\, e) \Leftarrow \textbf{let}\, f == \downarrow_{\mathcal{S}} (\mathsf{sem}\,\Omega\ e) \bullet \\ \qquad\qquad \textbf{if}\ \mathsf{dom} f = \mathsf{sem}\,(\mathsf{subs}\,\Omega.1\,((\mathbb{P}\,^{\sim})(\mathsf{type}\, e))) \wedge (f^{\sim}) \in \mathbb{B} \twoheadrightarrow UNIV \\ \qquad\qquad \textbf{then}\ (f^{\sim})\ \mathsf{true}\ \textbf{else}\ \mathsf{undef} \\ \quad \textbf{where}\ \Omega : SEMCTX;\ e : EXP\end{array}$$

Figure 3.14: Syntax and Meaning: Singleton Set Display and Selection

3.3.4 Basic Forms: Typing and Meaning

The tools for defining typing conditions and semantic meaning have been introduced in the previous sections. In the sequel, for each basic expression form its typing and meaning will be defined.

Constructor Application

A constructor application is denoted as $\rho(e_1, \ldots, e_n)$, with $n \geq 0$. The definitions are given in Figure 3.13.

In the typing rule, we use the function $\mathsf{tinst}\ \rho$, which delivers the type of a constructor with "fresh" type variables. The meaning is derived from the constructor's semantic interpretation, $\mathsf{sem}\ \sigma\ \rho$, for given ground type substitution σ found in the context.

Singleton Set Display and Selection

A set which contains exactly one element is denoted by the expression $\{e\}$, called a *singleton set display*. The element of a singleton set is selected with the expression $\mu\, e$, called a *singleton set selection*. The definitions are given in Figure 3.14.

The type of the display is constructed from the type of the element. The operand of the selection must denote a set. The type of the result is extracted from the type of this set.

The singleton set display denotes a characteristic function which is exactly true for the display contents, and false for all other elements of the type. The meaning of singleton

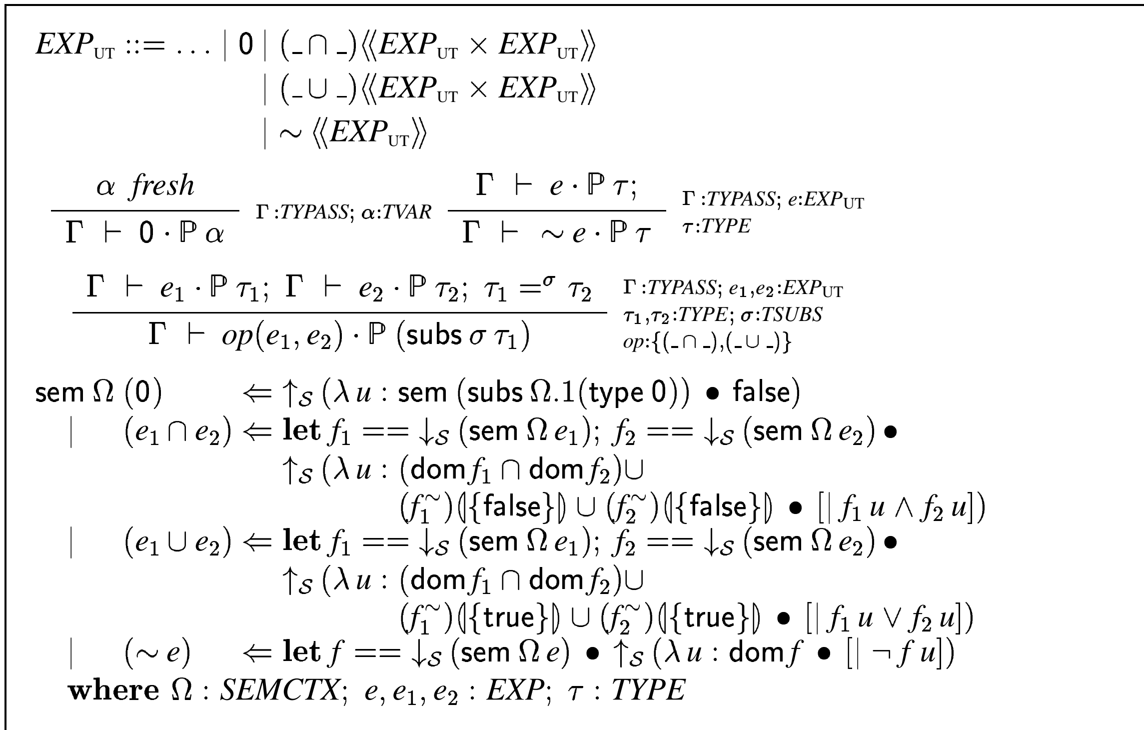


Figure 3.15: Syntax and Meaning: Set Algebra

set selection is defined only if the operand denotes a characteristic function over the entire types domain, which is true for exactly one element. Note that for a partial characteristic function, the μ form is undefined.

Set Algebra

The empty set is denoted by the expression $\mathsf{0}$, the set intersection by the expression $e_1 \cap e_2$, set union by $e_1 \cup e_2$, and set complement by $\sim e$. The definitions are given in Figure 3.15.

The type of the empty set is a powerset over some fresh type variable α. For set intersection and union, both operands must denote sets of types which can be unified. For the set complement, the operand must be a set.

The meaning of the empty set is a total characteristic function which constantly yields false. The meaning of set intersection, set union, and set complement has been already discussed in Section 3.2.1 (on page 43). Here, we take care for the encoding details.

Set Translation

The translation of a set, $e[p_1 \mapsto p_2]$, describes the set of values from e which are transformed by the pattern p_1 to p_2. The definitions are given in Figure 3.16 (on the next page).

The patterns must be typeable under a shared environment, and must be linear. The type of e must be unifiable with the type of p_1; the resulting type is the substitution of the unifier in the type of p_2.

$$EXP_{\text{UT}} ::= \ldots \mid (_[_ \mapsto _]) \langle\!\langle EXP_{\text{UT}} \times PAT_{\text{UT}} \times PAT_{\text{UT}} \rangle\!\rangle$$

$$\frac{\Gamma' \vdash p_1 \cdot \tau_1;\ \Gamma' \vdash p_2 \cdot \tau_2;\ p_1 \in PAT_{\text{LIN}};\ p_2 \in PAT_{\text{LIN}} \quad \Gamma \vdash e \cdot \mathbb{P}\,\tau;\ \tau =^{\sigma} \tau_1}{\Gamma \vdash e[p_1 \mapsto p_2] \cdot \mathbb{P}\,(\mathsf{subs}\,\sigma\tau_2)} \quad \begin{array}{l}\Gamma,\Gamma':TYPASS;\ e:EXP_{\text{UT}}\\ p_1,p_2:PAT_{\text{UT}}\\ \tau,\tau_1,\tau_2:TYPE\\ \sigma:TSUBS\end{array}$$

$$_ \overset{\mapsto}{\rightsquigarrow} _ == \{u_1 : UNIV;\ p_1,p_2 : PAT;\ u_2 : UNIV \mid \exists\,\sigma : TSUBS_{\varnothing};\ \iota : UNIASS \bullet u_1 \in \mathsf{sem}\,(\mathsf{subs}\,\sigma\,(\mathsf{type}\,p_1)) \wedge u_2 \in \mathsf{sem}\,(\mathsf{subs}\,\sigma\,(\mathsf{type}\,p_2)) \wedge p_1 \rhd^{\iota} u_1 \wedge p_2 \rhd^{\iota} u_2\}$$

$$\mathsf{tr} == \lambda\,p_1,p_2 : PAT \bullet \lambda\,u : UNIV \bullet \{u' : UNIV \mid u \overset{p_1 \mapsto p_2}{\rightsquigarrow} u'\}$$

$$\begin{array}{l}\mathsf{sem}\,\Omega\,(e[p_1 \mapsto p_2]) \Leftarrow\\ \quad \mathbf{let}\,f == \downarrow_{\mathcal{S}}(\mathsf{sem}\,\Omega\,e);\\ \qquad D_1 == \mathsf{sem}\,(\mathsf{subs}\,\Omega.1(\mathsf{type}\,p_1));\\ \qquad D_2 == \mathsf{sem}\,(\mathsf{subs}\,\Omega.1(\mathsf{type}\,p_2)) \bullet\\ \quad \uparrow_{\mathcal{S}}((\lambda\,u : \bigcup(\mathsf{tr}(p_1 \mapsto p_2)(\!|\mathsf{dom} f|\!)) \bullet \bigvee(f(\!|\mathsf{tr}(p_2 \mapsto p_1)u|\!))) \cup\\ \qquad (\lambda\,u : D_2 \setminus \bigcup(\mathsf{tr}(p_1 \mapsto p_2)(\!|D_1|\!)) \bullet \mathsf{false}))\\ \quad \mathbf{where}\ \Omega : SEMCTX;\ e : EXP;\ p_1,p_2 : PAT\end{array}$$

Figure 3.16: Syntax and Meaning: Set Translation

For the specification of the meaning of translation, we define an auxiliary relation, $u_1 \overset{p_1 \mapsto p_2}{\rightsquigarrow} u_2$, which relates values which are translated by two patterns. The relation is described polymorphically, by relating all those values u_1, u_2 which are in an instance of the pattern's types under a common ground type substitution σ and which can be matched by the patterns using the same value assignment ι. The auxiliary function `tr` yields the set of values which are reached under the translation by a pattern pair.

For the meaning of translation, two cases have to be distinguished: those elements which are defined in the characteristic function denoted by the set e and which are reached by $p_1 \mapsto p_2$, are in the domain of the resulting characteristic function; their membership is derived by Boolean disjunction of their image under the reverse of pattern translation. Those values which can *not* be reached by the translation are defined to be constantly not members of the set. This definition reflects that a translation may "filter out" undefined elements of the translated set. Consider for example a set over lists where the membership of the empty list, denoted by the constructor *nil*, is unknown, whereas the membership of all elements denoted by the constructor $cons(X, XS)$ is known. Translating such a set by $cons(X, XS) \mapsto cons(X, XS)$ results in a set of which *nil* cannot be member. This is reflected by the second λ in Figure 3.16. The example also shows that a translation by identical patterns does not necessarily yield an identical set value.

Schema, Fixed-Point, and Variable

A schema is denoted by the expression $\{p \mid e\}$, where p is a pattern, and e an expression of type $\mathbb{B}$. A fixed-point is denoted by $\mathbf{fix}\,p \lhd e$. A variable application is written as x. The definitions are given in Figure 3.17 (on the next page).

$$EXP_{\mathrm{UT}} ::= \ldots \mid (\{_ \mid _\})\langle\langle PAT_{\mathrm{UT}} \times EXP_{\mathrm{UT}}\rangle\rangle \mid (\mathbf{fix}\ _ \lhd _)\langle\langle PAT_{\mathrm{UT}} \times EXP_{\mathrm{UT}}\rangle\rangle \mid (_)\langle\langle VAR\rangle\rangle$$

$$\frac{\mathsf{dom}\,\Gamma' = \mathsf{vars}\,p;\ p \in PAT_{\mathrm{LIN}} \quad \Gamma' \vdash p \cdot \tau;\ \Gamma \oplus \Gamma' \vdash e \cdot \mathbb{B}}{\Gamma \vdash \{p \mid e\} \cdot \mathbb{P}\,\tau} \quad \begin{array}{l}\Gamma,\Gamma'{:}TYPASS\\ p{:}PAT_{\mathrm{UT}};\ e{:}EXP_{\mathrm{UT}}\\ \tau{:}TYPE\end{array} \qquad \frac{x \in \mathsf{dom}\,\Gamma}{\Gamma \vdash x \cdot \Gamma\,x} \quad \begin{array}{l}\Gamma{:}TYPASS\\ x{:}VAR\end{array}$$

$$\frac{\mathsf{dom}\,\Gamma' = \mathsf{vars}\,p;\ p \in PAT_{\mathrm{LIN}} \cap PAT_{\mathrm{EX}};\ \mathsf{ran}\,\Gamma' \subseteq \mathsf{ran}\,\mathbb{P} \quad \Gamma' \vdash p \cdot \tau_1;\ \Gamma \oplus \Gamma' \vdash e \cdot \tau_2;\ \tau_1 =^{\sigma} \tau_2}{\Gamma \vdash \mathbf{fix}\,p \lhd e \cdot \mathsf{subs}\,\sigma\,\tau_1} \quad \begin{array}{l}\Gamma,\Gamma'{:}TYPASS;\ \sigma{:}TSUBS\\ p{:}PAT_{\mathrm{UT}}\\ e{:}EXP_{\mathrm{UT}}\\ \tau_1,\tau_2{:}TYPE\end{array}$$

$$\begin{array}{lll}
\mathsf{sem}\,\Omega\,(\{p \mid e\}) & \Leftarrow & \\
\multicolumn{3}{l}{\quad \uparrow_S \{u : \mathsf{sem}\,(\mathsf{subs}\,\Omega.1(\mathsf{type}\,p));\ f : UNIV \mathrel{+\!\!\!\to} \mathbb{B};\ \iota : UNIASS \mid} \\
\multicolumn{3}{l}{\qquad p \rhd^{\iota} u \Rightarrow \uparrow_S f = \mathsf{sem}\ (\Omega.1, \Omega.2 \oplus \iota)\,e \wedge \uparrow_S f \neq \mathsf{undef} \bullet} \\
\multicolumn{3}{l}{\qquad\quad u \mapsto [\![p \rhd^{\iota} u \wedge f = \mathsf{tt}]\!]\}} \\
\mid\ (\mathbf{fix}\,p \lhd e) & \Leftarrow & \mathbf{let}\ \tau == \mathsf{subs}\,\Omega.1(\mathsf{type}\,p) \\
 & & \quad f == \downarrow_S (\mathsf{sem}\,\Omega\{p \mid p = e\}) \bullet \\
 & & \mathbf{let}\ cands == (f^{\sim})(\!|\{\mathsf{true}\}|\!) \bullet \\
 & & \mathbf{if}\ \sqcap cands \in cands\ \mathbf{then}\ \sqcap cands \\
 & & \mathbf{else}\ \mathsf{undef} \\
\mid\ (x) & \Leftarrow & \Omega.2\,x \\
\multicolumn{3}{l}{\mathbf{where}\ \Omega : SEMCTX;\ p : PAT;\ e : EXP;\ x : VAR}
\end{array}$$

Figure 3.17: Syntax and Meaning: Schema, Fixed-Point, and Variable

The pattern of a schema must be typed under a minimal type assignment, and must be linear. The property must be typed as a Boolean under an extension of the environment by the type assignment used for the pattern, where variables of the pattern hide variables in the environment. The resulting type of the schema is a powerset of the pattern's type. The pattern of a fixed-point must be typed under a minimal type assignment, must be linear, and must be exhaustive. Each variable in the pattern must have a set type. The definition must be typed under an extension of the environment by the type assignment for the pattern, where variables from the pattern hide variables in the environment, and must unify with the pattern's type. The resulting type is this unified type.

The meaning is defined as follows. For schemas, a characteristic function is constructed, the domain of which are the elements from the schema's pattern type such that, if they match the pattern, then the property evaluates to a defined Boolean value under the bindings resulted by this match. The schema's characteristic function yields true for an element if the pattern has matched and the properties Boolean value is true.

For the fixed-point, we use the meaning of $\{p \mid p = e\}$ to construct the set of candidates. By the typing conditions we know that the variables in p represent sets, and the domain of p's type is ordered by $_ \sqsubseteq _$. Thus, if the infimum of the candidate set is a candidate, we select it, otherwise undef is delivered. A sufficient condition for existence of the fixed-point is that e is order-preserving, see Section 2.3.4 (on page 33).

$$\mathsf{1} == \sim \mathsf{0} \qquad \bot == \mu\, \mathsf{0}$$

$$?_{\mathsf{1}}_, ?_{\mathsf{0}}_ : EXP_{\mathrm{UT}} \rightarrowtail EXP_{\mathrm{UT}}$$

$$\begin{array}{l} ?_{\mathsf{1}} e = e[x \mapsto \diamond] \\ ?_{\mathsf{0}} e = \sim ?_{\mathsf{1}} e \\ \quad \textbf{where } e : EXP_{\mathrm{UT}};\ x : VAR \end{array}$$

$$_ \in _, _ = _ : EXP_{\mathrm{UT}} \times EXP_{\mathrm{UT}} \rightarrowtail EXP_{\mathrm{UT}}$$

$$\begin{array}{l} e \in e' = ?_{\mathsf{1}}(\{e\} \cap e') \\ e = e' = e \in \{e'\} \\ \quad \textbf{where } e, e' : EXP_{\mathrm{UT}} \end{array}$$

$$\bigcup, \bigcap : \operatorname{seq} EXP_{\mathrm{UT}} \rightarrowtail EXP_{\mathrm{UT}}$$

$$\begin{array}{l} \bigcup = \lambda\, \overline{e} : \operatorname{seq}_1 EXP_{\mathrm{UT}} \bullet \textbf{if } \#\overline{e} = 0 \textbf{ then } \mathsf{0} \textbf{ else} \\ \qquad \textbf{if } \#\overline{e} = 1 \textbf{ then } head\,\overline{e} \textbf{ else } head\,\overline{e} \cup \bigcup(tail\,\overline{e}) \\ \bigcap = \lambda\, \overline{e} : \operatorname{seq}_1 EXP_{\mathrm{UT}} \bullet \textbf{if } \#\overline{e} = 0 \textbf{ then } \mathsf{1} \textbf{ else} \\ \qquad \textbf{if } \#\overline{e} = 1 \textbf{ then } head\,\overline{e} \textbf{ else } head\,\overline{e} \cap \bigcap(tail\,\overline{e}) \end{array}$$

Figure 3.18: Expression Abbreviation Forms

3.3.5 Expression Abbreviation Forms

We have already mentioned several syntactic abbreviations for frequent patterns of μZ expressions. The abbreviations are specified in Figure 3.18 as constants and injective functions which map into expressions. The functions work on untyped syntax; the typing conditions are inherited from the basic forms they map to.

The following explanations about the definitions. The universal set, $\mathsf{1}$, is the complement of the empty set. The undefined value, $\bot$, is the singleton selection on the empty set. The Boolean expression $?_{\mathsf{1}}e$, which tests whether the given set is non-empty, is an abbreviation for a translation to the Boolean unit, $\diamond$. The complementary form $?_{\mathsf{0}}e$ tests whether the given set is empty. The Boolean membership, $e \in e'$, is mapped to an intersection of a singleton set containing the element with the set to test membership in, and a subsequent test for emptiness of the result. The equality, $e = e'$, is derived from membership in a singleton set.

In order to work with sequences of unions and intersections, two further abbreviations are introduced: $\bigcup \overline{e}$ describes the *generalized union*, and $\bigcap \overline{e}$ the *generalized intersection* of the sequence of expressions $\overline{e}$. For empty expression sequences, the according neutral element ($\mathsf{0}$ resp. $\mathsf{1}$) is denoted.

$$\begin{array}{l}
\mathsf{vars} : EXP \longrightarrow \mathbb{P}\ VAR \\
EXPASS == VAR \twoheadrightarrow EXP \\
_ =^{-} _ : \mathbb{P}(PAT \times EXPASS \times PAT) \\
\mathsf{subs} : EXPASS \longrightarrow EXP \twoheadrightarrow EXP \\
\mathsf{mksubs} : PAT \times EXP \longrightarrow EXPASS \\
\hline
\mathsf{mksubs} = \lambda\, p : PAT;\ e : EXP \bullet \lambda\, x : \mathsf{vars}\, p \bullet \mu\, (\{e\}[p \mapsto x]) \\
PATCTX == PAT \rightarrowtail PAT \\
EXPCTX == EXP \rightarrowtail EXP \\
\mathsf{varorder} : \mathbb{F}\ VAR \longrightarrow \mathrm{seq}\ VAR
\end{array}$$

Figure 3.19: Syntactic Tools

3.4 Equational Theory

An equational theory for μZ will be developed in this section. The laws will be shown to be consistent with the model of μZ developed in the previous section.

3.4.1 Syntactic Tools

For defining the equivalences, some syntactic tools for working with expressions are required (Figure 3.19). The variables which occur free in an expression (are not bound by some schema's pattern) are delivered with $\mathsf{vars}\ e$. Two patterns are unified using the notation $p_1 =^{\gamma} p_2$ determining the smallest assignment $\gamma \in EXPASS = VAR \twoheadrightarrow EXP$ which makes them equal. An expression assignment can be substituted in an expression by $\mathsf{subs}\ \gamma\ e$. Substitution is partial because of typing conditions for the substituted expressions. These fairly standard concepts are not formalized.

Given a pattern p and an expression e, we build a substitution of the variables in the pattern by sub-components of the expression with $\mathsf{mksubs}\,(p, e)$. Variables x are substituted by a "selection" from the expression e expressed via a translation. Note that the meaning of this selection may be undefined (undef), in case the pattern p does not match the value of e.

A pattern context, $\mathrm{P} \in PATCTX$, is a "pattern with a hole". It is described by an injective function from a pattern to a pattern. Similar, an expression context $\mathrm{E} \in EXPCTX$ is defined. The context functions are partial because of typing conditions.

For variables a total order is assumed, such that we can convert a finite set of variables into a sequence of variables by the function $\mathsf{varorder}$.

3.4.2 Equivalence Relation

The equivalences are described by a relation over well-typed expressions of the same type. This relation has the equivalence properties and is a congruence (is closed under contexts). The declarations are given in Figure 3.20. The equivalence relation will be incrementally refined below: $_ \equiv _$ is supposed to be the smallest relation satisfying the laws in Figure 3.20 and the laws added later on.

$$
\begin{array}{ll}
\multicolumn{2}{l}{| _ \equiv _ : \mathbb{P}\{e, e' : EXP \mid \mathsf{type}\, e = \mathsf{type}\, e'\}} \\
e \equiv e & \\
e \equiv e' & \Rightarrow e' \equiv e \\
e \equiv e' \wedge e' \equiv e'' & \Rightarrow e \equiv e'' \\
e \equiv e' \wedge \{e, e'\} \subseteq \mathsf{dom}\, \mathrm{E} & \Rightarrow \mathrm{E}\ e \equiv \mathrm{E}\ e' \\
\quad \textbf{where } e, e', e'' : EXP;\ \mathrm{E} : EXPCTX &
\end{array}
$$

Figure 3.20: Equivalence Relation: Declaration

$$
\begin{array}{lcl}
e \cap e' & \equiv & e' \cap e \\
e \cap (e' \cap e'') & \equiv & (e \cap e') \cap e'' \\
e \cap \mathsf{0} & \equiv & \mathsf{0} \\
e \cap e & \equiv & e \\
e \cap (e' \cup e'') & \equiv & (e \cap e') \cup (e \cap e'') \\
\multicolumn{3}{l}{\quad \textbf{where } e, e', e'' : EXP}
\end{array}
$$

$$
\begin{array}{lcl}
e \cup e' & \equiv & e' \cup e \\
e \cup (e' \cup e'') & \equiv & (e \cup e') \cup e'' \\
e \cup \mathsf{1} & \equiv & \mathsf{1} \\
e \cup e & \equiv & e \\
e \cup (e' \cap e'') & \equiv & (e \cup e') \cap (e \cup e'') \\
\multicolumn{3}{l}{\quad \textbf{where } e, e', e'' : EXP}
\end{array}
$$

$$
\begin{array}{lcl}
\sim (e \cap e') & \equiv & \sim e \cup \sim e' \\
\sim (e \cup e') & \equiv & \sim e \cap \sim e' \\
\sim \mathsf{0} & \equiv & \mathsf{1} \\
\sim (\sim e) & \equiv & e \\
\multicolumn{3}{l}{\quad \textbf{where } e, e', e'' : EXP}
\end{array}
$$

Figure 3.21: Boolean Laws

3.4.3 Boolean Laws

The set operations follow almost all of the usual laws for *Boolean algebras*. Commutativity and distributivity of union and intersection hold, and De Morgan's laws are satisfied, such that we can build a conjunctive or disjunctive normal form of expressions (Figure 3.21).

However, the "principle of the excluded middle" does not hold: $e \cap \sim e \equiv \mathsf{0}$ is *not* valid for arbitrary e, as well as $e \cup \sim e \equiv \mathsf{1}$. The reason for this is the partiality of sets: if e contains unknowns, then in the complement of e these unknowns keep unknown, and therefore do not absorb each other. As a consequence, the implication $e \Rightarrow e$, interpreted as the usual abbreviation for $(e \cap e) \cup \sim e$, does not equal to $\mathsf{1}$.

Proof. We prove one of the distributivity laws and one of the De Morgan laws.

- Let f_1, f_2, f_3 be characteristic functions. We want to show (ignoring encoding details) that the pair $u \mapsto b$ is in the characteristic function $h_1 = f_1 \cap (f_2 \cup f_3)$ iff it is in the function $h_2 = (f_1 \cap f_2) \cup (f_1 \cap f_3)$:

  Case $u \mapsto \mathsf{true}$: According to definition of union and intersection, $u \mapsto \mathsf{true} \in h_1$ iff $u \mapsto \mathsf{true} \in f_1 \wedge (u \mapsto \mathsf{true} \in f_2 \vee u \mapsto \mathsf{true} \in f_3)$. This is equivalent to $(u \mapsto \mathsf{true} \in f_1 \wedge u \mapsto \mathsf{true} \in f_2) \vee (u \mapsto \mathsf{true} \in f_1 \wedge u \mapsto \mathsf{true} \in f_3)$, which in turn holds iff $u \mapsto \mathsf{true} \in h_2$.

$$\{e\} \cap \sim \{e\} \equiv 0$$
$$\{e\} \cup \sim \{e\} \equiv 1$$
where $e : EXP$

Figure 3.22: Excluded Middle Laws

$$\mu \{e\} \equiv e$$
where $e : EXP$

Figure 3.23: Selection Elimination Law

Case $u \mapsto \mathsf{false}$**:** Then we have $u \mapsto \mathsf{false} \in h_1$ iff $u \mapsto \mathsf{false} \in f_1 \vee (u \mapsto \mathsf{false} \in f_2 \wedge u \mapsto \mathsf{false} \in f_3)$. This is equivalent to $(u \mapsto \mathsf{false} \in f_1 \vee u \mapsto \mathsf{false} \in f_2) \wedge (u \mapsto \mathsf{false} \in f_1 \vee u \mapsto \mathsf{false} \in f_3)$, which holds iff $u \mapsto \mathsf{false} \in h_2$.

- Let f_1 and f_2 be characteristic functions. We show that the pair $u \mapsto b$ is in the characteristic function $h_1 = \sim (f_1 \cap f_2)$ iff it is in the function $h_2 = \sim f_1 \cup \sim f_2$. We thereby use the following fact: let $g' = \sim g$, then it follows from the definition of complement: $u \mapsto b \in g'$ iff $u \mapsto \neg\, b \in g$, where $\neg\, b$ is (Z-level) negation of the Boolean value b. (Note the difference to the (wrong) assertion, $u \mapsto b \in g' \Leftrightarrow \neg\, (u \mapsto b \in g)$.)

Case $u \mapsto \mathsf{true}$**:** We have $u \mapsto \mathsf{true} \in h_1$ iff $u \mapsto \mathsf{false} \in (f_1 \cap f_2)$, iff $u \mapsto \mathsf{false} \in f_1 \vee u \mapsto \mathsf{false} \in f_2$, iff $u \mapsto \mathsf{true} \in \sim f_1 \vee u \mapsto \mathsf{true} \in \sim f_2$, iff $u \mapsto \mathsf{true} \in h_2$.

Case $u \mapsto \mathsf{false}$**:** We have $u \mapsto \mathsf{false} \in h_1$ iff $u \mapsto \mathsf{true} \in (f_1 \cap f_2)$, iff $u \mapsto \mathsf{true} \in f_1 \wedge u \mapsto \mathsf{true} \in f_2$, iff $u \mapsto \mathsf{false} \in \sim f_1 \wedge u \mapsto \mathsf{false} \in \sim f_2$, iff $u \mapsto \mathsf{false} \in h_2$.

□

3.4.4 Singleton Set Laws

Excluded Middle

For singleton sets the "excluded middle" holds (Figure 3.22). One can easily see the validity by considering the characteristic function constructed for a singleton set: it is total and true exactly for the element e. The complement yields a total characteristic function which is true for all elements except e.

Selection Elimination

Singleton set selection on a singleton set construction reduces to the singleton sets member (Figure 3.23). The validity can be easily seen from the representation of singleton sets and the meaning of the μ -operator.

$$
\begin{array}{lll}
\{\rho(\overline{e})\} & \equiv & \bigcap((\lambda\, i : \mathbb{N} \bullet \{\overline{e}\, i\}[(\overline{x}\, i) \mapsto \rho((_) \circ \overline{x})]) \circ (\mathsf{id}(1 \mathinner{..} \#\overline{e}))) \\
\{\rho\} & \equiv & \{\diamond\}[\diamond \mapsto \rho] \\
\multicolumn{3}{l}{\quad \mathbf{where}\ \rho : \mathit{CONS};\ \overline{e} : \mathrm{seq}\, \mathit{EXP};\ \overline{x} : \mathrm{seq}\, \mathit{VAR} \mid \#\overline{e} = \#\overline{x} > 0}
\end{array}
$$

Figure 3.24: Constructor Decomposition Laws

$$
\begin{array}{lll}
e[p_1 \mapsto p_2][p_3 \mapsto p_4] & \equiv & e[\mathsf{subs}\ \gamma\, p_1 \mapsto \mathsf{subs}\ \gamma\, p_4] \quad ,\mathbf{if}\, p_2 =^{\gamma} p_3 \\
e[p_1 \mapsto p_2][p_3 \mapsto p_4] & \equiv & \mathsf{0} \quad ,\mathbf{if}\, \neg\, p_2 =^{\gamma} p_3 \\
e[p' \mapsto p'] & \equiv & e \\
\multicolumn{3}{l}{\quad \mathbf{where}\ e : \mathit{EXP};\ p_1, p_2, p_3, p_4 : \mathit{PAT};\ p' : \mathit{PAT}_{\mathrm{EX}};\ \gamma\ : \mathit{EXPASS}}
\end{array}
$$

Figure 3.25: Translation Composition and Identity Laws

Constructor Decomposition

A singleton set containing a constructor application with arity greater zero can be decomposed into an intersection of projections. A singleton set containing a zero-arity constructor equals to a translation of a truth value (Figure 3.24). The first law is formulated in "..." notation as follows:

$$\{\rho(e_1, \ldots, e_n)\} \equiv \{e_1\}[x_1 \mapsto \rho(x_1, \ldots, _)] \cap \ldots \cap \{e_n\}[x_n \mapsto \rho(_, \ldots, x_n)]$$

It holds since the translation $\{e_i\}[x_i \mapsto \rho(\ldots, x_i, \ldots)]$ yields the set of all values $\rho(\ldots, e_i, \ldots)$, where the ith argument is fixed by e_i and all other are arbitrary. Intersecting the sets resulting from such a translation for each argument of the constructor must thus yield in $\{\rho(e_1, \ldots, e_n)\}$. Thereby, definedness of e_i is not significant, since constructors and singleton sets are not strict.

3.4.5 Translation Laws

Composition and Identity

Consecutive translations are composable, if the intermediate patterns can be unified. If they cannot be unified, consecutive translation reduces to the empty set. A translation by an exhaustive pattern has no effect (Figure 3.25).

Proof. We prove composition of consecutive translation. We want to show that the pair $u \mapsto b$ is in the characteristic function $h_1 = f[p_1 \mapsto p_2][p_3 \mapsto p_4]$ iff there exists an assignment γ such that $p_2 =^{\gamma} p_3$ and $u \mapsto b$ is in the characteristic function $h_2 = f[\mathsf{subs}\ \gamma\, p_1 \mapsto \mathsf{subs}\ \gamma\, p_4]$.

Case $u \mapsto \mathsf{true}$: When $u \mapsto \mathsf{true} \in h_1$, according to definition of translation there exist value assignments ι, ι', and values u', u'', such that $p_4 \rhd^{\iota} u$, $p_3 \rhd^{\iota} u'$, $p_2 \rhd^{\iota'} u'$, and $p_1 \rhd^{\iota'} u''$. Since p_3 and p_2 match the same value u', there must exists a smallest pattern assignment γ such that $p_2 =^{\gamma} p_3$, and $(\mathsf{subs}\ \gamma\ p_2) \rhd^{\iota''} u'$ as well as $(\mathsf{subs}\ \gamma\ p_3) \rhd^{\iota'''} u'$. The assignment γ specializes the patterns compatible to the value u', and hence we also have $(\mathsf{subs}\ \gamma\ p_1) \rhd^{\iota''} u''$ and $(\mathsf{subs}\ \gamma\ p_4) \rhd^{\iota'''} u$, and therefore $u \mapsto \mathsf{true} \in h_2$.

$$
\begin{array}{lll}
(e \cup e')[p_1 \mapsto p_2] & \equiv & e[p_1 \mapsto p_2] \cup e'[p_1 \mapsto p_2] \\
(e \cap e')[p_1 \mapsto p_2] & \equiv & e[p_1 \mapsto p_2] \cap e'[p_1 \mapsto p_2] \quad \text{, if } \mathsf{vars}\, p_1 \subseteq \mathsf{vars}\, p_2 \\
(\sim e)[p_1 \mapsto p_2] & \equiv & \sim (e[p_1 \mapsto p_2]) \quad \text{, if } \mathsf{vars}\, p_1 \subseteq \mathsf{vars}\, p_2 \wedge \\
 & & \quad p_1 \in PAT_{\text{EX}} \wedge p_2 \in PAT_{\text{EX}} \\
\textbf{where } e, e' : EXP;\ p_1, p_2 : PAT & &
\end{array}
$$

Figure 3.26: Translation Distributivity Laws

Case $u \mapsto \mathsf{false}$**:** Similar argumentation as above, but with negated propositions.

□

Distributivity over Set-Operators

Any translation distributes over set union. A translation by an embedding distributes over set intersection. An embedding distributes over set complement if it is exhaustive (Figure 3.26).

An analogy to predicate logic gives an intuition why hiding translations (where $\mathsf{vars}\, p_1 \supset \mathsf{vars}\, p_2$) do not distribute over intersection. Hiding in μZ can be interpreted as an existential quantifier. In predicate calculus, the following holds: $\exists x \bullet P \vee Q$ iff $(\exists x \bullet P) \vee (\exists x \bullet Q)$. However, it does not hold $\exists x \bullet P \wedge Q$ iff $(\exists x \bullet P) \wedge (\exists x \bullet Q)$, since in the later case P and Q must be satisfied simultaneously for the same x.

*Proof***.** We prove distributivity of any translation over union and of embedding translations over intersection, using the auxiliary relation $v \overset{p_1 \mapsto p_2}{\leadsto} u$ (Section 3.3.4 (on page 54)).

- We prove that $u \mapsto b$ is in $h_1 = (f \cup f')[p_1 \mapsto p_2]$ iff it is in $h_2 = f[p_1 \mapsto p_2] \cup f/p_1 p_2$.

  Case $u \mapsto \mathsf{true}$**:** We have $u \mapsto \mathsf{true} \in h_1$ iff $\exists\, v : UNIV \bullet v \overset{p_1 \mapsto p_2}{\leadsto} u \bullet v \mapsto \mathsf{true} \in (f \cup f')$ iff $\exists\, v : UNIV \mid v \overset{p_1 \mapsto p_2}{\leadsto} u \bullet v \mapsto \mathsf{true} \in f \vee v \mapsto \mathsf{true} \in f'$ iff $(\exists\, v : UNIV \bullet v \overset{p_1 \mapsto p_2}{\leadsto} u \bullet v \mapsto \mathsf{true} \in f) \vee (\exists\, v : UNIV \bullet v \overset{p_1 \mapsto p_2}{\leadsto} u \bullet v \mapsto \mathsf{true} \in f')$ iff $u \mapsto \mathsf{true} \in h_2$.

  Case $u \mapsto \mathsf{false}$**:** similar.

- We prove that $u \mapsto b$ is in $h_1 = (f \cap f')[p_1 \mapsto p_2]$ iff it is in $h_2 = f[p_1 \mapsto p_2] \cap f[p_1 \mapsto p_2]$, where $\mathsf{vars}\, p_1 \subseteq \mathsf{vars}\, p_2$. Thereby the following property is used: if $\mathsf{vars}\, p_1 \subseteq \mathsf{vars}\, p_2$, then for a given u there exists not more then one v in the same type such that $v \overset{p_1 \mapsto p_2}{\leadsto} u$. This follows from the fact that all variables in p_1 are bound by a match of p_2 against u.

  Case $u \mapsto \mathsf{true}$**:** We have $u \mapsto \mathsf{true} \in h_1$ iff $\exists_1\, v : UNIV \mid v \overset{p_1 \mapsto p_2}{\leadsto} u \bullet v \mapsto \mathsf{true} \in f \wedge v \mapsto \mathsf{true} \in f'$ iff $(\exists_1\, v : UNIV \bullet v \overset{p_1 \mapsto p_2}{\leadsto} u \bullet v \mapsto \mathsf{true} \in f) \wedge (\exists_1\, v : UNIV \bullet v \overset{p_1 \mapsto p_2}{\leadsto} u \bullet v \mapsto \mathsf{true} \in f')$ iff $u \mapsto \mathsf{true} \in h_2$. Since of the unique existential quantifier, the conjunction could have been pulled out.

$$(e[p_1 \mapsto p_2] \cap e'[p'_1 \mapsto p'_2])[p_3 \mapsto p_4] \equiv$$
$$e[p_1 \mapsto p_2][p_3 \mapsto p_4] \cap e'[p'_1 \mapsto p'_2][p_3 \mapsto p_4]$$
$$\textbf{where } e, e' : EXP;\ p_1, p_2, p'_1, p'_2, p_3, p_4 : PAT;\ \gamma : EXPASS \mid$$
$$p_2 =^{\gamma} p'_2;\ \mathsf{vars}\ (\mathsf{subs}\ \gamma\ p_1) \cap \mathsf{vars}\ (\mathsf{subs}\ \gamma\ p'_1) = \varnothing$$

Figure 3.27: Translation Product Law

Case $u \mapsto \mathsf{false}$**:** similar.

□

Distributivity of Hiding over Product

A hiding applied to a product can be distributed. A product is an intersection of translated sets which are independent (Figure 3.27). The independence of the operands is checked by unifying their target translation patterns, und stating that the source patterns under the resulting substitution have disjoint variables.

A formal proof is omitted, but an analogy to predicate calculus can serve as an intuition for the validity of the law. In predicate calculus the following holds: $\exists x, y \bullet P\,x \vee Q\,y \Leftrightarrow (\exists x, y \bullet P\,x) \vee (\exists x, y \bullet Q\,y)$, provided that y is not free in $P\,x$ and x is not free in $Q\,y$. A similar situation is described here.

The product law, in combination with the constructor decomposition law (Figure 3.24 (on page 60)), provides a generalization of decomposition in unification. Consider an equality such as $\rho(a, b) = \rho(c, d)$. Removing the syntactic sugar yields in

$$(\{\rho(a, b)\} \cap \{\rho(c, d)\})[x \mapsto \diamond]$$

Applying the constructor decomposition law yields:

$$(\{a\}[x \mapsto \rho(x, y)] \cap \{b\}[y \mapsto \rho(x, y)] \cap \{c\}[x \mapsto \rho(x, y)] \cap \{d\}[y \mapsto \rho(x, y)]) \quad [x \mapsto \diamond]$$

Next we apply associativity of intersection and distributivity of embeddings to get

$$((\{a\} \cap \{c\})[x \mapsto \rho(x, y)] \cap (\{b\} \cap \{d\})[y \mapsto \rho(x, y)])[x \mapsto \diamond]$$

Now, since the outer intersection is a product, we can apply the distributivity of the hiding $x \mapsto \diamond$. The resulting consecutive translations are immediately contracted such that we finally get:

$$(\{a\} \cap \{c\})[x \mapsto \diamond] \cap (\{b\} \cap \{d\})[y \mapsto \diamond]$$

Folding the syntactic sugar for equality, we reach at $a = c \cap b = d$.

Lifting of Hiding

A hiding as an operand of an intersection can be lifted out of the intersection (Figure 3.28 (on the next page)). The hidden part is introduced by an embedding for the right operand,

$$e[p_1 \mapsto p_2] \cap e' \equiv (e \cap e'[p_2 \mapsto p_1])[p_1 \mapsto p_2] \quad \text{, if } \mathsf{vars}\, p_2 \subset \mathsf{vars}\, p_1$$
$$\textbf{where } e, e' : EXP;\; p_1, p_2 : PAT$$

Figure 3.28: Translation Lifting Law

$$e[p_1 \mapsto p_2] \equiv e[p_1 \mapsto ((_) \circ \overline{x})][((_) \circ \overline{x}) \mapsto p_2]$$
$$\textbf{where } e : EXP;\; p_1, p_2 : PAT;\; \overline{x} : \text{seq}\, VAR \mid$$
$$\neg\; \mathsf{vars}\, p_1 \subseteq \mathsf{vars}\, p_2;\; \neg\; \mathsf{vars}\, p_2 \subseteq \mathsf{vars}\, p_1;\; \mathsf{ran}\, \overline{x} = \mathsf{vars}\, p_1 \cup \mathsf{vars}\, p_2$$

Figure 3.29: Translation Splitting Law

e', by just reverting the translation. On the intersection, the hiding is then performed again.

We do not provide a formal proof, but just point to an analogy in predicate calculus. $(\exists x \bullet P\, x) \wedge Q$ is equivalent to $\exists x \bullet P\, x \wedge Q$, provided x is not free in Q. Lifting of hiding is a generalization of this.

Splitting into Embedding and Hiding

A translation which is neither hiding nor embedding can be splitted into an embedding and a consecutive hiding (Figure 3.29). The correctness of this law follows directly from the law of composition of translations (Figure 3.25 (on page 60)): unifying $(\overline{x})$ against itself yield in an empty assignment, henceforth applying the composition law of translation yields in $e[\mathsf{subs}\, \varnothing\, p_1 \mapsto \mathsf{subs}\, \varnothing\, p_2]$, which equals to the left hand side expression in Figure 3.29.

It should be noted that since embeddings can be distributed and hidings can be lifted out of intersections, meanwhile other translations can be decomposed into embeddings and hidings, it is possible to normalize expressions such that embeddings are pushed to the leafs and hidings are pulled to the top of intersections.

Intersection Elimination

Two translated sets in intersection, with target patterns which cannot be unified, reduce to the empty set (Figure 3.30).

Note that in combination with the constructor decomposition law (Figure 3.24 (on page 60)) this law leads to the reduction of intersections $\{\rho(\overline{e})\} \cap \{\rho'(\overline{e}')\}$, with $\rho \neq \rho'$, to the empty set.

$$e[p_1 \mapsto p_2] \cap e'[p_1' \mapsto p_2'] \equiv \mathtt{0} \quad \text{, if } \neg\, p_2 =^{\gamma} p_2'$$
$$\textbf{where } e, e' : EXP;\; p_1, p_2, p_1', p_2' : PAT;\; \gamma : EXPASS$$

Figure 3.30: Translation Intersection Elimination Law

$$
\begin{array}{lll}
\{p \mid 0\} & \equiv & 0 \\
\{p \mid 1\} & \equiv & 1[p \mapsto p] \\
\{p \mid e \cap e'\} & \equiv & \{p \mid e\} \cap \{p \mid e'\} \\
\{p \mid e \cup e'\} & \equiv & \{p \mid e\} \cup \{p \mid e'\} \\
\{p \mid \sim e\} & \equiv & (\sim \{p \mid e\})[p \mapsto p] \\
\multicolumn{3}{l}{\quad \textbf{where } e, e' : \mathit{EXP};\ p : \mathit{PAT}}
\end{array}
$$

Figure 3.31: Schema Elimination and Distributivity Laws

3.4.6 Schema Laws

Elimination and Distributivity

A schema with a false property can be reduced to the empty set, one with a true property to a translated universal set preserving the pattern (this is necessary for the case the pattern is not exhaustive). Schema construction distributes over set union, intersection, and complement (Figure 3.31). The distributivity over complement requires a translation of the resulting set with $p \mapsto p$ for the case that p is not exhaustive. In this case, the negated schema denotes also those values which do not fit to p. These are filtered out by the translation.

*Proof***.** We prove one of the distributivity laws. In order to represent the property e of a schema, $\{p \mid e\}$, functions $F \in \mathit{UNIASS} \longrightarrow \mathsf{sem}\,\langle\rangle \twoheadrightarrow \mathbb{B}$ are used: $F\ \iota$ yields the characteristic Boolean-valued function of a property in dependency of a value assignment ι.

We want to show that the pair $u \mapsto b$ is in the characteristic function $h_1 = \{p \mid F_1 \cup F_2\}$ iff it is in the function $h_2 = \{p \mid F_1\} \cup \{p \mid F_2\}$. Suppose there is not match $p \rhd^{\iota} u$. Then obviously $u \mapsto \mathsf{false} \in h_1$ iff $u \mapsto \mathsf{false} \in h_2$. Suppose there is a match, and let it be ι:

Case $u \mapsto \mathsf{true}$**:** We have $u \mapsto \mathsf{true} \in h_1$ iff $\diamond \mapsto \mathsf{true} \in (F_1\ \iota \cup F_2\ \iota)$ (where $\diamond$ represents the constructor of the unit value), iff $\diamond \mapsto \mathsf{true} \in F_1\ \iota \vee \diamond \mapsto \mathsf{true} \in F_2\ \iota$, iff $u \mapsto \mathsf{true} \in h_2$.

Case $u \mapsto \mathsf{false}$**:** dual.

The proof for intersection is similar.

□

Complement Lifting

We have described how schema distributes over complement. The opposite of this law is given in Figure 3.32 (on the facing page). The operand $\sim (1[p \mapsto p])$ adds those values which are in the result set since they do not match the pattern p. For exhaustive patterns it reduces to 0, since a translation by the same exhaustive pattern can be eliminated.

$$\sim \{p \mid e\} \equiv \{p \mid \sim e\} \cup \sim (1[p \mapsto p])$$
$$\textbf{where } e : \mathit{EXP};\ p : \mathit{PAT}$$

Figure 3.32: Schema Complement Lifting Law

$$\{p \mid e\}[p_1 \mapsto p_2] \equiv \mathsf{0} \qquad , \textbf{if} \neg\, p =^{\gamma} p_1$$
$$\{p \mid e\}[p_1 \mapsto p_2] \equiv \{\mathsf{subs}\ \gamma\ p_2 \mid \mathsf{subs}\ \gamma\ e\} \quad , \textbf{if}\, p =^{\gamma} p_1 \wedge \mathsf{vars}\, p_1 \subseteq \mathsf{vars}\, p_2$$
$$\textbf{where } e : \mathit{EXP};\ p, p_1, p_2 : \mathit{PAT};\ \gamma : \mathit{EXPASS}$$

Figure 3.33: Schema Translation Elimination Laws

Translation Elimination

Any translation of a schema where the schema's pattern and the translation's source pattern cannot be unified reduces to the empty set. An embedding translation on a schema can be eliminated (Figure 3.33).

Membership Lifting

A schema which contains a membership-test, $p \in e$, where p is linear and contains only variables from the schema's pattern, can be eliminated provided the expression e does not contain variables dependent on the schema's pattern (Figure 3.34).

Note that membership lifting also captures equality: $\{x \mid x = e\}$ is an abbreviation for $\{x \mid x \in \{e\}\}$, which eventually reduces to $\{e\}$ provided x is not free in e.

Property Substitution

A schema intersected with a translated singleton set is equivalent to an expression where the singleton set's element is "imported" by substitution into the schema's property (Figure 3.35 (on the next page)). Since we can always add the identity translation, $x \mapsto x$, the law captures also the case where no real translation of the singleton set is involved. Since other laws have ensured that embedding translations applied to schemas can be eliminated, and hiding translations can be lifted out of intersections, we do not need to consider the case of the translation of the schema itself.

Proof (informal). Recall that $\mathsf{mksubs}\,(p, e)$ is defined as

$$\lambda\, p : \mathit{PAT};\ e : \mathit{EXP} \bullet \lambda\, x : \mathsf{vars}\, p \bullet \mu\,(\{e\}[p \mapsto x]).$$

$$\{p \mid p' \in e\} \equiv e[p' \mapsto p]$$
$$\textbf{where } e : \mathit{EXP};\ p, p' : \mathit{PAT} \mid$$
$$p' \in \mathit{PAT}_{\text{LIN}};\ \mathsf{vars}\, p' \subseteq \mathsf{vars}\, p;\ \mathsf{vars}\, e \cap \mathsf{vars}\, p = \varnothing$$

Figure 3.34: Schema Membership Lifting Law

$$\{p \mid e\} \cap \{e'\}[p_1 \mapsto p_2] \equiv$$
$$\{\mathsf{subs}\ \gamma\ p \mid \mathsf{subs}\,(\mathsf{mksubs}\,(\mathsf{subs}\ \gamma\ p_1, e'))\ e\} \cap \{e'\}[p_1 \mapsto p_2]$$
$$\textbf{where}\ e, e' : \mathit{EXP};\ p, p_1, p_2 : \mathit{PAT};\ \gamma\ : \mathit{EXPASS} \mid p_2 =^{\gamma} p$$

Figure 3.35: Schema Property Substitution Law

$$\mathbf{fix}\, p \lhd e \equiv\ \mathsf{subs}\,(\mathsf{mksubs}\,(p, \mathbf{fix}\, p \lhd e))\ e$$
$$\textbf{where}\ p : \mathit{PAT};\ e : \mathit{EXP} \mid$$
$$\forall\, \Omega : \mathit{SEMCTX} \mid \mathbf{fix}\, p \lhd e \in \mathsf{dom}(\mathsf{sem}\ \Omega) \bullet$$
$$\exists_1\, D == \mathsf{sem}\ (\mathsf{subs}\,\Omega.1\ (\mathsf{type}\, p)) \bullet$$
$$(\lambda\, u : D \bullet\ \textbf{let}\ \iota\ : \mathit{UNIASS} \mid p \rhd^{\iota}\ u\ \bullet\ \mathsf{sem}\ (\Omega.1, \Omega.2 \oplus \iota\,)\ e)$$
$$\in D \twoheadrightarrow D$$

Figure 3.36: Fixed-Point Unrolling Law

In Figure 3.35, the selection of a component from e' will simplify if $\mathsf{subs}\ \gamma\ p_1$ is a simple variable, name it x: then we have the substitution

$$\mathsf{mksubs}\,(x, e') = \{x \mapsto \mu\ (\{e'\}[x \mapsto x])\},$$

and, after further simplification steps, the replacement of x in the schema's property e reduces to e'. However, if $\mathsf{subs}\ \gamma\ p_1$ is a proper pattern, then the selections via the μ operator in the substituted expressions may be in principle undefined. Yet, in this case, though the property $\mathsf{subs}\,(\mathsf{mksubs}\,(\mathsf{subs}\ \gamma\ p_1, e'))\ e$ of the schema becomes undefined, the singleton set expression $\{e'\}[p_1 \mapsto p_2]$ must also reduce to $\mathsf{0}$, such that the undefinedness of the property does not count. □

3.4.7 Fixed-Point Unrolling Law

The fixed-point operator, $\mathbf{fix}\, p \lhd e$, can be *unrolled*, provided the expression e represents a continuous function on the domain of the pattern's type (Figure 3.36). This law differs from all others in that the application condition is semantic. The pragmatics is that the unrolling law is used in the operational semantics of μZ under the unproved *assumption* of the above condition, and is thus correct only relative to such assumptions.

Proof (informal). The correctness of this law follows from the fixed-point theorem for lattices discussed in Section 2.3.4 (on page 33). By the typing conditions, we know that p is an exhaustive pattern where the variables represent set types, and thus its type's domain is a complete lattice. □

3.5 Discussion

We have presented the μZ calculus, which provides a way of viewing the λ-calculus, predicate calculus, schema calculus, set algebra, etc., in a unified framework. A set-based model for the calculus has been defined, syntax, typing conditions and semantics have been provided, and a set of equational laws has been given. A few further points are discussed below, including the relation to other work.

Partial Sets and Recursion

Features like sets with unknowns and the explicit representation of undefinedness makes μZ's model different from the one normally used for Z. However, as discussed in Section 2.1.4 (on page 26), the forthcoming ISO Standard does not prescribe a particular position on undefinedness. The treatment of undefinedness is left to conventions introduced by specifiers and/or tools.

Representing undefinedness in the model of μZ is *essential*. The need for its representation is usually attributed to computation. For example, the reason that VDM [Plat and Larsen, 1992; Larsen and Pawlowski, 1995] represents undefinedness, unlike Z, is often explained by its need to deal with computation. For μZ, the more important argument is that explicit representation of undefinedness is the key to a simple equational theory, which can be applied without testing guards for definedness (which are, in reality, intractable). Central laws, such as the distributivity of schema construction over conjunction, $\{p \mid e_1 \cap e_2\} \equiv \{p \mid e_1\} \cap \{p \mid e_2\}$, would otherwise become more complex. Other laws affected are property substitution (related to β-reduction or unfolding), constructor decomposition and many more. The observation can be compared with the cleaner equational properties of lazy functional languages as opposed to strict ones.

It is notable that, although we have undefinedness, it is decoupled from fixed-point construction, which is based on classical lattice theory. In fact, the least element for fixed-point construction is the empty set, not the undefined set. Building fixed-points would also work for models without undefinedness.

Since μZ's approach is not constructive *a priori*, a fixed-point may be undefined. At the level of the model semantics, this causes no problems. In the equational theory, however, we have to place conditions about continuity of the expression e in $\mathbf{fix}\, p \lhd e$ over the variables from p to allow symbolic computation of fixed-points. The application condition of this law is indeed intractable, but in practice – when mapping Z to μZ and converting some recursive equations into the fixed-point form – a pragmatic solution is acceptable, which adds the unproved *assumption* that the condition for the fixed expression is obeyed.

Set Algebra

The μZ calculus is based on set algebra, and its operators are well known, except that they are interpreted with a partial semantics. A new contribution is the combination of set-algebraic operators with the translation construct. Translation is used to model micro-elements, like function application and quantifiers, and at the same time it scales up to describe morphisms between macro-elements, like entire specifications. Translations are a way of describing morphisms in the category of sets of the μZ universe, and some of

the equational laws seem to have a direct correspondence to concepts in category theory [Barr and Wells, 1990]. However, the relation to category theory needs to be further investigated.

Sets In Programming Languages

Sets for use in programming were proposed by the designers of SETL back in the eighties [Kruchten et al., 1984]. Recently, sets in declarative programming have again attracted interest in the community. Several set-oriented programming languages have been defined. This work usually adds special primitives to functional or logic languages, providing predicates for membership or subset relation and introducing non-free constructors for decomposing sets in pattern-based rule definitions. One approach along these lines is the language $\{log\}$ (pronounced SETLOG), which is an extension of Prolog by special terms for set expressions, including extensional and intensional representation, and encapsulated search [Dovier et al., 1996; Rossi, 1997]. Another interesting approach here is *subset logic programming*, described e.g. in [Jana and Jayaraman, 1999; Jayaraman and Moon, 1999], which allows for recursive subset clauses.

μZ differs from these approaches in that it embeds functional and logic concepts into *sets* and not vice versa, thereby using a pure expression language in the spirit of the λ-calculus. This makes μZ suitable as an intermediate language for the integration of paradigms. On the other hand, as will be seen in Chapter 4, the computation model for μZ presented in this thesis is not as ambitious as the one for subset logic programming [Moon, 1997], which (in a first-order setting) employs set unification and fixed-points over subset constraints, quite expensive features [Arenas-Sanchez and Dovier, 1997; Stolzenburg, 1999]. These symbolic computation techniques are, in principle, entailed in μZ's model, but no attempt has been made to formulate equational laws which reflect them.

Higher-Order Logic

μZ is related to higher-order logic [Gordon and Melham, 1993] in its ambition to reduce mechanical reasoning to a small calculus. However, μZ has more redundancy, since it contains extensional elements (singleton sets) and set algebra, whereas basic HOL is purely intensional, and uses only abstraction and application as the kernel expression forms. In [Santen, 1999], it has been shown that Z can be consistently embedded into HOL, with the practical effect that tools like Isabelle [Paulson, 1994] can be used for mechanical reasoning. Here we propose that the redundancy of μZ, supporting extensional description and set algebra, is in fact an advantage, because it allows the construction of kernel machines that are specialized for the basic concept that is of interest in reasoning – boolean algebras. This is at least indicated for the applications of partial evaluation and (symbolic) computation, as will be investigated in the next chapter.

Concurrent Constraint, Functional Logic Languages

Most authors presenting implementation techniques for concurrent constraint or functional logic languages base their work on a kernel language which is usually an extension of a subset of ML or Haskell. Special "choice" constructs (generalizations of the **case** construct of functional languages) introduce indeterminism, and logic variables can be

declared [Kuchen et al., 1990; Lock, 1992]. In languages that support concurrent constraints, collections of constraints can also be used in position of Boolean expressions [Chakravarty et al., 1997; Hanus, 1999; Mehl, 1999]. These kernel languages are often tailored to the application in hand. In contrast, the μZ calculus uses a more general approach, since it is based on standard notions of set algebra.

An approach similar to that of μZ are the calculi for the Oz language [Smolka, 1998], described in [Smolka, 1994a,b]. The γ-Calculus, Calculus A and Calculus B use a language over conjunctions and disjunctions of constraints. Basic constraints are declarations of predicates and functions, as well as equalities and applications of predicates, and committed choice. The calculi are pure expression languages, similar to μZ. An equational theory is given which serves to define the operational semantics modulo expression congruences. As will be discussed in Chapter 4, the computation model has some similarities to that of μZ.

μZ differs from the Oz calculi in that it uses set algebra instead of predicate logic. It incorporates extensional representation of disjunctive information, which the Oz calculi do not support. On the other hand, μZ currently has no equivalent of committed choice, which is important for representing *explicit* concurrency. Committed choice *cannot* be represented as syntactic sugar in μZ. By a way of an attempt, let $g_1 \to e_1 \oplus g_2 \to e_2$ be a choice in μZ, where g_1 and g_2 are Boolean guards and the meaning is defined as

$$\begin{array}{l}\{x \mid g_1 \cap \sim g_2 \cap x \in e_1\} \cup \{x \mid g_2 \cap \sim g_1 \cap x \in e_2\} \cup \\ \{x \mid g_2 \cap g_2 \cap x \in e_1 \cup e_2\}\end{array}$$

The problem is the case where both g_1 and g_2 are true: in a committed choice, we expect that either e_1 or e_2 is chosen nondeterministically. This kind of non-determinism is “don’t care”, which is not supported by μZ: the calculus is, in fact, purely declarative, unfolding “don’t know” indeterminism into sets. Thus, the property $x \in e_1 \cup e_2$ encounters *all* possibilities in the union of e_1 and e_2. How choice can be incorporated into μZ requires further research. A more general approach than only supporting choice is aimed at, based on temporal interval logic [Büssow and Grieskamp, 1997]. This will be discussed under future work in the conclusion in Chapter 7.

Daß man nicht weiß,
Was man denkt,
Wenn man denkt;
alles ist als wie geschenkt.
(Goethe)

Chapter 4

Computation in μZ

Models for computation in μZ are investigated in this chapter. The principal goal of computation is seen to be the *transformation* of a μZ expression into a "more" extensional but equivalent form. The process, thereby, is incremental: given a set expression e, one computes, for example, the equivalent expression $\{e_1\} \cup e_2$. Computation may then be continued with e_1 (in case it is not already completely reduced) or with e_2.

In general, the pragmatics of computation differs if it is performed "on-line" or "off-line"[1]. On-line computation works by direct symbolic *interpretation* of expressions, whereas off-line computation involves *compilation* steps, targeting at a tailored encoding of expressions and values at execution time. In this chapter, we will have this distinction not explicitly: all the computation models we investigate are symbolic, in the sense that they work on expressions (or slight extensions of them). However, some models are better suited to off-line computation and compilation, as we shall see.

Underlying every model of computation we investigate is a *normalization* transformation which establishes a *disjunctive normal form*, including simplifications of expressions (Section 4.2 (on page 74)). The normalization already exploits most of the equational laws given for μZ in Chapter 3. It is confluent and terminating, which is important to make it applicable to the partial evaluation and optimization of μZ programs at compile time. Indeed, since it terminates, it has to leave out laws such as the fixed-point unrolling law.

We obtain a *narrowing* semantics of μZ by adding to normalization some further reductions, which are based on the property substitution law, membership lifting law, and fixed-point unrolling law (Section 4.3 (on page 82)). By constraining the reduction order of narrowing to outermost-in and strict narrowing, we obtain some interesting submodels.

The narrowing semantics is well suited to on-line computation, however, not to off-line computation (as will be discussed). Therefore, a further model is developed, using the style of *natural semantics*, and constituting μZ's *reference computation model*, RCM (Section 4.4 (on page 86)). This model defines computation by making a distinction between (strict, functional) *deterministic reduction* and (logic, concurrent) *indeterministic resolution*, and prepares a straight transition to the abstract machine defined in the next chapter. The chapter concludes with a discussion of the results obtained and of related work.

[1]The notions "on-line" and "off-line" are taken from the realm of partial evaluation (e.g., Consel and Danvy [1993]).

$$| _ \sim _ : EXP \leftrightarrow EXP$$

$$\frac{\forall i : \mathsf{dom}\, \overline{e}_1 \bullet \overline{e}_1\, i \sim \overline{e}_2\, i}{\rho(\overline{e}_1) \sim \rho(\overline{e}_2)} \quad {\scriptstyle \rho:CONS \atop \overline{e}_1, \overline{e}_2 : \mathrm{seq}\, EXP}$$

$$\frac{\begin{array}{c}\forall i : \mathsf{dom}\, \overline{e}_1 \bullet \exists j : \mathsf{dom}\, \overline{e}_2 \bullet \overline{e}_1\, i \sim \overline{e}_2\, j \\ \forall j : \mathsf{dom}\, \overline{e}_2 \bullet \exists i : \mathsf{dom}\, \overline{e}_1 \bullet \overline{e}_2\, j \sim \overline{e}_1\, i\end{array}}{\bigcup((\{_\}) \circ \overline{e}_1) \sim \bigcup((\{_\}) \circ \overline{e}_2)} \quad {\scriptstyle \overline{e}_1, \atop \overline{e}_2 : \mathrm{seq}\, EXP}$$

$$| _ \not\sim _ : EXP \leftrightarrow EXP$$

$$\frac{\rho_1 \neq \rho_2 \vee (\exists i : \mathsf{dom}\, \overline{e}_1 \bullet \overline{e}_1\, i \not\sim \overline{e}_2\, i)}{\rho_1(\overline{e}_1) \not\sim \rho_2(\overline{e}_2)} \quad {\scriptstyle \rho_1, \rho_2 : CONS \atop \overline{e}_1, \overline{e}_2 : \mathrm{seq}\, EXP}$$

$$\frac{\begin{array}{c}(\exists i : \mathsf{dom}\, \overline{e}_1 \bullet \forall j : \mathsf{dom}\, \overline{e}_2 \bullet \overline{e}_1\, i \not\sim \overline{e}_2\, j) \\ \vee (\exists j : \mathsf{dom}\, \overline{e}_2 \bullet \forall i : \mathsf{dom}\, \overline{e}_1 \bullet \overline{e}_2\, j \not\sim \overline{e}_1\, i)\end{array}}{\bigcup((\{_\}) \circ \overline{e}_1) \not\sim \bigcup((\{_\}) \circ \overline{e}_2)} \quad {\scriptstyle \overline{e}_1, \overline{e}_2 : \mathrm{seq}\, EXP}$$

Figure 4.1: Extensional Equality and Unequality

4.1 Preliminaries

Some preliminary definitions are given in this section: extensional equality, extensional confluence and matching contexts.

Extensional Equality

We introduce the notion of *extensional equality* of expressions. This (partial) equality identifies expressions that are built from constructor terms and union of singleton sets; it cannot identify schemas and translations. Thus, the relation $_ \sim _$ is an approximation of expression equivalence, such that $e_1 \sim e_2 \Rightarrow e_1 \equiv e_2$ holds. Since it is approximative, the negation $\neg\ e_1 \sim e_2 \Rightarrow \neg\ e_1 \equiv e_2$ does *not* hold. We therefore define a second relation, *extensional inequality*, written as $e_1 \not\sim e_2$, which obeys the property $e_1 \not\sim e_2 \Rightarrow \neg\ e_1 \equiv e_2$. Both relations are defined by the rule system in Figure 4.1.

The use of $\bigcup$ requires that unions be bracketed to the right (cf. the definition of $\bigcup$, Section 3.3.5 (on page 56)), and that the empty set be removed in a non-empty union. Since equality will only be applied to normalized expressions that puts unions into such a form, this constraint is feasible.

No attempt is made, by the given notion of equality, to identify further expression forms. It is generally assumed that intersection is resolved before equality is tested. Sets described by translation, complement and, in particular, by schemas cannot be identified.

Extensional Confluence

On the basis of the equality relation $_ \sim _$, the notion of an expression relation R being *extensionally confluent* is defined. For such a relation, all terminating derivations that lead to an expression in the domain of $_ \sim _$ must yield equal results under $_ \sim _$. Thus, if

$$
\begin{array}{l}
[A] \\
\mathrm{SCTX}\,_ : \mathbb{P}(\mathrm{seq}\,A \longrightarrow \mathrm{seq}\,A) \\
\hline
\mathit{UNCTX} == \{\mathcal{S} : \mathrm{SCTX}\,\mathit{EXP} \bullet \lambda\, s : \mathrm{seq}\,\mathit{EXP} \bullet \bigcup(\mathcal{S}\ s)\} \\
\mathit{INCTX} == \{\mathcal{S} : \mathrm{SCTX}\,\mathit{EXP} \bullet \lambda\, s : \mathrm{seq}\,\mathit{EXP} \bullet \bigcap(\mathcal{S}\ s)\}
\end{array}
$$

Figure 4.2: Matching Contexts

$e\ \underline{R}\ \ldots\ \underline{R}\ e_1$ and $e\ \underline{R}\ \ldots\ \underline{R}\ e_2$, where $e_1, e_2 \notin \mathsf{dom}\,R$, are terminating derivations, then $e_1 \in \mathsf{dom}(_\sim_) \vee e_2 \in \mathsf{dom}(_\sim_) \Rightarrow e_1 \sim e_2$. This notion of confluence reflects the fact that as long as "symbolic" computation is done, confluence is not asserted; only if a derivation leads to an extensional form it counts.

Matching Contexts

A *sequence matching context* serves for the convenient formulation of rewriting rules. With a sequence matching context, we can match one or more elements out of a sequence and replace them by new elements or remove them. For example, if $\mathcal{S}\,\langle 2, 3\rangle$ matches some sequence containing 2 and 3 at arbitrary positions, then $\mathcal{S}\,\langle -2, -3\rangle$ matches the sequence where one occurence of 2 is replaced by -2 and one occurence of 3 is replaced by -3. $\mathcal{S}\,\langle -2\rangle$ is the sequence where the occurence of 2 is replace by -2, and the occurence of 3 is removed. Sequence matching context $\mathcal{S} \in \mathrm{SCTX}\,\alpha$ (α the element type of the sequence) are declared in Figure 4.2.

For the common case of sequence matching in combination with generalized union and intersection, in Figure 4.2 we also define *union matching contexts* $\dot{\bigcup} \in \mathit{UNCTX}$ and *intersection matching contexts* $\dot{\bigcap} \in \mathit{INCTX}$. $\dot{\bigcup}\,\overline{e}$ abbreviates $\bigcup(\mathcal{S}\,\overline{e})$ for some sequence matching context $\mathcal{S}$ (similar for intersection).

$$
\begin{array}{llll}
b \in EXP_B & ::= & d \mid \rho(\overline{b}) \mid \mu\, d \mid x & \textit{expressions} \\
d \in EXP_D & ::= & \bigcup \overline{c} & \textit{disjunctions} \\
c \in EXP_C & ::= & \bigcap \overline{l} & \textit{conjunctions} \\
l \in EXP_L & ::= & \sim a \mid a & \textit{literals} \\
a \in EXP_A & ::= & 0 \mid x \mid \{p \mid \phi\} \mid c[p_1 \mapsto p_2] \mid & \textit{atoms} \\
 & & \mu\, d \mid \mathbf{fix}\, p \lhd d \mid \{b\} & \\
\phi \in EXP_P & ::= & p_1 = p_2 \mid p \in c \mid ?_1 c \mid ?_0 c & \textit{properties}
\end{array}
$$

Figure 4.3: Normalized Expressions

4.2 Normalization

Underlying every model of computation we investigate is a transformation that establishes a disjunctive normal form, including simplifications of expressions. Normalization is described by the relation $_ \rightarrow_{\mathcal{S}} _$:

$$| _ \rightarrow_{\mathcal{S}} _ : EXP \leftrightarrow EXP$$

Applying this relation as a context rewriting establishes a *disjunctive normal form* as described by the grammar in Figure 4.3. The DNF is justified by the de-Morgan laws of the calculus. In addition to the DNF form, some further assertions hold about normalized expressions:

- Any translation that is neither hiding nor embedding has been decomposed into an embedding followed by a hiding. Hidings are pulled, embeddings pushed and consecutive translations contracted. Complement is pushed as far as possible – justified by the lifting and distributivity laws of translation

  Note that hiding translations cannot be completely pulled out of expressions, since complement is not distributive w.r.t. hiding; thus, for example, the following expression is in normal form: $c \cap \sim((c_1 \cap c_2)[p_1 \mapsto p_2])$.

- Potential for simplification is exploited to a maximum: the only laws not considered for this purpose are membership lifting, property substitution and fixed-point unrolling.

- Schema properties are decomposed into one of the forms: pattern equality, $p_1 = p_2$, pattern membership, $p \in c$, or test for emptiness and nonemptiness, $?_1 c$ or $?_0 c$. The variables appearing in the patterns p, p_1, and p_2 must be directly bound by the enclosing schema, and the patterns must be linear. In fact, after removing the syntactic sugar, the forms overlap. However, for the sake of simplicity, it will be assumed in the sequel that the property forms can be distinguished.

4.2.1 Property Normalization

We take a closer look at the normalization of schema properties. The benefit of having a schema property in the form $\{p \mid p' \in c\}$ is that the membership lifting law (Figure 3.34 (on page 65)) can be applied as soon as c does not depend on variables from p, yielding the expression $c[p' \mapsto p]$[2].

Normalization ensures that any so-called *flexible* appearances of variables in a property are converted into a membership property, $p \in c$, together with a sequence of simple equations of flexible variables, $x = x'$. By a flexible appearance, we mean a variable x used as in $\{P\,x \mid C\,\{\Pi\,x\}\}$, where C is a positive conjunctive context built from intersections and translations, and Π a context of constructor applications.

The normalization of flexible variables works as follows. Let $\{p \mid ?_1(\bigcap \bar{l})\}$ be a schema with a positive conjunctive property as an intermediate result of normalization, where constructor decomposition has been performed, hidings have been pulled and absorbed by the overall hiding of $?_1_$, and embeddings have been pushed to the leaves. Henceforth, $?_1(\bigcap \bar{l})$ must match (modulo associativity and commutativity)

$$?_1(\{x_1\}[p_1 \mapsto q_1] \cap \ldots \cap \{x_n\}[p_n \mapsto q_n] \cap c)$$

where the x_i are flexible and where c does not contain variables in a flexible position. Next, we unify the patterns q_i against each other (if this fails, then the property reduces to $\mathsf{0}$ already at normalization time). Let γ be the resulting assignment, and let $\gamma' = \{x_1 \mapsto \mathsf{subs}\ \gamma\ p_1, \ldots, x_n \mapsto \mathsf{subs}\ \gamma\ p_n\}$ be an assignment for the flexible variables x_i, and $q = \mathsf{subs}\ \gamma\ q_1$ the unified translation target. Let $\bar{x} = \mathsf{vars}\,q \cap \bigcup(\mathsf{vars}\ (\!|\mathsf{ran}\,\gamma'|\!))$ be the variables in q that are bound to the flexible variables (q can contain further variables introduced by embeddings). Then, we construct the schema

$$\{\mathsf{subs}\ \gamma'\,p \mid (\bar{x}) \in (\mathsf{subs}\ \gamma'\,c)[q \mapsto (\bar{x})]\}.$$

However, this schema might not be well-formed, since $\mathsf{subs}\ \gamma'\,p$ may result in a non-linear pattern. But in this case, a simple renaming of double occurrences of variables can be performed, adding aliases like $x = x'$ for each renamed variable.

By way of an example, consider the schema $\{x \mid (e, x) \in e'\}$. This results in the normalized schema

$$\{x \mid x \in (\{e\}[x \mapsto (x, _)] \cap e')[(_, y) \mapsto y]\}.$$

As a further example, demonstrating aliasing, consider the schema $\{(x, y, z) \mid (e, x) = (y, z) \in e'\}$. For the property one literally gets

$$?_1(\{x\}[x \mapsto (_, x)] \cap \{y\}[x \mapsto (x, _)] \cap \{z\}[x \mapsto (_, z)] \cap \{e\}[x \mapsto (x, _)] \cap e')$$

Unification of $(_, x)$ with $(y, _)$ and $(_, z)$ yields (y, z). The resulting schema is

$$\{(z, y, z') \mid (y, z) \in (\{e\}[x \mapsto (x, _)] \cap e')\ \cap\ z = z'\}.$$

In a subsequent normalization step, this is reduced to a conjunction of schemas, one holding the pattern membership, the other the alias.

[2]Membership elimination is not part of normalization – but the computation models based on normalized expressions use it as a central simplification step.

$$\mathcal{S}1\ \frac{}{\mu\,\{e\}\ \rightarrow_{\mathcal{S}}\ e}\ e{:}EXP$$

Figure 4.4: Singleton Set Normalization

$$\mathcal{S}2\ \frac{}{\{\rho\}\ \rightarrow_{\mathcal{S}}\ \{\diamond\}[x \mapsto \rho]}\ \begin{array}{l}\rho{:}CONS\\ x{:}VAR\end{array}$$

$$\mathcal{S}3\ \frac{\#\overline{e} = \#\overline{x} > 0}{\begin{array}{c}\{\rho(\overline{e})\}\ \rightarrow_{\mathcal{S}}\\ \bigcap\,((\lambda\, i : \mathbb{N} \bullet \{\overline{e}\, i\}[(\overline{x}\, i) \mapsto \rho((_) \circ \overline{x})]) \circ \mathsf{id}(1\,..\,\#\overline{e}))\end{array}}\ \begin{array}{l}\rho{:}CONS\\ \overline{e}{:}\mathsf{seq}\,EXP\\ \overline{x}{:}\mathsf{seq}\,VAR\end{array}$$

Figure 4.5: Constructor Normalization

4.2.2 Normalization Rules

The rewriting rules for simplification and normalization are defined in the sequel, using the notation style of inference rules. Though this kind of notation is particularly suitable for inductive definitions over the structure of expressions (which does not apply to the rewriting rules below because they are not recursive in their premises), it is used here because the application conditions of rules are sometimes complex, and are better readable when denoted in rule style.

Singleton-Set Elimination

Rule $\mathcal{S}1$ in Figure 4.4 resembles the reduction of singleton-set selection.

Constructor Decomposition

Rules $\mathcal{S}2$ and $\mathcal{S}3$ in Figure 4.5 handle the decomposition of singleton sets containing constructor terms (according to the laws in Figure 3.24 (on page 60)). Even 0-ary constructors are transformed into translations, since incompatible constructions (e.g. $\{\rho\} \cap \{\rho'\}$ with $\rho \neq \rho'$) are detected by the intersection of translations (Rule $\mathcal{S}16$).

Union

Rules $\mathcal{S}4$ to $\mathcal{S}8$ in Figure 4.6 (on the facing page) model the simplification of set union based on Boolean laws (Figure 3.21 (on page 58)). Unions are rebracketed such that the generalized union, $\bigcup$, can apply. Tautologies are reduced and duplicate elements are removed. The "excluded middle" for singleton sets is tackled by rule $\mathcal{S}8$.

Intersection

Rules $\mathcal{S}9$ to $\mathcal{S}15$ in Figure 4.7 (on the facing page) model the simplification and normalization of set intersection based on boolean laws (Figure 3.21 (on page 58)). As for union, intersections are rebracketed, tautologies are reduced and duplicate elements are removed. The "excluded middle" for singleton sets is tackled by rule $\mathcal{S}13$. Rule $\mathcal{S}15$ distributes intersection over union.

$$\mathcal{S}4\,\frac{}{(e_1 \cup e_2) \cup e_3 \;\to_{\mathcal{S}}\; e_1 \cup (e_2 \cup e_3)}\; e_1, e_2, e_3{:}EXP$$

$$\mathcal{S}5\,\frac{}{\dot{\bigcup}\langle 0, e\rangle \;\to_{\mathcal{S}}\; \dot{\bigcup}\langle e\rangle}\; \begin{matrix} e{:}EXP \\ \dot{\bigcup}{:}UNCTX \end{matrix} \qquad \mathcal{S}6\,\frac{}{\dot{\bigcup}\langle 1, e\rangle \;\to_{\mathcal{S}}\; 1}\; \begin{matrix} e{:}EXP \\ \dot{\bigcup}{:}UNCTX \end{matrix}$$

$$\mathcal{S}7\,\frac{e_1 \sim e_2}{\dot{\bigcup}\langle e_1, e_2\rangle \;\to_{\mathcal{S}}\; \dot{\bigcup}\langle e_1\rangle}\; \begin{matrix} e_1, \\ e_2{:}EXP \\ \dot{\bigcup}{:}UNCTX \end{matrix} \qquad \mathcal{S}8\,\frac{e_1 \sim e_2}{\dot{\bigcup}\langle \{e_1\}, \sim\{e_2\}\rangle \;\to_{\mathcal{S}}\; 1}\; \begin{matrix} e_1, \\ e_2{:}EXP \\ \dot{\bigcup}{:}UNCTX \end{matrix}$$

Figure 4.6: Union Normalization

$$\mathcal{S}9\,\frac{}{(e_1 \cap e_2) \cap e_3 \;\to_{\mathcal{S}}\; e_1 \cap (e_2 \cap e_3)}\; e_1, e_2, e_3{:}EXP$$

$$\mathcal{S}10\,\frac{}{\dot{\bigcap}\langle 0, e\rangle \;\to_{\mathcal{S}}\; 0}\; \begin{matrix} e{:}EXP \\ \dot{\bigcap}{:}INCTX \end{matrix} \qquad \mathcal{S}11\,\frac{}{\dot{\bigcap}\langle 1, e\rangle \;\to_{\mathcal{S}}\; \dot{\bigcap}\langle e\rangle}\; \begin{matrix} e{:}EXP \\ \dot{\bigcap}{:}INCTX \end{matrix}$$

$$\mathcal{S}12\,\frac{e_1 \sim e_2}{\dot{\bigcap}\langle e_1, e_2\rangle \;\to_{\mathcal{S}}\; \dot{\bigcap}\langle e_1\rangle}\; \begin{matrix} e_1, \\ e_2{:}EXP \\ \dot{\bigcap}{:}INCTX \end{matrix} \qquad \mathcal{S}13\,\frac{e_1 \sim e_2}{\dot{\bigcap}\langle \{e_1\}, \sim\{e_2\}\rangle \;\to_{\mathcal{S}}\; 0}\; \begin{matrix} e_1, \\ e_2{:}EXP \\ \dot{\bigcap}{:}INCTX \end{matrix}$$

$$\mathcal{S}14\,\frac{e_1 \not\sim e_2}{\dot{\bigcap}\langle e_1, e_2\rangle \;\to_{\mathcal{S}}\; 0}\; \begin{matrix} e_1, \\ e_2{:}EXP \\ \dot{\bigcap}{:}INCTX \end{matrix} \qquad \mathcal{S}15\,\frac{}{\dot{\bigcap}\langle e_1 \cup e_2\rangle \;\to_{\mathcal{S}}\; \dot{\bigcap}\langle e_1\rangle \cup \dot{\bigcap}\langle e_2\rangle}\; \begin{matrix} e_1, e_2{:}EXP \\ \dot{\bigcap}{:}INCTX \end{matrix}$$

$$\mathcal{S}16\,\frac{\neg\,(\exists\,\gamma : EXPASS \bullet p'_1 =^{\gamma} p'_2)}{\dot{\bigcap}\langle e_1[p_1 \mapsto p'_1], e_2[p_2 \mapsto p'_2]\rangle \;\to_{\mathcal{S}}\; 0}\; \begin{matrix} e_1, \\ e_2{:}EXP \\ p_1, p_2, p'_1, p'_2{:}PAT \\ \dot{\bigcap}{:}INCTX \end{matrix}$$

$$\mathcal{S}17\,\frac{\neg\,(\exists\,\gamma : EXPASS \bullet p'_1 =^{\gamma} p_2)}{\dot{\bigcap}\langle e_1[p_1 \mapsto p'_1], \{p_2 \mid e_2\}\rangle \;\to_{\mathcal{S}}\; 0}\; \begin{matrix} e_1, \\ e_2{:}EXP \\ p_1, p_2, p'_1{:}PAT \\ \dot{\bigcap}{:}INCTX \end{matrix}$$

$$\mathcal{S}18\,\frac{\neg\,(\exists\,\gamma : EXPASS \bullet p_1 =^{\gamma} p_2)}{\dot{\bigcap}\langle \{p_1 \mid e_1\}, \{p_2 \mid e_2\}\rangle \;\to_{\mathcal{S}}\; 0}\; \begin{matrix} e_1, \\ e_2{:}EXP \\ p_1, p_2{:}PAT \\ \dot{\bigcap}{:}INCTX \end{matrix}$$

$$\mathcal{S}19\,\frac{\mathsf{vars}\, p' \subset \mathsf{vars}\, p}{\dot{\bigcap}\langle e[p \mapsto p']\rangle \;\to_{\mathcal{S}}\; (e \cap (\dot{\bigcap}\langle\rangle)[p' \mapsto p])[p \mapsto p']}\; \begin{matrix} \dot{\bigcap}{:}INCTX \\ e{:}EXP \\ p, p'{:}PAT \end{matrix}$$

Figure 4.7: Intersection Normalization

Rules $\mathcal{S}16$ to $\mathcal{S}18$ handle the intersection of translations and/or schemas which cannot fit, because the corresponding patterns cannot be unified.

Rule $\mathcal{S}19$ models the lifting of hiding translations out of an intersection context (cf. Figure 3.28 (on page 63)).

Complement

Rule $\mathcal{S}20$ in Figure 4.8 (on the next page) absorbs consecutive complement. Rules $\mathcal{S}21$ and $\mathcal{S}22$ distribute set complement over union and intersection. Rule $\mathcal{S}23$ distributes complement over embedding translations that are exhaustive. Rule $\mathcal{S}24$ distributes complement over schema, adding those values that cannot match the schema's pattern.

$$\mathcal{S}20\ \frac{}{\sim(\sim e)\ \to_{\mathcal{S}}\ e}\ {\scriptstyle e:EXP}$$

$$\mathcal{S}21\ \frac{}{\sim(e_1 \cup e_2)\ \to_{\mathcal{S}}\ \sim e_1 \cap \sim e_2}\ {\scriptstyle e_1,\ e_2:EXP} \qquad \mathcal{S}22\ \frac{}{\sim(e_1 \cap e_2)\ \to_{\mathcal{S}}\ \sim e_1 \cup \sim e_2}\ {\scriptstyle e_1,\ e_2:EXP}$$

$$\mathcal{S}23\ \frac{\mathsf{vars}\,p \subseteq \mathsf{vars}\,p';\ p \in PAT_{\text{EX}};\ p' \in PAT_{\text{EX}}}{\sim(e[p \mapsto p'])\ \to_{\mathcal{S}}\ (\sim e)[p \mapsto p']}\ {\scriptstyle e:EXP\ \ p,p':PAT}$$

$$\mathcal{S}24\ \frac{}{\sim\{p \mid e\}\ \to_{\mathcal{S}}\ \{p \mid \sim e\} \cup \sim(1[p \mapsto p])}\ {\scriptstyle p:PAT\ \ e:EXP}$$

Figure 4.8: Complement Normalization

Translation

Rules $\mathcal{S}25$ and $\mathcal{S}26$ in Figure 4.9 (on the facing page) distribute translation over set union and intersection. For intersection, only embedding translations can be distributed; hidings are in fact lifted as described earlier (Rule $\mathcal{S}19$). Rule $\mathcal{S}27$ reduces the translation of the empty set. Rule $\mathcal{S}28$ eliminates a translation by the same exhaustive pattern. (An instance where this rule applies is a translation by the same variable, $x \mapsto x$.)

Rule $\mathcal{S}29$ handles consecutive translations that are contradictory, because the target pattern of the first translation cannot be unified with the source pattern of the second.

Rule $\mathcal{S}30$ tackles consecutive translation that can be contracted. It is *only* applied if the resulting translation is a hiding or an embedding, in order to prevent a circular application w.r.t. Rule $\mathcal{S}31$, that splits a translation which is not a hiding or embedding into a consecutive embedding and hiding.

Rule $\mathcal{S}32$ reduces a translation of a schema whose pattern does not unify with the translation's source pattern. Rule $\mathcal{S}33$ eliminates an embedding translation of a schema by substituting in the schema's property the assignment resulting from unification of the schema's pattern with the translation's source pattern.

Schema

Rules $\mathcal{S}34$ and $\mathcal{S}35$ in Figure 4.10 (on the next page) handle distribution of schema over set union and intersection. Rules $\mathcal{S}36$ and $\mathcal{S}37$ eliminate schemas with tautological properties.

Rule $\mathcal{S}38$ describes the conversion of a schema property, $?_1(\bigcap \bar{l})$, into a pattern membership-test, $p \in c$ (as discussed above on page 75). This rule is sketched only informally. We assume that $?_1(\bigcap \bar{l})$ is normalized as far as possible, and that constructor decomposition is not applied after this rule (otherwise we run into a cycle, since a candidate for constructor decomposition is introduced by the rule). Moreover, the fact that a renaming of double variables in $\mathsf{subs}\ \gamma'\,p$ has to be performed, adding corresponding alias equations, is not formalized.

$$\mathcal{S}25\ \frac{}{(e_1 \cup e_2)[p \mapsto p'] \ \to_{\mathcal{S}}\ e_1[p \mapsto p'] \cup e_2[p \mapsto p']}\ \ {\scriptstyle e_1,e_2:EXP \atop p,p':EXP}$$

$$\mathcal{S}26\ \frac{\mathsf{vars}\, p \subseteq \mathsf{vars}\, p'}{(e_1 \cap e_2)[p \mapsto p'] \ \to_{\mathcal{S}}\ e_1[p \mapsto p'] \cap e_2[p \mapsto p']}\ \ {\scriptstyle e_1,e_2:EXP \atop p,p':EXP}$$

$$\mathcal{S}27\ \frac{}{\mathsf{0}[p_1 \mapsto p_2] \ \to_{\mathcal{S}}\ \mathsf{0}}\ \ {\scriptstyle p_1, \atop p_2:PAT} \qquad \mathcal{S}28\ \frac{p \in PAT_{\mathrm{EX}}}{e[p \mapsto p] \ \to_{\mathcal{S}}\ e}\ \ {\scriptstyle e:EXP \atop p:PAT}$$

$$\mathcal{S}29\ \frac{\neg\,(\exists\,\gamma\ :\ EXPASS \bullet\ p_2 =^{\gamma}\ p_3)}{e[p_1 \mapsto p_2][p_3 \mapsto p_4] \ \to_{\mathcal{S}}\ \mathsf{0}}\ \ {\scriptstyle e:EXP \atop p_1,p_2,p_3,\ p_4:PAT}$$

$$\mathcal{S}30\ \frac{p_2 =^{\gamma}\ p_3 \quad \mathsf{vars}\,(\mathsf{subs}\ \gamma\, p_1) \subseteq \mathsf{vars}\,(\mathsf{subs}\ \gamma\, p_4) \vee \mathsf{vars}\,(\mathsf{subs}\ \gamma\, p_4) \subseteq \mathsf{vars}\,(\mathsf{subs}\ \gamma\, p_1)}{e[p_1 \mapsto p_2][p_3 \mapsto p_4] \ \to_{\mathcal{S}}\ e[\mathsf{subs}\ \gamma\, p_1 \mapsto \mathsf{subs}\ \gamma\, p_4]}\ \ {\scriptstyle e:EXP;\ p_1,p_2,\ p_3,p_4:PAT;\ \gamma:EXPASS}$$

$$\mathcal{S}31\ \frac{\neg\ \mathsf{vars}\, p_1 \subseteq \mathsf{vars}\, p_2;\ \neg\ \mathsf{vars}\, p_2 \subseteq \mathsf{vars}\, p_1 \quad p' = (\,(_) \circ \mathsf{varorder}\,(\mathsf{vars}\, p_1 \cup \mathsf{vars}\, p_2)\,)}{e[p_1 \mapsto p_2] \ \to_{\mathcal{S}}\ e[p_1 \mapsto p'][p' \mapsto p_2]}\ \ {\scriptstyle e:EXP;\ p_1,p_2,\ p':PAT}$$

$$\mathcal{S}32\ \frac{\neg\,(\exists\,\gamma\ :\ EXPASS \bullet\ p_1 =^{\gamma}\ p_2)}{\{p_1 \mid e\}[p_2 \mapsto p_3] \ \to_{\mathcal{S}}\ \mathsf{0}}\ \ {\scriptstyle e:EXP;\ p_1,p_2,\ p_3:PAT}$$

$$\mathcal{S}33\ \frac{p_1 =^{\gamma}\ p_2;\ \mathsf{vars}\, p_2 \subseteq \mathsf{vars}\, p_3}{\{p_1 \mid e\}[p_2 \mapsto p_3] \ \to_{\mathcal{S}}\ \{\mathsf{subs}\ \gamma\ p_3 \mid \mathsf{subs}\ \gamma\ e\}}\ \ {\scriptstyle e:EXP;\ p_1,p_2,\ p_3:PAT;\ \gamma:EXPASS}$$

Figure 4.9: Translation Normalization

$$\mathcal{S}34\ \frac{}{\{p \mid e_1 \cup e_2\} \ \to_{\mathcal{S}}\ \{p \mid e_1\} \cup \{p \mid e_2\}}\ \ {\scriptstyle p:PAT \atop e_1,e_2:EXP} \qquad \mathcal{S}35\ \frac{}{\{p \mid e_1 \cap e_2\} \ \to_{\mathcal{S}}\ \{p \mid e_1\} \cap \{p \mid e_2\}}\ \ {\scriptstyle p:PAT \atop e_1,e_2:EXP}$$

$$\mathcal{S}36\ \frac{}{\{p \mid \mathsf{0}\} \ \to_{\mathcal{S}}\ \mathsf{0}}\ \ {\scriptstyle p:PAT} \qquad \mathcal{S}37\ \frac{}{\{p \mid \mathsf{1}\} \ \to_{\mathcal{S}}\ \mathsf{1}[p \mapsto p]}\ \ {\scriptstyle p:PAT}$$

$$\mathcal{S}38\ \frac{x_i \in \mathsf{vars}\, p;\ q_i =^{\gamma}\ q_j \quad \gamma' = \{x_1 \mapsto \mathsf{subs}\ \gamma\ p_1, \ldots, x_n \mapsto \mathsf{subs}\ \gamma\ p_n\} \quad \bar{x} = \mathsf{varorder}\,(\mathsf{vars}\,(\mathsf{subs}\ \gamma\ q_1) \cap \bigcup(\mathsf{vars}\,(\!|\mathsf{ran}\ \gamma'|\!)))}{\{p \mid ?_1(\dot{\bigcap}\langle\{x_1\}[p_1 \mapsto q_1], \ldots, \{x_n\}[p_n \mapsto q_n]\rangle)\} \ \to_{\mathcal{S}}\ \{\mathsf{subs}\ \gamma'\, p \mid (\,\bar{x}\,) \in (\mathsf{subs}\ \gamma'(\dot{\bigcap}\langle\rangle))[\mathsf{subs}\ \gamma\ q_1 \mapsto (\,\bar{x}\,)]\}}\ \ {\scriptstyle p,\ p_i,q_i:PAT;\ \dot{\bigcap}:INCTX;\ \gamma,\gamma':EXPASS;\ \bar{x}:\ \mathrm{seq}\ VAR}$$

Figure 4.10: Schema Normalization

4.2.3 Properties of Normalization

The relation $_ \to_{\mathcal{S}} _$ has some properties that will be investigated below. Let $_ \rightsquigarrow_{\mathcal{S}} _$ be the lifting of $_ \to_{\mathcal{S}} _$ to a context rewriting, such that $e_1 \rightsquigarrow_{\mathcal{S}} e_2$ iff there exists a context E and subexpressions e_1', e_2' such that $e_1 = \mathrm{E}\ e_1'$, $e_1' \to_{\mathcal{S}} e_2'$, and $e_2 = \mathrm{E}\ e_2'$.

PROPOSITION 4.1 The relation $_ \rightsquigarrow_{\mathcal{S}} _$ is sound w.r.t. the meaning of μZ. $\Diamond$

Proof (informal). The rules are directly derived from the equational laws. $\square$

PROPOSITION 4.2 The relation $_ \rightsquigarrow_{\mathcal{S}} _$ is terminating. $\Diamond$

Proof (informal). The rules of $_ \rightarrow_{\mathcal{S}} _$ can be characterized in terms of three classes: elimination rules which decrease the size of their operands; distributivity rules which push a μZ operator down into an expression, thus possibly duplicating some operands; and lifting rules which do the opposite of distribution. The elimination rules are obviously safe as regards termination. Distributivity and lifting rules are safe if they exclude each other: what has been distributed must not be lifted to its original form in a later rewriting step. There are two problematic cases here:

- Rule $\mathcal{S}30$ composes and rule $\mathcal{S}31$ splits translations. But the application conditions have been chosen such that composition is only performed if splitting cannot be applied to the result.
- Rule $\mathcal{S}3$ decomposes patterns and rule $\mathcal{S}38$ synthesizes them. Preventing a cycle in the application of these rules is not been formalized (because Rule $\mathcal{S}38$ is left informal) but it is easily possible: since the effect of $\mathcal{S}38$ is localized to a schema's property, normalization can be sequentialized. First the rule system is comprehensively applied without using Rule $\mathcal{S}38$, then in a second pass $\mathcal{S}38$ is comprehensively applied.

$\square$

PROPOSITION 4.3 $_ \rightsquigarrow_{\mathcal{S}} _$ is confluent w.r.t. extensional equality. $\Diamond$

Proof (informal). Since we can assume soundness, the problem reduces to showing that for each derivation that leads to an extensional value (one which is in the domain of $_ \sim _$), each alternative derivation also leads to an extensional value. The sources for increasing extensionality in a derivation are elimination rules. The question is whether the possibility of applying some of these rules is obstructed if we prefer applying another rule before a possible elimination step:

- The Boolean elimination rules cause no problems here. A union or intersection context is either reduced to a tautology, which is the "best" result that can be achieved, or a duplicate operand is removed, still allowing elimination in its context with the remaining operand.
- The application of distributivity and lifting rules actually increases the potential for simplification, so there is no problem if one prefers distributing before a possible elimination step.

$\square$

On the basis of Proposition 4.2 and Proposition 4.3, we can assert the existence of a total function nrm which normalizes and simplifies an expression. For nonextensional

results of normalization, where confluence cannot be asserted, some arbitrary but fixed reduction order is assumed:

$$| \ \mathsf{nrm} : \mathit{EXP} \longrightarrow \mathit{EXP}$$

PROPOSITION 4.4 The function nrm puts an expression into disjunctive normal form, as specified by the grammar in Figure 4.3. ◊

Proof (informal). In general, establishing a disjunctive normal form is a standard process. Our rule system differs from this only in that translations need to be treated, and properties of schemas must be normalized. The decomposition of any nonhiding and nonembedding into an embedding followed by a hiding, and the strict distribution of embeddings and lifting of hidings, ensures that translations are normalized as required. Establishing the normal form for properties was justified on Page 75. □

PROPOSITION 4.5 There exists an innermost-out reduction strategy, an *inductive algorithm over the expression structure*, which implements nrm. ◊

Proof (informal). The algorithm is the usual one for establishing a disjunctive normal form. For example, for distributivity of intersection over union, we can give an inductive rule in the style of:

$$\frac{\dot{\bigcap}\langle e_1\rangle \rightsquigarrow^*_{\mathcal{S}} e'_1;\ \dot{\bigcap}\langle e_2\rangle \rightsquigarrow^*_{\mathcal{S}} e'_2;\ e'_1 \cup e'_2 \rightarrow^*_{\mathcal{S}} e'}{\dot{\bigcap}\langle e_1 \cup e_1\rangle \rightsquigarrow^*_{\mathcal{S}} e'}$$

In general, distributivity and lifting rules drive the algorithm. Elimination rules are applied at each expression depth (here depicted by $e'_1 \cup e'_2 \ \rightarrow_{\mathcal{S}} \ e'$) before the final result expression is returned. □

$$\mathcal{N}1\ \frac{}{\mathrm{E}\,(\dot{\bigcap}\,\langle\{p \mid \phi\},\{b\}\rangle) \leadsto_{\mathcal{N}} \mathsf{nrm}\,(\mathrm{E}\,(\dot{\bigcap}\,\langle\{p \mid \mathsf{subs}\ (\mathsf{mksubs}\,(p,b))\,\phi\},\{b\}\rangle))} \quad \begin{array}{l}\mathrm{E}:EXPCTX;\ \dot{\bigcap}:INCTX\\ b:EXP_B;\ \phi:EXP_P\\ p:PAT\end{array}$$

$$\mathcal{N}2\ \frac{p =^{\gamma} p_2;}{\begin{array}{l}\mathrm{E}\,(\dot{\bigcap}\,\langle\{p \mid \phi\},\{b\}[p_1 \mapsto p_2]\rangle) \leadsto_{\mathcal{N}}\\ \mathsf{nrm}\,(\mathrm{E}\,(\dot{\bigcap}\,\langle\{\mathsf{subs}\ \gamma\ p \mid \mathsf{subs}\ (\mathsf{mksubs}\,(\mathsf{subs}\ \gamma\ p_1, b))\,\phi\},\\ \{b\}[p_1 \mapsto p_2]\rangle))\end{array}} \quad \begin{array}{l}\mathrm{E}:EXPCTX;\ \dot{\bigcap}:INCTX\\ b:EXP_B;\ \phi:EXP_P\\ p,p_1,p_2:PAT\\ \gamma:EXPASS\end{array}$$

$$\mathcal{N}3\ \frac{\mathsf{vars}\ c \cap \mathsf{vars}\ p = \varnothing}{\mathrm{E}\,(\{p \mid p' \in c\}) \leadsto_{\mathcal{N}} \mathsf{nrm}\,(\mathrm{E}\,(c[p' \mapsto p]))} \quad \begin{array}{l}\mathrm{E}:EXPCTX\\ p,p':PAT\\ c:EXP_C\end{array}$$

$$\mathcal{N}4\ \frac{\textit{assert continuity of } d}{\mathrm{E}\,(\mathbf{fix}\,p \lhd d) \leadsto_{\mathcal{N}} \mathsf{nrm}\,(\mathrm{E}\,(\mathsf{subs}\,(\mathsf{mksubs}\,(p,\mathbf{fix}\,p \lhd d))\,d))} \quad \begin{array}{l}\mathrm{E}:EXPCTX\\ p:PAT;\ d:EXP_D\end{array}$$

Figure 4.11: Narrowing Rules

4.3 Narrowing Semantics

In normalized expressions nearly all the equational laws of μZ are exploited. The laws not used are *property substitution*, *membership lifting* and *fixed-point unrolling*. Considering the normal form of expressions, these are the laws driving reduction: with property substitution, variables are bound; with membership lifting, schemas are eliminated; and with fixed-point unrolling, a circular expression is lifted one level upwards in its context. Applying each of these steps may provide further simplification and normalization potential.

We will thus define the *narrowing semantics* of μZ as consecutively applying substitution, membership lifting and unrolling in an expression context, and afterwards renormalizing the entire context. The strategy for selecting the next redex can thereby be altered. In the general case, where redex selection is arbitrary, we conjecture a completeness result w.r.t. μZ's equational theory. A possible restriction is an outermost-in strategy. A feasible restriction is "strict narrowing", which is not complete, but more adequate for μZ than outermost-in, as will be discussed.

4.3.1 General Narrowing

The relation $_ \leadsto_{\mathcal{N}} _$ represents a narrowing step:

$$|\ _ \leadsto_{\mathcal{N}} _ : EXP_B \leftrightarrow EXP_B$$

The rules for $_ \leadsto_{\mathcal{N}} _$, using an arbitrary expression context E, are given in Figure 4.11. Rule $\mathcal{N}1$ implements property substitution for the case of a plain intersected singleton set; rule $\mathcal{N}2$ does so for a translated singleton set. These rules are direct applications of the law found in Figure 3.35 (on page 66). Rule $\mathcal{N}3$ realizes membership lifting (cf. Figure 3.34 (on page 65)). Rule $\mathcal{N}4$ implements fixed-point unrolling (cf. Section 3.4.7 (on page 66)), where continuity of the expression d needs to be asserted.

We investigate some relevant properties of $_ \leadsto_{\mathcal{N}} _$: soundness, extensional confluence, and extensional completeness.

PROPOSITION 4.6 The relation $_ \leadsto_{\mathcal{N}} _$ is sound w.r.t. the meaning of μZ, modulo the assertion that in each appearance of the fixed-point operator, the fixed expression is continuous. ◊

Proof (informal). Normalization is sound. The additional rules of $_ \leadsto_{\mathcal{N}} _$ are directly derived from the equational laws of the μZ calculus. □

PROPOSITION 4.7 The relation $_ \leadsto_{\mathcal{N}} _$ is confluent w.r.t. extensional equality. ◊

Proof (informal). We can assume extensional confluence and termination of normalization as well as soundness. As with the argument for confluence in Proposition 4.3 (on page 80), the question is whether indeterminism in the reduction order obstructs the potential for obtaining an extensional value in a derivation. This is excluded because each individual rule in fact increases potential for simplification, regardless of the order in which they are applied: rules $\mathcal{N}1$ and $\mathcal{N}2$ instantiate variables, and Rules $\mathcal{N}3$ and $\mathcal{N}4$ lift expressions out of their isolated context of a schema property into an intersection context, where further interactions are enabled. □

CONJECTURE 4.8 The relation $_ \leadsto_{\mathcal{N}} _$ is *extensionally complete* w.r.t. the equational theory of μZ: whenever $e \equiv e'$ and e' is extensional, then there exists an $e'' \sim e'$, such that $\mathsf{nrm}\, e \leadsto_{\mathcal{N}} e''$. ◊

A formal proof of this conjecture is omitted. In fact, the proof has been indirectly provided by *construction* of normalization and narrowing, providing a strategy for the comprehensive, directed application of μZ's equational laws:

- The normalization of translations – hidings are pulled, embeddings pushed – enables maximal elimination of them.
- All possibilities for applying property substitution are detected.
- The normalization to DNF, together with constructor decomposition, enables all simplifications in intersections.
- The normalization of schema properties recovers all possibilities for applying the membership lifting law.
- Complements, which hinder the application of property substitution and other laws, are pushed as far as possible. The "excluded middle" for singletons is not obstructed by this. Consider
$$\{e\} \cap \sim \{x \mid x \in \{e\}\}.$$
Using membership lifting *before* pushing complement yields $\{e\} \cap \sim \{e\}$, enabling the application of excluded middle. On the other hand, pushing the complement over the schema yields in
$$\{e\} \cap \{x \mid \sim (x \in \{e\})\}.$$

However, now property substitution can be applied, resulting in

$$\{e\} \cap \{x \mid \sim (e \in \{e\})\},$$

which likewise reduces to $\mathsf{0}$ after some steps.

General narrowing is so powerful because the reduction order is undetermined. Since all orders of reduction are possible , the "right" one (which terminates if termination is possible at all) is also contained in $_ \rightsquigarrow_{\mathcal{N}} _$. However, this also means that there are many derivations leading to a non-terminating "dead-end", and thus general narrowing is not very feasible for implementing computation.

4.3.2 Outermost-in Narrowing

The narrowing semantics defined in the previous section can be modified by supplying an outermost-in strategy for selecting the next redex. In order to model "parallel-or" and "parallel-and", indeterminism still needs to be present in the reduction order, but it can be significantly pruned.

The selection of an outermost-in redex is described by a context, declared as $\mathcal{R} \in$ *OUTEREDEX* $=$ *EXP* $\rightarrowtail\!\!\!\!\twoheadrightarrow$ *EXP* and specified by the following "grammar with a hole":

$$\mathcal{R}\cdot ::= \dot{\bigcup}\langle \mathcal{R}\cdot\rangle \mid \dot{\bigcap}\langle \mathcal{R}\cdot\rangle \mid \sim(\mathcal{R}\cdot) \mid (\mathcal{R}\cdot)[p_1 \mapsto p_2] \mid \mu\,(\mathcal{R}\cdot) \mid \cdot$$

The redex selection is indeterministic in the choice of the member of an intersection or a union, which models the "concurrency" of reduction by interleaving. Note that we do not select subexpressions in constructor terms, schema, and singleton set: this reflects the non-strictness of this order.

By replacing the expression context E in the narrowing rules $\mathcal{N}1$ to $\mathcal{N}4$ by a context $\mathcal{R}$, we get an *outermost-in* narrowing strategy. We conjecture that this strategy is "shallow complete": for *forced* expressions it obtains the same results as general narrowing. A forced expression is one captured by the context $\mathcal{R}$ during a derivation sequence. For example, if we have initially

$$\{(x, y) \mid x \in y\} \cap \{c\}[y \mapsto (_, y)]$$

then c will be protected by the singleton set, but after substitution we get

$$\{(x, y) \mid x \in c\} \cap \{c\}[y \mapsto (_, y)]$$

which is then transformed by membership lifting to

$$c[x \mapsto (x, _)] \cap \{c\}[y \mapsto (_, y)]$$

such that c is now captured by $\mathcal{R}$. In an actual implementation of this strategy, one would use sharing techniques such that the reduction of the bubbled c in the translation would also rewrite the original c in the singleton set – yielding a "lazy" reduction strategy.

The problem with this strategy is that, by contrast to functional logic languages which base on lazy narrowing [Hanus, 1994], in μZ equality and unequality is required for simplification rules concerned with *sets of sets* – a further possible source apart from the

redex selection for forcing reduction. For example, $\{\{e_1\}\} \cap \{\{e_2\}\}$ reduces (by normalization Rule $\mathcal{S}14$) to $\mathsf{0}$, if e_1 and e_2 reduce to an extensionally unequal value. Should we force the reduction of elements of singleton sets in an intersection, such that extensional equality or unequality *might* be applicable? The problem is thus to transparently define when values are forced. For this and other reasons, we prefer another strategy, called "strict narrowing".

4.3.3 Strict Narrowing

The basic decision in strict narrowing is to extend the selection of a redex to the elements of singleton sets, such that we get the following refined redex definition compared with outermost-in narrowing:

$$\mathcal{S}\cdot ::= \dot{\bigcup}\langle \mathcal{S}\cdot\rangle \;\mid\; \dot{\bigcap}\langle \mathcal{S}\cdot\rangle \;\mid\; \sim(\mathcal{S}\cdot) \;\mid\; (\mathcal{S}\cdot)[p_1 \mapsto p_2] \;\mid\; \mu\,(\mathcal{S}\cdot) \;\mid\; \{\mathcal{S}\cdot\} \;\mid\; \cdot$$

By replacing the general expression context E with $\mathcal{S}$ in the narrowing rules, and by further demanding that the *property substitution* rules, $\mathcal{N}1$ and $\mathcal{N}2$, are only performed if the element of the singleton set cannot be further reduced by $_ \rightsquigarrow_{\mathcal{N}} _$, we obtain a strict narrowing strategy. This strategy is still indeterministic because of the undetermined choice of which operand in union and intersection is to be selected next. It is not completely identical to what is usually described as "innermost-out" narrowing [Hanus, 1994], since we still do not look inside of schemas. Rather, it is related to applicative order reduction and weak head-normal form in the λ-calculus [Barendregt, 1984].

Since, by constructor decomposition, expressions of the kind $\{\rho(b)\}$ are normalized to $\{b\}[x \mapsto \rho(x)]$, it is useful for reasons of transparency to also make constructor application strict. We therefore add to the above definition of $\mathcal{S}$ contexts that select operands of constructor application (we do not formalize this).

Strict narrowing will be our operational semantics of choice, and the computation model developed in the next section will be based on this strategy. Its restriction is that diverging reduction of elements of singleton sets which would be not "forced" in lazy narrowing propagates to the context. However, if in this context other reductions determine the result, as in $\{\mathit{diverges}\} \cap \mathit{reducesToEmpty}$, then the nontermination will be caught by the concurrent reduction model.

4.4 The Reference Computation Model

In this section, the *reference computation model* of μZ, called RCM, is described, using the style of natural semantics [Kahn, 1987]. The intention is to obtain a model that is abstract enough to still support a conceptual idea of computation in μZ, but also near enough to an implementation to serve as a starting point for the design of an abstract machine for "off-line" computation. The strict narrowing semantics given in the previous section is not well suited to this purpose, since it is based on the symbolic manipulation of terms and requires renormalization of complete expression contexts after each narrowing step. In the model given here, substitution is to be replaced by environments and renormalization by techniques which can be mapped to backtracking and search.

In the previous section, we have already discussed why for conceptual reasons we prefer a strict semantics for computation. The model designed here has a few further restrictions, driven by the goals of efficient implementability and transparent *traceability* of execution. The last point is particularly important for the application to test-case evaluation, where the reason for execution failure needs to be traceable by humans. The following restrictions will be imposed in comparison with the strict narrowing semantics:

- In principle, the strict narrowing semantics provides "symbolic computation". An expression is irreducible if it cannot be further reduced by $_ \leadsto_{\mathcal{N}} _$, but such an irreducible expression may still be processed, for example, by substituting it in a property.

  In the RCM of μZ defined here, expressions need to be reducible to a value. If this value form is not reached, and the expression is needed to continue computation, then computation will stop. The restriction to value forms allows an efficient representation in an abstract machine, and makes the sources of failure traceable.

- The strict narrowing semantics given in the previous section allows arbitrary "parallel or" and "parallel and" reduction. In an implementation, this would mean that any subexpression of union and intersection has to be computed by concurrent threads, which causes significant efficiency problems.

  The RCM presented here makes a distinction between deterministic, "functional" reduction and indeterministic, "logic" resolution. The deterministic reduction is designed using the residuation technique [Ait-Kaci et al., 1987], which lets a computation suspend as soon as it encounters a free "logic" variable. The logic resolution uses parallel execution techniques.

  Reduction takes place for evaluating set expressions and plain expressions. Resolution is done when a reduced set value v is tested for membership, $p \in v$. "Parallel or" is not provided – disjunctions are executed from left to right, thus enabling backtracking techniques for their implementation.

For defining the RCM, we first specify *values*, a subset of expressions. Though values are still represented symbolically, we will later see in Chapter 5 that they correspond to an efficient representation in an implementation. We then define a set of inference rules which describe computation.

$$v \in EXP_V \quad ::= \quad \rho(\overline{v}) \mid \{v\} \mid [\![p \mid \theta]\!] \mid \bigcup \overline{v}$$

$$\theta \in CTR \quad ::= \quad \sigma \parallel \phi \mid \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2$$

$$\sigma \in VALASS \quad == \quad VAR \mathrel{+\!\!\!\rightarrow} EXP_V$$

$$\bigwedge, \bigvee : \mathrm{seq}\, CTR \longrightarrow CTR$$
$$_ \parallel _ : \mathrm{seq}\, VALASS \times \mathrm{seq}\, EXP_P \longrightarrow \mathrm{seq}\, CTR$$
$$\mathsf{vars} : CTR \longrightarrow \mathbb{F}\, VAR$$

$$_ \rhd _ : VALASS \times CTR \mathrel{+\!\!\!\rightarrow} CTR$$

$$\sigma \rhd (\theta_1 \vee \theta_2) = (\sigma \rhd \theta_1) \vee (\sigma \rhd \theta_2)$$
$$\sigma \rhd (\theta_1 \wedge \theta_2) = (\sigma \rhd \theta_1) \wedge (\sigma \rhd \theta_2)$$
$$\sigma \rhd (\sigma' \parallel \phi) = (\sigma \cup \sigma') \parallel \phi$$
$$\textbf{where } \sigma, \sigma' : VALASS;\ \theta_1, \theta_2 : CTR;\ \phi : EXP_P$$

$$\mathsf{ass} : CTR \mathrel{+\!\!\!\rightarrow} VALASS$$

$$\mathsf{ass}\,(\theta_1 \vee \theta_2) = \mathsf{ass}\,\theta_1 \cup \mathsf{ass}\,\theta_2$$
$$\mathsf{ass}\,(\theta_1 \wedge \theta_2) = \mathsf{ass}\,\theta_1 \cup \mathsf{ass}\,\theta_2$$
$$\mathsf{ass}\,\sigma \parallel \phi = \sigma$$
$$\textbf{where } \sigma : VALASS;\ \theta_1, \theta_2 : CTR;\ \phi : EXP_P$$

$$UVCTX, IVCTX : \mathbb{P}(\mathrm{seq}\, EXP_V \longrightarrow EXP)$$

$$_ \sim^{_} _, _ \not\sim_{_} _ : \mathbb{P}(EXP \times VALASS \times EXP)$$
$$_ \sim^{_}_{_} _, _ \not\sim^{_}_{_} _ : \mathbb{P}(EXP \times VALASS \times VALASS \times EXP)$$

Figure 4.12: Values and Constraints

4.4.1 Values and Constraints

Values are a subset of normalized expressions, enriched by a tailored representation of a schema-like construct, called an *intension*. An intension consists of a pattern and a *constraint*. A basic constraint, $\sigma \parallel \phi$, is a value assignment paired with a property expression. Constraints may be composed by conjunction and by disjunction, though we expect only conjunctions in intensions (disjunctions are needed later on). The definitions, together with some auxiliary functions on constraints and values, are given in Figure 4.12.

An intension $[\![p \mid \sigma \parallel \phi]\!]$ can be interpreted as the schema $\{p \mid \mathsf{subs}\ \sigma\ \phi\}$. However, there are a few differences. First, it is possible that the above expansion results in an infinite term, since σ may contain an assignment referring to the intension itself. This circularity of assignments is used to express fixed-points in the computation model. Second, the property ϕ may contain free variables which are bound neither by p nor by σ. These variables may be considered "locally existential" for the intension, and are used to represent hiding translations by intensions. When an intension is instantiated in a context, then these variables need to be renamed away from the variables of the context.

Some auxiliary functions for working with constraints are introduced. We write $\bigwedge \overline{\theta}$ for a conjunction of constraints, and $\bigvee \overline{\theta}$ for a disjunction. For a sequence of basic constraints, $\overline{\sigma} \parallel \overline{\theta}$ is written. The set of free variables used in a constraint can be retrieved by $\mathsf{vars}\,\theta$.

$\sigma \rhd \theta$ describes the propagation of an assignment into a constraint, and distributes over

conjunction and disjunction. For a basic constraint, we have $\sigma \rhd (\sigma \parallel \phi) = (\sigma \cup \sigma') \parallel \phi$. Here, propagation is only defined if $\sigma \cup \sigma'$ yields a proper function. This is only the case if, in their common domain, both assignments map to the same value (or the common domain is empty).

Given a constraint, its overall assignment is selected with the function $\mathsf{ass}\,\theta$. The assignment $\mathsf{ass}\,\theta$ is only defined if the union of the assignments of the subconstraints yields a proper function.

Some auxiliary functions for working with values are required. Value union and intersection contexts, $\dot{\bigcup} \in \mathit{UVCTX}$ and $\dot{\bigcap} \in \mathit{IVCTX}$, are a special case of general union and intersection contexts, *UNCTX* and *INCTX*, but expect the sequence elements to be values.

In Section 4.1 (on page 72) we defined a (partial) *extensional equality* and *extensional unequality*. These definitions carry over to values. As schemas can not be decided as equal or unequal, intensions cannot. On the basis of value equality and unequality, a unification for "extended patterns", built from constructor application, variables and values, and written as $p_1 \sim^{\sigma} p_2$, is assumed. This unification uses $_ \sim _$ for comparing values found at the bottom of constructor terms. The form $p_1 \not\sim_{\sigma} p_2$ decides whether unification fails (using $_ \not\sim _$). The notation $p_1 \sim_{\sigma}^{\sigma'} p_2$ "refines" the assignment σ to the assignment σ' by the result of unifying $\mathsf{subs}\ \sigma\ p_1$ with $\mathsf{subs}\ \sigma\ p_2$: similarly $p_1 \not\sim_{\sigma}^{\sigma'} p_2$ for the failure case.

In principle, more powerful notions of unification can be embedded in our computation model (e.g. set unification). This would require some technical modifications to the rules given in the next section to deal with *sets* of unifiers returned by unification instead of a single unifier.

4.4.2 Computation

Computation is defined by two relations, which are mutually dependent. The first relation, $e \xrightarrow{\sigma}_{\mathcal{E}} e'$, describes a strict reduction step on expressions under the value assignment σ. It cannot be applied to unbound variables, which means, in the context of its usage, that the corresponding computation "suspends". The relation $e \xrightarrow{\sigma}_{\mathcal{E}} e'$ has indeterminism, but it is "don't care": an implementation can choose some arbitrary fixed order to implement reduction.

The second relation, $\theta \rightsquigarrow_{\mathcal{C}} \theta'$, describes a resolution step by mapping a constraint onto a constraint. θ is expected to be in disjunctive normal form, and θ' will be, too. The indeterminism in this relation (provided it does not result from recursive application of $e \xrightarrow{\sigma}_{\mathcal{E}} e'$) is "don't know". Thus, an implementation has to provide a concurrent (interleaving) application strategy.

$$
\begin{array}{|l}
_ \xrightarrow{_}_{\mathcal{E}} _ : \mathbb{P}(\mathit{EXP} \times \mathit{VALASS} \times \mathit{EXP}) \\
_ \rightsquigarrow_{\mathcal{C}} _ : \mathit{CTR} \leftrightarrow \mathit{CTR}
\end{array}
$$

We expect that, before computation starts, expressions will be normalized to disjunctive normal form, as specified by the function $\mathsf{nrm}\,e$ in Section 4.2. However, during computation, no renormalization is necessary. Thus normalization can be done at "compile time". In contrast to the narrowing semantics, where simplification is an intrinsic part of normalization, the strict computation model does not actually depend on initial

$$\mathcal{E}1\ \frac{e \xrightarrow{\sigma}_{\mathcal{E}} e'}{\mathcal{S}\ e \xrightarrow{\sigma}_{\mathcal{E}} \mathcal{S}\ e'} \quad \begin{array}{l} e,e'{:}EXP \\ \sigma{:}VALASS \\ \mathcal{S}{:}SREDEX \end{array}$$

Figure 4.13: Strict Context Reduction

$$\mathcal{E}2\ \frac{x \in \mathsf{dom}\,\sigma}{x \xrightarrow{\sigma}_{\mathcal{E}} \sigma\, x} \quad \begin{array}{l} x{:}VAR \\ \sigma{:}VALASS \end{array} \qquad \mathcal{E}3\ \frac{\mathsf{vars}\ e \subseteq \mathsf{vars}\ p \cup \mathsf{dom}\,\sigma}{\{p \mid e\} \xrightarrow{\sigma}_{\mathcal{E}} [\![p \mid \sigma \parallel e]\!]} \quad \begin{array}{l} p{:}PAT;\ e{:}EXP \\ \sigma{:}VALASS \end{array}$$

$$\mathcal{E}4\ \frac{p \sim^{\sigma'}_{\sigma} v;\ e \xrightarrow{\sigma'}_{\mathcal{E}} e_1 \xrightarrow{\sigma'}_{\mathcal{E}} \ldots e_n \xrightarrow{\sigma'}_{\mathcal{E}} v}{\mathbf{fix}\, p \lhd e \xrightarrow{\sigma}_{\mathcal{E}} v} \quad \begin{array}{l} p{:}PAT;\ e{:}EXP \\ \sigma,\sigma'{:}VALASS;\ v{:}EXP_V \end{array}$$

Figure 4.14: Variable, Schema, and Fixed-Point Reduction

simplifications. The essential conditions are that properties of schemas are normalized, as described in Section 4.2 (on page 74), and complement is pushed.

Expression Reduction Rules

Rule $\mathcal{E}1$ in Figure 4.13 describes an inductive reduction step in a strict expression context, $\mathcal{S} \in SREDEX \subseteq EXP \rightarrowtail\!\!\!\!\!\rightarrow EXP$. A strict context $\mathcal{S}$ is specified as with the strict narrowing semantics. As explained above, the freedom of choice in selecting a strict redex in expression reduction amounts to "don't care" indeterminism instead of "don't know".

The rules in Figure 4.14 model the step where an expression is converted into a value. Rule $\mathcal{E}2$ describes the substitution of a variable, which will only be possible if the variable is bound by σ. This is one source of "suspension". Rule $\mathcal{E}3$ reduces a schema to an intension. The rule is only applicable if free variables of the property are either bound by the pattern or by the substitution – *i.e.* variables from outer scopes are not allowed to be free. This is the other source of "suspension". Rule $\mathcal{E}4$ describes the construction of a fixed-point. The rule is cyclic in its premise, forestalling the resulting value v in an assignment σ' which is used to calculate this value (the usual technique used in natural semantics for constructing fixed-points). A finite number of reduction steps is required to map the expression e to a value v: the expression e may contain several schemas, and the value form is not reached until all these schemas have been converted to intensions.

The rules in Figure 4.15 describe the reduction of translation and complement. Rule $\mathcal{E}5$ converts a translation into an intension. It exploits the possibility of using existential variables in an intensions constraint: p_1 may contain variables not bounded by p_2. Rule $\mathcal{E}6$ describes the translation of a complement into an intension.

The rules in Figure 4.16 (on the next page) describe the evaluation of set operators

$$\mathcal{E}5\ \frac{}{v[p_1 \mapsto p_2] \xrightarrow{\sigma}_{\mathcal{E}} [\![p_2 \mid \varnothing \parallel p_1 \in v]\!]} \quad \begin{array}{l} v{:}EXP_V;\ p_1,p_2{:}PAT \\ \sigma{:}VALASS \end{array}$$

$$\mathcal{E}6\ \frac{x \in fresh}{\sim v \xrightarrow{\sigma}_{\mathcal{E}} [\![x \mid \varnothing \parallel ?_0(x \in v)]\!]} \quad \begin{array}{l} v{:}EXP_V;\ x{:}VAR \\ \sigma{:}VALASS \end{array}$$

Figure 4.15: Translation and Complement Reduction

$$\mathcal{E}7\ \frac{}{\dot{\bigcup}\,\langle 0\rangle \xrightarrow{\sigma}_{\mathcal{E}} \dot{\bigcup}\,\langle\rangle}\ \begin{array}{l}\sigma{:}VALASS\\ \dot{\bigcup}{:}UVCTX\end{array} \qquad \mathcal{E}8\ \frac{v_1 \sim v_2}{\dot{\bigcup}\,\langle v_1, v_2\rangle \xrightarrow{\sigma}_{\mathcal{E}} \dot{\bigcup}\,\langle v_1\rangle}\ \begin{array}{l}v_1,v_2{:}EXP_V\\ \sigma{:}VALASS\\ \dot{\bigcup}{:}UVCTX\end{array}$$

$$\mathcal{E}9\ \frac{}{\dot{\bigcap}\,\langle 0\rangle \xrightarrow{\sigma}_{\mathcal{E}} 0}\ \begin{array}{l}\sigma{:}VALASS\\ \dot{\bigcap}{:}UVCTX\end{array} \qquad \mathcal{E}10\ \frac{v_1 \not\sim v_2}{\dot{\bigcap}\,\langle v_1, v_2\rangle \xrightarrow{\sigma}_{\mathcal{E}} 0}\ \begin{array}{l}v_1,v_2{:}EXP_V\\ \sigma{:}VALASS\\ \dot{\bigcap}{:}UVCTX\end{array}$$

$$\mathcal{E}11\ \frac{v_1 \sim v_2}{\dot{\bigcap}\,\langle v_1, v_2\rangle \xrightarrow{\sigma}_{\mathcal{E}} \dot{\bigcap}\,\langle v_1\rangle}\ \begin{array}{l}v_1,v_2{:}EXP_V\\ \sigma{:}VALASS\\ \dot{\bigcap}{:}UVCTX\end{array}$$

$$\mathcal{E}12\ \frac{\begin{array}{c}p_1 \sim^{\sigma'} p_2\\ exvs_2 = \mathsf{vars}\,\theta_2 \setminus \mathsf{vars}\,p_2;\ \gamma \in exvs_2 \rightarrowtail fresh\end{array}}{\dot{\bigcap}\,\langle [\![p_1 \mid \theta_1]\!], [\![p_2 \mid \theta_2]\!]\rangle \xrightarrow{\sigma}_{\mathcal{E}} \dot{\bigcap}\,\langle [\![\mathsf{subs}\ \sigma'\,p_1 \mid \sigma' \rhd\ (\theta_1 \wedge (\gamma\ \rhd \theta_2))]\!]\rangle}\ \begin{array}{l}p_1,p_2{:}PAT\\ \theta_1,\theta_2{:}CTR\\ \sigma,\sigma'{:}VALASS\\ exvs_2{:}\mathbb{F}\ VAR\\ \gamma{:}VALASS\\ \dot{\bigcap}{:}IVCTX\end{array}$$

$$\mathcal{E}13\ \frac{p_1 \not\sim_{\sigma'} p_2}{\dot{\bigcap}\,\langle [\![p_1 \mid \theta_1]\!], [\![p_2 \mid \theta_2]\!]\rangle \xrightarrow{\sigma}_{\mathcal{E}} 0}\ \begin{array}{l}p_1,p_2{:}PAT\\ \theta_1,\theta_2{:}CTR\\ \sigma,\sigma'{:}VALASS\\ \dot{\bigcap}{:}IVCTX\end{array}$$

$$\mathcal{E}14\ \frac{}{\dot{\bigcap}\,\langle v_1 \cup v_2\rangle \xrightarrow{\sigma}_{\mathcal{E}} \bigcap\,\langle v_1\rangle \cup \bigcap\,\langle v_2\rangle}\ \begin{array}{l}v_1,v_2{:}EXP_V\\ \sigma,\sigma'{:}VALASS\\ \dot{\bigcap}{:}IVCTX\end{array}$$

Figure 4.16: Set Value Reduction

$$\mathcal{E}15\ \frac{x \in fresh}{\mu\ v \xrightarrow{\sigma}_{\mathcal{E}} \mu_{\mathcal{E}}\,(x, \varnothing \parallel x \in v)}\ \begin{array}{l}v{:}EXP_V\\ x{:}VAR\\ \sigma{:}VALASS\end{array} \qquad \mathcal{E}16\ \frac{\theta \leadsto_{\mathcal{C}} \theta'}{\mu_{\mathcal{E}}\,(x,\theta) \xrightarrow{\sigma}_{\mathcal{E}} \mu_{\mathcal{E}}\,(x,\theta')}\ \begin{array}{l}x{:}VAR\\ \sigma{:}VALASS\\ \theta,\theta'{:}CTR\end{array}$$

$$\mathcal{E}17\ \frac{\begin{array}{c}\#\,\overline{\sigma} > 0\\ \forall\, i : \mathsf{dom}\,\overline{\sigma} \bullet x \in \mathsf{dom}(\overline{\sigma}\ i);\ \forall\, i,j : \mathsf{dom}\,\overline{\sigma} \bullet \overline{\sigma}\ i\,x \sim \overline{\sigma}\ j\,x\end{array}}{\mu_{\mathcal{E}}\,(x, \bigvee\,(\overline{\sigma} \parallel \overline{1})) \xrightarrow{\sigma}_{\mathcal{E}} \overline{\sigma}\ 1\,x}\ \begin{array}{l}x{:}VAR\\ \sigma{:}VALASS\\ \overline{\sigma}{:}\mathsf{seq}\ VALASS\end{array}$$

Figure 4.17: Mu-Value Reduction

on set *values*. Rules $\mathcal{E}7$ to $\mathcal{E}8$ describe the elimination of tautologies in a union of set values; Rules $\mathcal{E}9$ to $\mathcal{E}11$ for the intersection of set values. Rules $\mathcal{E}12$ and $\mathcal{E}13$ model the combination of intensional values by intersection: if the patterns can be unified, then we simply concatenate the constraints of the intensions, making local existential variables disjoint by a renaming γ into fresh variables. If the patterns do not unify, the intersection reduces to 0. Finally, Rule $\mathcal{E}14$ describes the distribution of intersection over union values. It should be noted that, in an implementation, intersection can be realized more efficiently as suggested by the rules: given a total order on values and set values in a normalized form $\bigcup\,\{\overline{v}\} \cup \bigcup\,[\![\overline{p} \mid \overline{\theta}]\!]$, the intersection of two such values can be realized by processing the extensional part, $\overline{v}$, in linear time, and then the combinations with the intensional part.

The remaining rules in Figure 4.17 describe the handling of the μ-operator. Once the argument of μ has been reduced to a value v, the elements of v are enumerated by resolving a constraint $x \in v$. The state of this subresolution is stored in an intermediate expression construct, $\mu_{\mathcal{E}}\,(x,\theta)$. When the subresolution is in solved form, $\bigvee\,(\overline{\sigma} \parallel \overline{1})$, a

$$\mathcal{C}1\frac{\theta \rightsquigarrow_{\mathcal{C}} \bigvee \overline{\theta}'}{\bigvee (\overline{\sigma} \parallel \overline{1} \frown \langle\theta\rangle \frown \overline{\theta}) \rightsquigarrow_{\mathcal{C}} \bigvee (\overline{\sigma} \parallel \overline{1} \frown \overline{\theta}' \frown \overline{\theta})} \quad \begin{array}{l}\theta{:}CTR\\ \overline{\theta},\overline{\theta}'{:}\text{seq } CTR\\ \overline{\sigma}{:}\text{seq } VALASS\end{array}$$

$$\mathcal{C}2\frac{}{\bigvee (\langle\sigma \parallel 0\rangle \frown \overline{\theta}) \rightsquigarrow_{\mathcal{C}} \bigvee \overline{\theta}} \quad \begin{array}{l}\sigma{:}VALASS\\ \overline{\theta}{:}\text{seq } CTR\end{array}$$

$$\mathcal{C}3\frac{}{\bigwedge (\mathcal{S}\langle\sigma \parallel 0\rangle) \rightsquigarrow_{\mathcal{C}} \sigma \parallel 0} \quad \begin{array}{l}\sigma{:}VALASS\\ \mathcal{S}{:}\\ \text{SCTX } CTR\end{array} \qquad \mathcal{C}4\frac{}{\bigwedge (\mathcal{S}\langle\sigma \parallel 1\rangle) \rightsquigarrow_{\mathcal{C}} \bigwedge (\mathcal{S}\langle\rangle)} \quad \begin{array}{l}\sigma{:}VALASS\\ \mathcal{S}{:}\\ \text{SCTX } CTR\end{array}$$

$$\mathcal{C}5\frac{\theta \rightsquigarrow_{\mathcal{C}} \bigvee \overline{\theta};\ n = \#\overline{\theta}}{\begin{array}{c}\bigwedge (\mathcal{S}\langle\theta\rangle) \rightsquigarrow_{\mathcal{C}}\\ \bigvee \langle \mathsf{ass}\,(\overline{\theta}\,1) \rhd \bigwedge (\mathcal{S}\langle\overline{\theta}\,1\rangle), \ldots, \mathsf{ass}\,(\overline{\theta}\,n) \rhd \bigwedge (\mathcal{S}\langle\overline{\theta}\,n\rangle)\rangle\end{array}} \quad \begin{array}{l}\theta{:}CTR\\ \overline{\theta}{:}\text{seq } CTR\\ \mathcal{S}{:}\text{SCTX } CTR\\ n{:}\mathbb{N}\end{array}$$

Figure 4.18: Compound Constraint Resolution

$$\mathcal{C}6\frac{e \xrightarrow{\sigma}_{\mathcal{E}} e'}{\sigma \parallel p \in e \rightsquigarrow_{\mathcal{C}} \sigma \parallel p \in e'} \quad \begin{array}{l}\sigma{:}VALASS\\ e,e'{:}EXP\\ p{:}PAT\end{array} \qquad \mathcal{C}7\frac{e \xrightarrow{\sigma}_{\mathcal{E}} e'}{\sigma \parallel ?_s e \rightsquigarrow_{\mathcal{C}} \sigma \parallel ?_s e'} \quad \begin{array}{l}\sigma{:}VALASS\\ e,e'{:}EXP\\ s{:}\{1,0\}\end{array}$$

Figure 4.19: Expression Reduction in Constraints

unique assignment for x is extracted, if it exists.

Constraint Resolution Rules

Compound constraints are resolved by the rules in Figure 4.18. Rule $\mathcal{C}1$ describes the left-to-right resolution of disjunctive constraints, Rule $\mathcal{C}2$ the elimination of unsolvable constraints. Rules $\mathcal{C}3$ and $\mathcal{C}4$ describe the elimination of unsolvable and solved constraints in a conjunction. Rule $\mathcal{C}5$ describes a resolution step of a basic constraint in a conjunction, where $S\langle\theta\rangle$ is a sequence context. This rule models the concurrent execution of constraints by "interleaving". An arbitrary basic constraint is selected (remember that constraints are in DNF) and resolved one step, possibly yielding a disjunction. Maintaining the DNF by pushing the conjunctive context $\mathcal{S}$ over each member of the disjunction corresponds to backtracking in an implementation. The assignments of the conjuncts, $\overline{\theta}\,i$, are propagated over $\mathcal{S}$, such that a steady refinement of the assignments is guaranteed.

The rules in Figure 4.19 describe the reduction of expressions in basic constraints. These rules are only able to fire if all free variables in e are bounded by σ – otherwise $e \xrightarrow{\sigma}_{\mathcal{E}} e'$ is "suspended".

The rules in Figure 4.20 (on the next page) describe the resolution of pattern membership in readily reduced set values, and resolution of equality constraints. Rule $\mathcal{C}8$ decomposes the membership test on a union of values into a disjunctive constraint. This is the only way basic constraints can rewrite into disjunctions, thereby introducing the need for backtracking. Rules $\mathcal{C}9$ and $\mathcal{C}10$ describe a basic resolution step, reducing a membership test on a singleton set to an assignment or a failure. Rules $\mathcal{C}11$ and $\mathcal{C}12$ describe the resolution of a membership test on an intension. Variables used in the intension's pattern and constraint are renamed by γ to fresh ones. Rules $\mathcal{C}13$ and $\mathcal{C}14$ model the resolution of a pattern equality.

The remaining rules in Figure 4.21 (on the following page) describe the resolution of

$$\mathcal{C}8\;\frac{}{\sigma \parallel p \in (v_1 \cup v_2) \rightsquigarrow_{\mathcal{C}} (\sigma \parallel p \in v_1) \vee (\sigma \parallel p \in v_2)} \quad \begin{array}{l}\sigma{:}VALASS\\ p{:}PAT\\ v_1,v_2{:}EXP_V\end{array}$$

$$\mathcal{C}9\;\frac{p \sim^{\sigma'}_{\sigma} v}{\sigma \parallel p \in \{v\} \rightsquigarrow_{\mathcal{C}} \sigma' \parallel 1} \quad \begin{array}{l}\sigma,\\ \sigma'{:}VALASS\\ p{:}PAT\\ v{:}EXP_V\end{array} \qquad \mathcal{C}10\;\frac{p \not\sim^{\sigma'}_{\sigma} v}{\sigma \parallel p \in \{v\} \rightsquigarrow_{\mathcal{C}} \sigma \parallel 0} \quad \begin{array}{l}\sigma,\\ \sigma'{:}VALASS\\ p{:}PAT\\ v{:}EXP_V\end{array}$$

$$\mathcal{C}11\;\frac{\gamma \in \mathsf{vars}\,\theta \cup \mathsf{vars}\,p' \rightarrowtail fresh;\; p \sim^{\sigma'}_{\sigma} \mathsf{subs}\;\gamma\; p'}{\sigma \parallel p \in [\![p' \mid \theta]\!] \rightsquigarrow_{\mathcal{C}} \sigma' \rhd (\gamma \rhd \theta)} \quad \begin{array}{l}\sigma,\sigma'{:}VALASS\\ p,p'{:}PAT\\ \theta{:}CTR\\ \gamma{:}VALASS\end{array}$$

$$\mathcal{C}12\;\frac{\gamma \in \mathsf{vars}\,p' \rightarrowtail fresh;\; p \not\sim^{\sigma'}_{\sigma} \mathsf{subs}\;\gamma\; p'}{\sigma \parallel p \in [\![p' \mid \theta]\!] \rightsquigarrow_{\mathcal{C}} \sigma \parallel 0} \quad \begin{array}{l}\sigma,\sigma'{:}VALASS\\ p,p'{:}PAT\\ \theta{:}CTR\\ \gamma{:}VALASS\end{array}$$

$$\mathcal{C}13\;\frac{p_1 \sim^{\sigma'}_{\sigma} p_2}{\sigma \parallel p_1 = p_2 \rightsquigarrow_{\mathcal{C}} \sigma' \parallel 1} \quad \begin{array}{l}\sigma,\sigma'{:}VALASS\\ p_1,p_2{:}PAT\end{array} \qquad \mathcal{C}14\;\frac{p_1 \not\sim^{\sigma'}_{\sigma} p_2}{\sigma \parallel p_1 = p_2 \rightsquigarrow_{\mathcal{C}} \sigma \parallel 0} \quad \begin{array}{l}\sigma,\sigma'{:}VALASS\\ p_1,p_2{:}PAT\end{array}$$

Figure 4.20: Membership and Equality Resolution

$$\mathcal{C}15\;\frac{x \in fresh}{\begin{array}{c}\sigma \parallel ?_s v \rightsquigarrow_{\mathcal{C}}\\ \sigma \parallel ?_s(x, \varnothing \parallel x \in v)_{\mathcal{C}}\end{array}} \quad \begin{array}{l}\sigma{:}VALASS\\ v{:}EXP_V\\ x{:}VAR\\ s{:}\{0,1\}\end{array} \qquad \mathcal{C}16\;\frac{\theta \rightsquigarrow_{\mathcal{C}} \theta'}{\begin{array}{c}\sigma \parallel ?_s(x,\theta)_{\mathcal{C}} \rightsquigarrow_{\mathcal{C}}\\ \sigma \parallel ?_s(x,\theta')_{\mathcal{C}}\end{array}} \quad \begin{array}{l}\sigma{:}VALASS\\ x{:}VAR\\ \theta,\theta'{:}CTR\\ s{:}\{0,1\}\end{array}$$

$$\mathcal{C}17\;\frac{}{\sigma \parallel ?_s(x, \bigvee(\langle \sigma' \parallel 1\rangle \frown \overline{\theta}))_{\mathcal{C}} \rightsquigarrow_{\mathcal{C}} \sigma \parallel s} \quad \begin{array}{l}\sigma,\sigma'{:}VALASS\\ x{:}VAR\\ \overline{\theta}{:}\mathsf{seq}\,CTR\\ s{:}\{0,1\}\end{array}$$

$$\mathcal{C}18\;\frac{}{\sigma \parallel ?_s(x, \sigma' \parallel 0)_{\mathcal{C}} \rightsquigarrow_{\mathcal{C}} \sigma \parallel compl\, s} \quad \begin{array}{l}\sigma,\sigma'{:}VALASS\\ x{:}VAR\\ \overline{\theta}{:}\mathsf{seq}\,CTR\\ s{:}\{0,1\}\end{array}$$

Figure 4.21: Test-Emptiness Resolution

set tests, $?_s v$, where the tested set has been reduced to a value. Similar to the treatment of μ in expression reduction, a subresolution is started which enumerates the elements of the set v by resolving the constraint $x \in v$. The state of this resolution is stored in an intermediate property expression, $?_s(x,\theta)_{\mathcal{C}}$, where s is either 1 or 0, depending on the pole of the test. The test succeeds (or fails, depending on s) if at least one constraint can be solved; it fails (or succeeds) if no constraint can be solved (which requires enumerating the entire set).

4.5 Discussion

We have developed several models for computation in μZ. Further points and related work are discussed below.

Normalization and Simplification

Normalization has been defined as a prerequisite for any kind of computation, and will be also required for compilation in the next chapter. Applying normalization ensures that negative information is distributed as far as possible to the leaves of expressions, enabling resolution techniques in the lifted positive contexts. Local resolution can be used in the remaining (minimized) negative contexts. Universal quantification, which is represented in μZ as the combination of complement and hiding, is, for example, solved by local resolution.

A well-known problem of normalization toward a disjunctive normal form is the explosion of the expression size. This is one reason why we interleaved normalization with simplification: pruning this explosion early. As a rule of thumb, "sensitive" μZ expressions do not explode if simplification is interleaved with DNF production. However, pathological cases might still cause serious problems. Temporary variables may be used to avoid the explosion of the DNF. For example, the expression $e \cap (e_1 \cup e_2)$ can be normalized to the form:

$$(\{(x,t) \mid x = e \cap t\} \cap \{e_1 \cup e_2\}[t \mapsto (_,t)])[(x,_) \mapsto x]$$

With this technique, expression sharing can also be preserved, and new sharing potentials can be exploited using common subexpression elimination. However, the technique also has its disadvantages: the number of constraints increases and the introduction of new temporary variables implies administrative overhead for binding and accessing these variables.

An alternative approach for avoiding explosion of expression size on normalization is to use *directed acylcic graphs* (DAGs) with *optimal sharing* for representing expressions. Structurally equal expressions do not appear twice in such a DAG. It is known e.g. from symbolic model checking [McMillan, 1993] that, for boolean algebras, quite efficient algorithms for constructing and maintaining this representation exist. However, further experiments are required, particularly regarding the treatment of the non-Boolean operators of μZ, like translation, μ -selection, etc.

Narrowing

Narrowing can be elegantly expressed in μZ's framework by pure expression transformation, combining normalization and simplification with substitution, membership lifting and unrolling. This is a result of the "higher-orderness" of the μZ calculus, which allows us to represent "meta" information, like the set of (partial) solutions, by expressions of the calculus itself.

In rule based functional logic languages, solutions are represented as substitutions on the meta level. Here, narrowing is defined as follows (see Hanus [1994] for a survey): given a (nonground) goal expression, a subexpression is selected which unifies

with the left-hand side of some rule under a minimal substitution, and is replaced by its right-hand side instantiated with the substitution. The substitution is then applied to the entire expression. At a first sight, this looks different from what we call narrowing in μZ. However, the same elements are, in fact, present as a special case in μZ. The "rules" belonging to a function or relation symbol are simply represented as a union of schemas, $R = \{p_1 \mid e_1\} \cup \ldots \cup \{p_n \mid e_n\}$. A "goal" is then merely the intersection of R with some (partial) data binding, represented by a (translated) singleton set: $R \cap \{e\}[p_1 \mapsto p_2]$. Rule "selection" is represented by distributing the singleton set over the union R and performing property substitution. Each member of the resulting union is part of the explicitly represented search space, which, in narrowing for rule-based languages, is coded in the nondeterminism of the narrowing relation.

As e.g. demonstrated by Hanus [1994], modifying the selection strategy for the next redex in narrowing is a powerful tool for investigating variations of computation. This technique has been applied to our version of narrowing in μZ as well, defining general narrowing, outermost-in narrowing and strict narrowing. Strict narrowing is the operational semantics of choice for μZ because outermost-in narrowing lacks a transparent definition of forcing in μZ.

We have not formalized a strategy for *needed* narrowing [Antony et al., 1994], which can be considered "state-of-the-art" for narrowing in functional logic languages, since it yields certain optimality results regarding the length of derivations. Needed narrowing is expressible in our framework. Consider a goal like $(\{p_1 \mid e_1\} \cup \ldots \cup \{p_n \mid e_n\}) \cap \{e\}$: needed narrowing would mean that *before* distribution we force e as far as required by *every* p_i – *i.e.* to the shallow structure of the most special pattern, where each p_i is a specialization of. In the worst case, this pattern is just a variable, so that no forcing happens. Actually, we have "definitional trees" [Antony, 1992], as used for defining needed narrowing, directly in the calculus, because we have set union and schemas. However, it seemed more important to support *extensional equality*, which obstructs a transparent notion of forcing for needed narrowing. We should note that extensional equality and unequality is only required to deal with sets of sets – for example for reducing $\{\{e_1\}\} \cap \{\{e_2\}\}$ to 0, if e_1 and e_2 reduce to an extensionally unequal value. If μZ's computation model would be restricted to not use extensional equality and unequality, as it is the default in other frameworks, lazy or needed narrowing could be easily adapted.

Reference Computation Model

The reference computation model given in Section 4.4 (on page 86), using the technique of natural semantics, is a trade-off between computation power and execution efficiency. It prepares a straight-forward transition to an implementation by the abstract machine ZAM, as shown in the next chapter. The model is even more strict than in strict narrowing, using a combination of functional reduction by residuation [Ait-Kaci et al., 1987] and logic resolution. The justification for this decision is, on the one hand, easier traceability of execution by humans, which is important for the application to test data evaluation, and, on the other hand, a strongly assumed better overall performance. The last point draws on experience with functional programming and language implementation: the theoretical result of optimality of lazy or needed reduction does not coincide with the practical experience, because the overhead of laziness has to be paid for significantly,

which often results in that programmers use strictness annotations [Hartel et al., 1996].

The RCM is related to the one described by Hanus [1997] for Curry. Set values in our semantics, $\bigcup\{\overline{v}\} \cup \bigcup [\![\overline{p} \mid \overline{\theta}]\!]$, are comparable to the "definitional trees" used here and in work about narrowing. However, since set values are first-order inhabitants of the operational domain, which definitional trees are not, they can be combined by intersection and union at computation time, and, moreover, augmented by extensional data. This is important in the realm of set-based programming, where sets, relations and functions are often used for representing data structures, which are incrementally constructed.

Unlike the model of Hanus [1997], subresolutions are supported, which are invoked from expression reduction for the μ-operator and for dealing with set tests. This technique corresponds to "deep guards" and "encapsulated search", as found, for example, in the computation model of Oz [Smolka, 1994a] or $\{log\}$ [Dovier et al., 1996]. Our computation model has some similarities to the one given for Oz by Smolka [1994a]. The so-called "computation spaces" in this work, which represent a constraint store, are given in our model as a conjunction of properties paired with assignments. Writing to the constraint store is equivalent in our model to a resolution step, which updates the assignments. A subspace is opened when a subresolution is started for the μ-operator or emptiness-tests. However, as discussed in the previous chapter, there is no equivalent to committed choice in μZ and its computation model.

Computing More

The computation models we have presented are not the end of the story. There are three major potential ways of increasing computation power which are not considered here: *higher-order unification* (or narrowing), *set unification* and *subset logic programming*.

Higher-order unification [Snyder and Gallier, 1989] is used in theorem provers such as Isabelle [Paulson, 1994] for the highly generic formulation of inference rules. Higher-order narrowing is proposed as an extension for functional logic languages [Prehofer, 1994; Hanus and Prehofer, 1999] or logic languages such as λProlog [Nadathur and Miller, 1988]. The underlying idea of both applications is to extend unification to the synthesis of λ-*terms*. Substitution is then augmented by β-reduction. Carrying this potential over to μZ would require to synthesize *schemas* during unification, which needs a generalization of higher-order unification that deals only with functions.

Unification over (finite) sets extends unification on free term algebras to the nonfree algebra of sets, and amounts, technically, to unification modulo an associative, commutative and idempotent equational theory [Siekmann, 1989]. Algorithms and complexity investigations are found, e.g., in [Spratt, 1996; Arenas-Sanchez and Dovier, 1997; Stolzenburg, 1999]. Adapting these techniques to μZ would allow unification on the extensional part of sets. For example, let $S = \{(x,y,z) \mid z = x \cup y\}$. If we are going to bind the variable z to an extensional set, then, with set unification, we expect that x and y are assigned all possible partitions of z. Because of idempotency of set union, these partitions are not necessarily disjoint, which makes up the computational complexity of set unification. Leaving aside efficiency considerations, for the framework of μZ an integrated unification method is ultimately required, which combines higher-order unification with set-unification, yielding an algorithm that can deal with extensional as well as intensional set representations. More research needs to be done in this direction.

The major computation potential *subset logic programming* adds to the mentioned techniques is extending fixed-point construction from equations to subset relations [Moon, 1997; Jayaraman and Moon, 1999]. This technique allows us to compute the *smallest set* satisfying *cyclic subset constraints*. The computation of such fixed-points is quite complex, in the general case. Moon [1997] uses memoization techniques for relations to make it feasible. His model is *first-order*, and further investigation of its possible integration in the higher-order framework of μZ is required.

Mit anderen kann man sich belehren,
Begeistert wird man nur allein.
(Goethe)

Chapter 5

Abstract Machine

This chapter describes an abstract machine, called ZAM, a compilation of μZ to the machine's instruction set and an implementation of the machine in C++. The ZAM realizes the reference computation model (RCM) defined in Section 4.4 (on page 86) in the previous chapter. Its specification in Z is on a level of abstraction that addresses the issues of efficient implementation but is still sufficiently concise to reason about the machine.

The ZAM uses *concurrently executing threads* for constraint resolution (modeling the relation $_ \rightsquigarrow_{\mathcal{C}} _$ of the RCM). Each thread has a local value stack for expression reduction (modeling $_ \stackrel{-}{\longrightarrow}_{\mathcal{E}} _$). In the current implementation of the ZAM in C++, the stack is in fact mapped to registers (which is possible because its size is bound), but for the machine's specification the stack model is better suited since it is more abstract.

The ZAM's threads synchronize via variables: accessing an unbound variable suspends a thread and binding a variable resumes threads waiting for it (called *residuation*, [Ait-Kaci et al., 1987]). Backtracking is implemented by maintaining a choice point stack which is shared by the threads working on a resolution goal. Choice points are created if a thread executes a constraint $p \in v_1 \cup v_2$: in this case, the thread continues its execution with $p \in v_1$ and creates a choice point for the alternative $p \in v_2$. A simple but powerful optimization of the ZAM is to lower the scheduling priority of threads that try to create choice points, giving preference to threads that can continue deterministically under the current variable binding (called the "Andorra Principle", [Warren, 1990]).

The compilation of μZ to ZAM instructions is based on the normalization of expressions introduced in Chapter 4. Together with the stack-based machine model, this makes it easy to formulate and retrace.

This chapter is organized as follows. First, we introduce the model of the ZAM, defining its basic data types and instruction set. It follows a comprehensive specification of the meaning of the instructions in Z's sequential specification style. Next, we give the compilation schema. Finally, we discuss the prototypical implementation of the ZAM in C++, including optimizations such as implementing the reduction stack by registers. Some first benchmarks of the implementation are given. The chapter concludes with a discussion of the results and of related work.

$$
\begin{array}{lll}
\mathit{VarInx} == & \mathbb{N} \\
\mathit{Value} ::= & \mathit{Var} \ \langle\langle \mathit{VarInx} \rangle\rangle \\
\quad | & \mathit{Term} \langle\langle \mathit{CONS} \times \mathrm{seq}\, \mathit{Value} \rangle\rangle \\
\quad | & \mathit{Set} \ \langle\langle \mathrm{seq}\, \mathit{Value} \times \mathrm{seq}\, \mathit{Intension} \rangle\rangle
\end{array}
$$

Figure 5.1: ZAM Values

5.1 Basic Model of the ZAM

The basic model of the ZAM is described in this section. After defining the representation of *values*, *goals* and *threads*, the *instructions* are discussed informally. A formal specification of the machine's instructions is given in the next section.

5.1.1 Values

Figure 5.1 shows the basic types used by the ZAM to represent values. There are three variants of values: *variables* (represented by an index in a variable table, as described later on), *terms* and *sets*[1].

A set value, $\mathit{Set}(\mathit{extens}, \mathit{intens})$, consists of two sequences, the first containing the *extensional* part of a set and the second the *intensional* part (the type *Intension* is described in the next section – it corresponds to an intension of the RCM). Let v_j be the values in *extens* and i_k be the intensions in *intens*; then a set value of the ZAM represents the set value of the RCM

$$\{v_1\} \cup \ldots \cup \{v_n\} \cup i_1 \cup \ldots \cup i_m.$$

The ZAM's representation is well suited for implementing the union and intersection of sets:

- For set union, we simply unite the extensional and intensional parts:

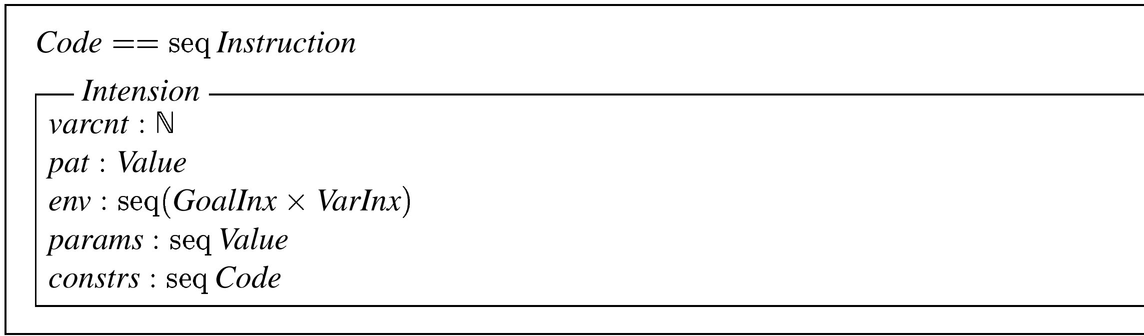
$$
\begin{array}{l}
\mathit{Set}(\mathit{extens}_1, \mathit{intens}_1) \cup \mathit{Set}(\mathit{extens}_2, \mathit{intens}_2) = \\
\qquad \mathit{Set}(\mathit{extens}_1 \sqcup \mathit{extens}_2, \mathit{intens}_1 \sqcup \mathit{intens}_2)
\end{array}
$$

  Here, $_ \sqcup _$ denotes a suitable union function which conforms to extensional equality in the RCM (cf. Section 4.1 (on page 72)). In an implementation, we would in fact use ordered sequences for representing *extens* and *intens*, and their union could thus be implemented with linear time complexity.

- Set intersection is realized as follows (using μZ's notation for intensions):

$$
\begin{array}{l}
\mathit{Set}(\mathit{extens}_1, \mathit{intens}_1) \cap \mathit{Set}(\mathit{extens}_2, \mathit{intens}_2) = \\
\quad \mathit{Set}(\mathit{extens}_1 \sqcap \mathit{extens}_2, \\
\qquad \langle [\![x \mid x \in \mathit{Set}(\mathit{extens}_1, \langle\rangle) \wedge x \in \mathit{Set}(\langle\rangle, \mathit{intens}_2)]\!], \\
\qquad [\![x \mid x \in \mathit{Set}(\mathit{extens}_2, \langle\rangle) \wedge x \in \mathit{Set}(\langle\rangle, \mathit{intens}_1)]\!], \\
\qquad [\![x \mid x \in \mathit{Set}(\langle\rangle, \mathit{intens}_1) \wedge x \in \mathit{Set}(\langle\rangle, \mathit{intens}_2)]\!] \rangle)
\end{array}
$$

[1] In fact, there is also a fourth variant for representing *native* values such as numbers; this, however, is not relevant to the conceptual model of the ZAM.



$Code == \text{seq}\, Instruction$

Intension

$varcnt : \mathbb{N}$
$pat : Value$
$env : \text{seq}(GoalInx \times VarInx)$
$params : \text{seq}\, Value$
$constrs : \text{seq}\, Code$

Figure 5.2: ZAM Intensions

Thus we can construct the disjunctive normal form of intersected set values in a constant number of steps. If any of the $extens_i$ or $intens_i$ is empty, the above construction can be further optimized, since in this case the corresponding constructed intension is known to denote the empty set and can be dropped. In particular, if both sets are completely extensional, we need to compute only the intersection $_ \sqcap _$, which – in an implementation by ordered sequences – can be done in linear time.

5.1.2 Intensions

An intension (Figure 5.2) is described by the *count of variables* it allocates for its execution, a *value* (representing the pattern in an μZ intension $[\![p \mid \theta]\!]$), a sequence of index references representing the *environment* the intension was constructed in, a sequence of values describing *parameters* and a sequence of *constraints*. A constraint is given by a sequence of instructions.

Instantiation. When an intension is instantiated in a goal context (goals are discussed in the next section), the number of variables it requires are allocated in a block of the *variable table* of the goal context (variable tables are just a mapping from variable indices to information about the state of the variable). Then, for each constraint of the intension, a thread is spawned, parameterized by the offset of the allocated variable block. From the executing thread, variables are accessed by adding a static offset found in the code to this dynamic offset.

Consider, for example, the μZ intension $[\![\rho(x,y) \mid \theta_1[x,z] \wedge \theta_2[z,y]]\!]$. Here, z is a "local existential" variable of the intension. This intension is represented in the ZAM as

$$\begin{array}{l} \langle\!| varcnt == 3, pat == Term(\rho, \langle Var\, 0, Var\, 1 \rangle), \\ \; env == \ldots, params == \ldots, \\ \; constrs == \langle \theta_1[0,2], \theta[2,1] \rangle |\!\rangle \end{array}$$

where the offsets used for addressing the variables are 0 for x, 1 for y, and 2 for z. The reentrant instantiation of an intension is realized as follows: let *varshift* be the base offset of the intension's variable block in the instantiation context; then, we simply

- "shift" the pattern *pat*, adding to each variable the offset *varshift* (resulting in $Term(\rho, \langle Var\,(varshift + 0), Var\,(varshift + 1) \rangle))$

- store *varshift* in the threads that execute the constraints θ_1 and θ_2, such that, for each access to a variable from a thread, *varshift* is added to the static offset 0, 1 or 2

Environment. The *environment* of an intension, *env*, is a sequence of indices which refer to pairs of *goal indices* (as discussed in the next section) and a variable block offset. The environment allows access to the variables of the lexical enclosing goal contexts this intension was created in.

For example, in the μZ expression $\mathbf{fix} f \lhd \{(x,y) \mid y = E[f,x]\}$, the variable f is bound by the enclosing fixed-point operator. f is accessed from the intension created for the inner schema via the first element of *env*.

For schemas, the RCM demands that before an according intension can be constructed, all context variables are bound (cf. expression reduction rule $\mathcal{E}3$). However, the situation is different for fixed-points (cf. Rule $\mathcal{E}4$): during the creation of an intension for the fixed-point, the environment may contain "holes" which are bind immediately after the intension has been created, constructing a circular dependency between the intension and its environment. For this reason, it is not possible to directly represent the environment by the bindings of the free variables. An extra level of indirection has to be introduced using goal indices.

Parameters. The *parameters* of an intension, *params*, are used to pass values from the creation context of an intension. In fact, parameters serve a purpose similar to that of the environment; however, they can be used to pass temporary values which do not have a variable counterpart in the context. On the other hand, parameters cannot have cyclic dependencies to the intension they belong to.

5.1.3 Threads and Goals

A *thread* executes the code of a constraint. A *goal* is a collection of threads which together work on a set of variables and which share a stack of choice points (Figure 5.3 (on the facing page)). A goal is created to execute an intension. During execution, further intension may be dynamically instantiated in the context of a goal (corresponding to the membership test of an element in an intension; cf. resolution rule $\mathcal{C}11$ of the RCM.)

For threads and goals, the index types *ThreadInx* and *GoalInx* are defined, which allow references of them via thread and goal tables defined in a configuration of the ZAM (see next section)[2].

A thread refers to two goals: the *parent* where it belongs to and a possible *child*, describing a subresolution. In turn, a subgoal refers via its *parent* field to the thread that executes it. Subgoal resolution corresponds to the execution of the μ-operator (Rules $\mathcal{E}15$-$\mathcal{E}17$ of the RCM) and the test for emptiness (Rules $\mathcal{C}15$-$\mathcal{C}18$).

A thread may be in the status *running*, indicating that it has not yet finished its execution, or it may be in one of the states *success*, *failure* or *error*, indicating that it has stopped executing. The *error* status results from the execution of the μ-operator on a

[2] Instead of using indices here, we could have well used a notion of *memory* and *references*. This would complicate the declarative description of the ZAM in Z, however.

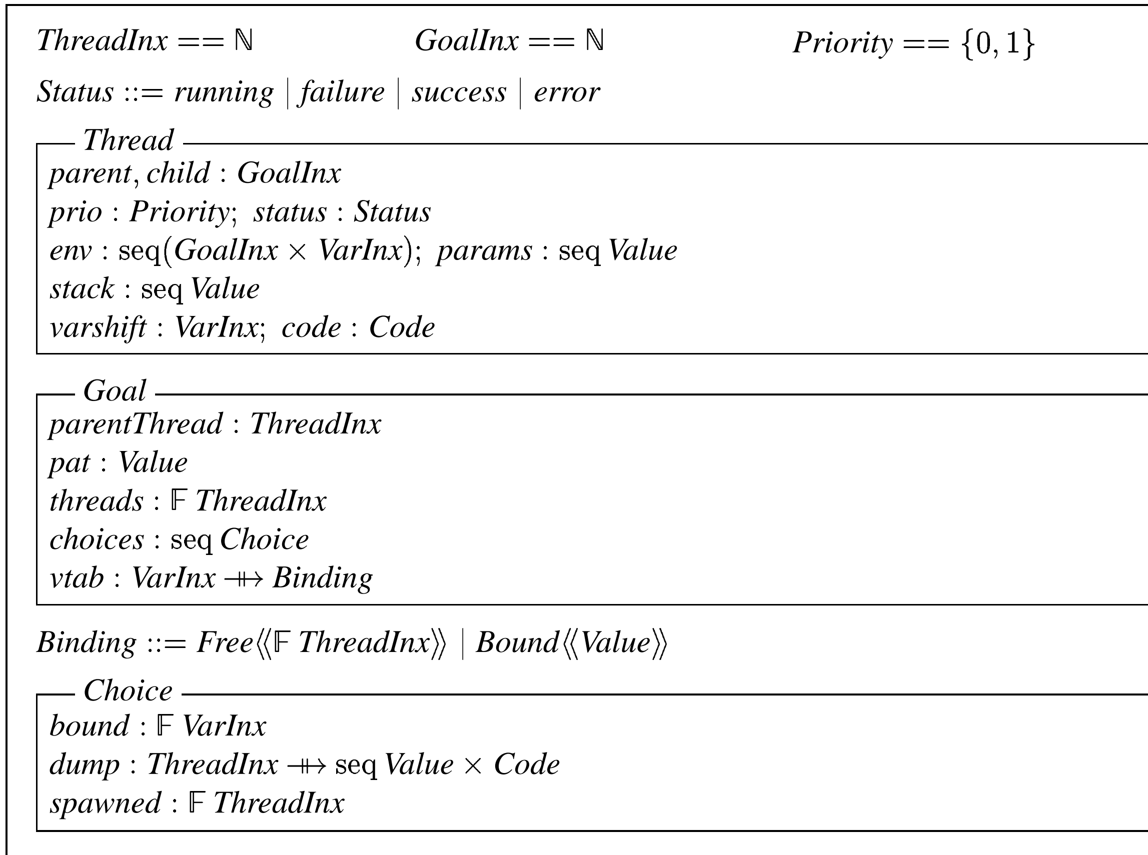


Figure 5.3: ZAM Threads and Goals

nonsingleton set, or from the undecidable equality test of intensions in unification. *failure* corresponds to failure in unification.

A thread inherits the environment of the intension whose constraint it executes as well as the parameters (fields *env* and *params*).

A goal holds the set of threads spawned in its realm. The resolution represented by the goal has succeeded if all its threads are in status *success*; it has failed if one of its threads is in status *failure*.

A goal's field *vtab* represents the variable table as a mapping from variable indices to a value of type *Binding*. If a variable is free, a set of threads may be attached which are waiting for the variable to become bound (variant *Free* of type *Binding*).

The field *choices* of a goal represents the choice stack. During execution of a thread, whenever the thread encounters a constraint such as $p \in v_1 \cup v_2$, it continues execution with $p \in v_1$ and pushes a new choice to the stack, holding the information for the alternative $p \in v_2$ and accumulating changes made to the goal which need to be undone on backtracking:

- *bound* indicates which variables were bound while this choice was active (the active choice is the topmost one on the choice stack). Whenever a variable is bound during unification, it is added to this set. On backtrack, these bindings are undone.

- *dump* saves the (earliest) state of threads which have advanced during this choice was active. Whenever a thread performs an execution step and has not yet dumped

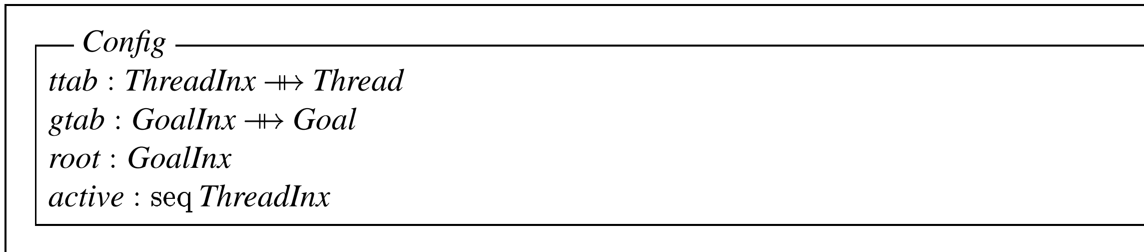


Figure 5.4: ZAM Configurations

its state to the active choice, it now does so. The dump of a thread is described by its stack and by its current constraint (instruction stream). On backtracking, these values are restored[3].

Note that the mapping *dump* also contains the information to resume the thread that initiated this goal with the next alternative, $p \in v_2$. For more details, see the specification (Section 5.2 (on page 105)).

- *spawned* describes the threads that have been spawned while this choice was active. On backtracking, these threads are killed.

The ZAM uses a simple but efficient strategy for thread dumping: whenever a thread executes its next instruction but has not yet dumped its state to the topmost choice, it now does so. This is made possible by choice point scheduling. Only if no other threads can continue (because all other threads are waiting for variables to become bound) is a thread creating a choice point scheduled for execution. Thus, computations that are independent of a particular choice are performed automatically before the corresponding choice point is created.

5.1.4 Configurations

A *configuration* describes the overall state of the ZAM (Figure 5.4). It contains tables of threads and goals, the *root* goal and a sequence of thread indices describing the *active* threads in the order they are to be scheduled (a formal specification of scheduling is given in Section 5.2 (on page 105)).

In the specification of the ZAM, we assume that an unlimited number of threads and goals can be allocated from the tables in a configuration. Similarly, we allocate from the variable table of goals without trying to retrieve “unused” variables. In the implementation in C++, garbage-collection techniques are used to manage this requirement (cf. Section 5.4 (on page 129)).

5.1.5 Instructions

The instructions of the ZAM are summarized in Figure 5.5 (on the facing page). Their meaning is informally described below.

[3] Dumping the entire stack is indeed expansive. In the implementation in C++, instead of stacks for storing temporaries, we use a “single assignment” register model, which makes dumping very efficient. This is discussed in Section 5.4 (on page 129).

$$
\begin{array}{lll}
\mathit{Instruction} ::= & \mathit{LOAD} & \langle\!\langle \mathit{VarInx} \rangle\!\rangle \\
\mid & \mathit{WAIT} & \langle\!\langle \mathit{VarInx} \rangle\!\rangle \\
\mid & \mathit{LOADENV} & \langle\!\langle \mathbb{N}_1 \times \mathit{VarInx} \rangle\!\rangle \\
\mid & \mathit{LOADPAR} & \langle\!\langle \mathbb{N}_1 \rangle\!\rangle \\
\mid & \mathit{STORE} & \langle\!\langle \mathit{VarInx} \rangle\!\rangle \\
\mid & \mathit{UNIFY} & \\
\mid & \mathit{MEMBER} & \\
\mid & \mathit{MKTERM} & \langle\!\langle \mathit{CONS} \times \mathbb{N} \rangle\!\rangle \\
\mid & \mathit{MKEMPTY} & \\
\mid & \mathit{MKSINGLE} & \\
\mid & \mathit{MKVAR} & \langle\!\langle \mathit{VarInx} \rangle\!\rangle \\
\mid & \mathit{MKINTEN} & \langle\!\langle \mathbb{N} \times \mathbb{N} \times \mathrm{seq}\, \mathit{Code} \rangle\!\rangle \\
\mid & \mathit{UNION} & \\
\mid & \mathit{ISECT} & \\
\mid & \mathit{TEST} & \\
\mid & \mathit{TESTNATIVE} & \\
\mid & \mathit{MU} & \\
\mid & \mathit{SUCCESS} &
\end{array}
$$

Figure 5.5: ZAM Instructions

Waiting for Variables. The $\mathit{WAIT}(\mathit{inx})$ instruction lets the executing thread suspend until the addressed variable (and all the variables it indirectly refers to by a possible binding) are bound.

Loading Values. The $\mathit{LOAD}(\mathit{inx})$ instruction loads the variable with the given index from the current goal's variable table to the executing thread's stack. If the variable is bound (maps to the variant *Bound val* in the variable table), the bound value is pushed, otherwise the term *Var inx*.

The $\mathit{LOADENV}(\mathit{dist}, \mathit{inx})$ instruction loads the value of the environment variable *inx* to the stack, from the goal described by the lexical nesting distance *dist*. The environment variable is accessed as follows: let *thr* be the executing thread and $(\mathit{goalInx}, \mathit{varshift}) = \mathit{thr}.\mathit{env}\ \mathit{dist}$ the environment entry; then, the variable $(\mathit{config}.\mathit{goals}\ \mathit{goalInx}).\mathit{vtab}\ (\mathit{varshift} + \mathit{inx})$ is selected. The addressed variable is expected to be bound, and its binding must not refer to other free variables, which has to be guaranteed by the compilation scheme (thus corresponding to the RCM, which ensures that intensions can only be created if context variables are bound or become immediately bound after fixed-point construction).

The $\mathit{LOADPAR}(\mathit{inx})$ instruction loads the parameter value indexed by *inx* ($\mathit{thr}.\mathit{params}\ \mathit{inx}$).

Storing and Unifying Values. The $\mathit{STORE}(\mathit{inx})$ instruction pops a value from the stack and unifies it with the variable addressed by *inx*. The *UNIFY* instruction pops two values from the stack and unifies them. If unification fails, the thread stops in status *failure*; if unification is not decidable (for instance, because it requires the comparison of intensions), it stops in status *error*.

Checking for Membership. The *MEMBER* instruction checks whether the value found beneath the top of the stack is a member of the set on top of the stack. This instruction is the only one that creates choice points.

Before execution of this instruction continues, the priority of the thread is lowered and the thread suspends (if the priority is not already low). Then a choice point is created and the thread proceeds as follows. Let $\mathit{Set}(\mathit{elems}, \mathit{intens})$ be the set on top of the stack, and u the element beneath the top, for which membership is tested:

- If $\#\mathit{elems} = \#\mathit{intens} = 0$, the executing thread stops in status *failure*, after removing its choice point from the stack.
- Otherwise, if $\mathit{elems} = v \cdot\!: \mathit{rest}$, then the top of the stack is replaced by v, and the alternative $\mathit{Set}(\mathit{rest}, \mathit{intens})$ is recorded in the choice. Then. the thread behaves as with the *UNIFY* instruction (the stack consists of $v \cdot\!: u \cdot\!: \ldots$ at this time).
- Otherwise, let $\#\mathit{elems} = 0$ and $\mathit{intens} = \mathit{inten} \cdot\!: \mathit{rest}$. The alternative $\mathit{Set}(\langle\rangle, \mathit{rest})$ is recorded in the choice and the intension *inten* is instantiated:

  1. A block of $\mathit{inten.varcnt}$ variables is allocated in the variable table of the goal. Let the start index of this variable block be called *varshift* in the sequel.
  2. The pattern $\mathit{inten.pat}$ is "shifted" by adding to each variable the offset *varshift* and pushed to the stack.
  3. For each constraint in $\mathit{inten.constrs}$. a thread is spawned and initialized with *varshift*. If the choice stack is not empty, then the new thread is added to the set $\mathit{choice.spawned}$.
  4. The thread behaves as with the *UNIFY* instruction (the stack consists of $\mathit{shift}(\mathit{inten.pat}, \mathit{varshift}) \cdot\!: u$ at this time).

Constructing Values. The $\mathit{MKTERM}(\rho, \mathit{arity})$ instruction pops *arity* number of arguments from the stack and pushes the value $\mathit{Term}(\rho, \mathit{args})$ to the stack. The $\mathit{MKVAR}(\mathit{inx})$ instruction pushes the value *Var inx* to the stack.

The *MKEMPTY* instruction pushes the empty set, $\mathit{Set}(\langle\rangle, \langle\rangle)$, to the stack. The *MKSINGLE* instruction pops the value v from the stack and pushes a singleton set, $\mathit{Set}(\{v\}, \langle\rangle)$.

The $\mathit{MKINTEN}(\mathit{parcnt}, \mathit{varcnt}, \mathit{constrs})$ instruction creates a new intension *inten* and pushes the set $\mathit{Set}(\langle\rangle, \{\mathit{inten}\})$ to the stack. The instruction expects *parcnt* parameters on the stack, which are stored in the *params* field. Above the parameters, on top of the stack, the pattern of the intension is expected, which is stored in the *pat* field. The created intension inherits its environment from the executing thread: let *thr* be this thread, then $\mathit{inten.env} = (\mathit{thr.parent}, \mathit{thr.varshift}) \cdot\!: \mathit{thr.env}$. The constraints of the intension are given by the sequence *constrs*.

Union and Intersection. The *UNION* instruction pops two set values from the stack and pushes their union, constructed as discussed in Section 5.1.1 (on page 98). The *ISECT* instructions pops two set values and pushes their intersection.

Unique Selection. The *MU* instruction pops a value $u = Set(elems, intens)$ from the stack and behaves as follows:

- If $\#elems = \#intens = 0$, or if $\#elems > 1$, then the thread stops in status *error*.
- If $elems = \{v\}$ and $\#intens = 0$, then the thread pushes the value v.
- Otherwise, a reference value is calculated: if $elems = \{v\}$, the reference value is v, otherwise it is the result of the first successful subresolution of one of the intensions. Then, for all backtracks over all subresolutions of all intensions in the set, it is checked whether they yield the same value as the reference value. If this is the case, the reference value is pushed to the stack; otherwise, the thread stops in status *error*.

For more details, the reader is referred to the specification of the ZAM's instructions in the next section.

Test for Emptiness/Nonemptiness. The *TEST* instruction pops a set value from the stack and checks if it is nonempty. If so, it continues; otherwise, it stops in status *failure*. The *TESTNATIVE* instruction fails if the value on the stack is empty.

The test of nonemptiness or emptiness immediately succeeds or fails if the set on the stack has a nonempty extensional part. Otherwise, intensions in the set are tried to resolve in a subresolution.

Succeeding. The *SUCCESS* instruction lets the executing thread stop in status *Succeed*.

5.2 Specification

In this section, the instructions of the ZAM are comprehensively specified in Z's sequential specification style (cf. e.g. Woodcock and Davies [1996]). The specification is intended to be executable by the ZAM itself.

5.2.1 Axiomatic Definitions

A set of axiomatic definitions for working with the data types of the ZAM is defined in this section.

Working with Values

The auxiliary function *shift offs v* yields a value where *offs* is added to each variable's index (Figure 5.6 (on the following page)). Since, by construction, variables cannot appear in sets, the definition needs only to recurse over term values.

The auxiliary function *freeze vtab v* returns a value where all variables in v which are bound in *vtab* are replaced by their actual binding (Figure 5.7 (on the next page)).

The auxiliary function *free v* returns the set of indices of variables found in the value v (Figure 5.8 (on the following page)). Variables are only expected to occur in terms, not in sets.

$$
\begin{array}{l}
shift : VarInx \longrightarrow Value \longrightarrow Value \\
\hline
shift\ offs\ (Var\ i) \Leftarrow Var(i + offs) \\
\quad \mid (Term(\rho, vs)) \Leftarrow Term(\rho, shift\ offs \circ vs) \\
\quad \mid v \Leftarrow v \\
\quad \mathbf{where}\ offs : VarInx;\ v : Value;\ vs : \mathrm{seq}\ Value;\ \rho : CONS;\ i : VarInx
\end{array}
$$

Figure 5.6: Shifting Values

$$
\begin{array}{l}
freeze : (VarInx \twoheadrightarrow Binding) \longrightarrow Value \longrightarrow Value \\
\hline
freeze\ vt\ (Var\ i) \Leftarrow \mathbf{if}_{\exists_1}\ v' : Value \mid i \in \mathsf{dom}\ vt \wedge vt\ i = Bound\ v' \\
\qquad \mathbf{then}\ freeze\ vt\ v' \\
\qquad \mathbf{else}\ \ Var\ i \\
\quad \mid (Term(\rho, vs)) \Leftarrow Term(\rho, freeze\ vt \circ vs) \\
\quad \mid v \Leftarrow v \\
\quad \mathbf{where}\ vt : VarInx \twoheadrightarrow Binding;\ v : Value;\ vs : \mathrm{seq}\ Value \\
\qquad \rho : CONS;\ i : VarInx
\end{array}
$$

Figure 5.7: Freezing Values

Equality and Unification

The (partial) auxiliary function $equal(v_1, v_2)$ determines equality of two values (Figure 5.9 (on the next page)). Confirming to extensional equality in μZ's RCM, the function is undefined if the values cannot be compared, which happens if sets are tried to compare which have a nonempty intensional part.

The (partial) auxiliary function $unify(vt, u, v)$ tries to unify the given values u and v, under the variable table vt (Figure 5.10 (on page 108)). On success, the function returns a new variable table and the set of indices of variables which have been bound. The definition is based on a function *uni* whose type is appropriate for sequence catamorphisms, and a function *uniVar* which unifies a variable with a value. The function may be undefined for similar reasons as the *equal* function (since it uses *equal* to determinate equality of sets). The function *uniVar* makes an occurrence test before it binds a variable.

Merging and Joining Value Sequences

The auxiliary function $vs \sqcup us$ constructs the union of two value sequences, removing any duplicates according to *equal*. It is undefined if equality is undefined (Figure 5.11). In

$$
\begin{array}{l}
vars : Value \longrightarrow \mathbb{F}\ VarInx \\
\hline
vars\ Var\ i \Leftarrow \{i\} \\
\quad \mid Term(\rho, vs) \Leftarrow \bigcup\{v : \mathsf{ran}\ vs \bullet vars\ v\} \\
\quad \mid v \Leftarrow \varnothing \\
\quad \mathbf{where}\ i : VarInx;\ v : Value;\ vs : \mathrm{seq}\ Value;\ \rho : CONS
\end{array}
$$

Figure 5.8: Getting the Variables of a Value

$$
\begin{array}{l}
equal : Value \times Value \rightarrow\!\!\!\!\!\!+ \;\mathbb{B} \\
\hline
equal\ (Var\, i, Var\, i) \Leftarrow \mathsf{true} \\
\quad \mid\ (Term(\rho, vs_1), Term(\rho, vs_2)) \Leftarrow [\mid \#vs_1 = \#vs_2; \\
\qquad\qquad \forall\, i : \mathsf{dom}\, vs_1 \bullet equal(vs_1\, i, vs_2\, i)] \\
\quad \mid\ (Set(es_1, \langle\rangle), Set(es_2, \langle\rangle)) \Leftarrow [\mid \#es_1 = \#es_2; \\
\qquad\qquad \forall\, v : \mathsf{ran}\, es_1 \bullet \exists\, u : \mathsf{ran}\, es_2 \bullet \\
\qquad\qquad\qquad equal(v, u)] \\
\quad \mid\ (Set(es_1, is_1), Set(es_2, is_2)) \Leftarrow \bot \\
\quad \mid\ (v, u) \Leftarrow \mathsf{false} \\
\quad \textbf{where}\ i : VarInx;\ \rho : CONS;\ vs_1, vs_2 : \mathrm{seq}\, Value \\
\qquad\qquad es_1, es_2 : \mathrm{seq}\, Value;\ is_1, is_2 : \mathrm{seq}\, Intension;\ v, u : Value
\end{array}
$$

Figure 5.9: Equality

order to remove undeterminism in the ordering of the result, we assume a (unspecified) predicate *someorder* on value sequences. Similarly, the function $vs \sqcap us$ constructs the intersection of two value sequences.

Adding Information to Choices

The auxiliary function *addBound* adds information to the *bound* field of a choice, the function *addDump* to the *dump* field and the function *addSpawned* to the *spawned* field (Figure 5.12).

Ordering Index Sets

Sometimes, we require to order a finite set of indices (for variables, threads and goals). The function *orderinx* implements some (fixed but arbitrary) order on index sets (Figure 5.13 (on page 109)), yielding a sequence containing the order.

5.2.2 Auxiliary Instructions

For representing *internal states* of threads auxiliary instructions are used. These can be considered as a kind of “micro programs” implementing other instructions. For example, let $MEMBER \cdot\!: rest$ be the instruction stream of a thread. For executing the *MEMBER* instruction, a choice point needs to be created, and the thread initiating the choice needs to dump a state that continues with an alternative to the choice. We realize this by dumping

$$TRYNEXT(pat, extens, intens) \cdot\!: rest$$

as the instruction state of the thread, where *TRYNEXT* is an auxiliary instruction executing the next alternative ($pat \in Set(extens, intens)$). A similar technique is used for specifying subresolutions of the *MU*, *TEST*, and *TESTNATIVE* instructions.

The auxiliary instructions are declared in Figure 5.14 (on page 109), and will be explained as they are used.

$$UniRes ::= Ok\langle\langle (VarInx \mathrel{+\!\!\!\!\!\!\mapsto\!\!\!\!\mapsto} Binding) \times \mathbb{F}\, VarInx \rangle\rangle \mid Fail$$

$$
\begin{array}{l}
unify : (VarInx \mathrel{-\!\!\!\!\!+\!\!\!\!\!\twoheadrightarrow} Binding) \times Value \times Value \mathrel{+\!\!\!\!\!\rightarrow} UniRes \\
uni : (Value \times Value) \times UniRes \mathrel{+\!\!\!\!\!\rightarrow} UniRes \\
uniVar : VarInx \times Value \times UniRes \mathrel{+\!\!\!\!\!\rightarrow} UniRes \\
\hline
unify = \lambda\, vt : VarInx \mathrel{-\!\!\!\!\!+\!\!\!\!\!\twoheadrightarrow} Binding;\ u, v : Value \bullet uni((u,v), Ok(vt, \varnothing)) \\[1ex]
uni\ ((u,v), Fail) \quad \Leftarrow Fail \\
\quad \mid ((Var\, i, Var\, i), res) \quad \Leftarrow res \\
\quad \mid ((Var\, i, v), res) \quad \Leftarrow uniVar(i, v, res) \\
\quad \mid ((u, Var\, i), res) \quad \Leftarrow uniVar(i, u, res) \\
\quad \mid ((Term(\rho, us), Term(\rho, vs)), res) \Leftarrow \mathbf{if}\ \#us = \#vs \\
\qquad\qquad \mathbf{then}\ (uni, res) \mathbin{/} \mathsf{zip}\ (us, vs) \\
\qquad\qquad \mathbf{else}\ \ Fail \\
\quad \mid ((u,v), Ok(vt, b)) \quad \Leftarrow \mathbf{if}\ equal(freeze\ vt\ u, freeze\ vt\ v) \\
\qquad\qquad \mathbf{then}\ Ok(vt, b) \\
\qquad\qquad \mathbf{else}\ \ Fail \\
\quad \mathbf{where}\ u, v : Value;\ us, vs : \mathrm{seq}\, Value;\ i : VarInx;\ \rho : CONS \\
\qquad\quad res : UniRes;\ vt : VarInx \mathrel{-\!\!\!\!\!+\!\!\!\!\!\twoheadrightarrow} Binding;\ b : \mathbb{F}\, VarInx \\[1ex]
uniVar\ (i, v, Ok(vt, b)) \Leftarrow \mathbf{if}_{\exists_1}\ u : Value \mid vt\ i = Bound\ u \\
\qquad\qquad \mathbf{then}\ uni((u,v), Ok(vt, b)) \\
\qquad\qquad \mathbf{else}\ \ \mathbf{if}\ i \notin vars(freeze\ vt\ v) \\
\qquad\qquad\qquad \mathbf{then}\ Ok(vt \oplus \{i \mapsto Bound\ v\}, b \cup \{i\}) \\
\qquad\qquad\qquad \mathbf{else}\ \ Fail \\
\quad \mathbf{where}\ i : VarInx;\ v : Value;\ vt : VarInx \mathrel{-\!\!\!\!\!+\!\!\!\!\!\twoheadrightarrow} Binding;\ b : \mathbb{F}\, VarInx
\end{array}
$$

Figure 5.10: Unification

$$
\begin{array}{l}
_ \sqcup _,\ _ \sqcap _ : \mathrm{seq}\, Value \times \mathrm{seq}\, Value \mathrel{+\!\!\!\!\!\rightarrow} \mathrm{seq}\, Value \\
someorder\ _ : \mathbb{P}(\mathrm{seq}\, Value) \\
\hline
(_ \sqcup _)\ (vs, us) \Leftarrow \\
\qquad (\mu\, ws : \mathrm{seq}\, Value \mid \\
\qquad\qquad \mathsf{ran}\, ws = \{w : Value \mid \exists\, v : \mathsf{ran}\, vs \cup \mathsf{ran}\, us \bullet equal(w, v)\}; \\
\qquad\qquad \neg\ \exists\, i, j : \mathsf{dom}\, ws \mid i \neq j \bullet equal(ws\, i, ws\, j);\ someorder\, ws) \\
\quad \mathbf{where}\ vs, us : \mathrm{seq}\, Value \\
(_ \sqcap _)\ (vs, us) \Leftarrow \\
\qquad (\mu\, ws : \mathrm{seq}\, Value \mid \\
\qquad\qquad \mathsf{ran}\, ws = \{w : Value \mid \exists\, v : \mathsf{ran}\, vs;\ u : \mathsf{ran}\, us \bullet equal(w, v) \wedge \\
\qquad\qquad\qquad equal(w, u)\}; \\
\qquad\qquad \neg\ \exists\, i, j : \mathsf{dom}\, ws \mid i \neq j \bullet equal(ws\, i, ws\, j);\ someorder\, ws) \\
\quad \mathbf{where}\ vs, us : \mathrm{seq}\, Value
\end{array}
$$

Figure 5.11: Merging and Joining Value Sequences

$$
\begin{array}{l}
addBound : \mathbb{F}\, VarInx \times Choice \longrightarrow Choice \\
addDump : ThreadInx \times \mathrm{seq}\, Value \times Code \times Choice \longrightarrow Choice \\
addSpawned : \mathbb{F}\, ThreadInx \times Choice \longrightarrow Choice \\
\hline
addBound\ (b, c) \Leftarrow \\
\quad \mu\, \Delta[bound]Choice \mid \\
\qquad \theta Choice = c;\ bound' = bound \cup b \bullet \theta Choice' \\
\quad \mathbf{where}\ b : \mathbb{F}\, VarInx;\ c : Choice \\
addDump\ (t, stack, code, c) \Leftarrow \\
\quad \mu\, \Delta[dump]Choice \mid \\
\qquad \theta Choice = c;\ dump' = dump \oplus \{t \mapsto (stack, code)\} \bullet \theta Choice' \\
\quad \mathbf{where}\ t : ThreadInx;\ stack : \mathrm{seq}\, Value;\ code : Code;\ c : Choice \\
addSpawned\ (ts, c) \Leftarrow \\
\quad \mu\, \Delta[spawned]Choice \mid \\
\qquad \theta Choice = c;\ spawned' = spawned \cup ts \bullet \theta Choice' \\
\quad \mathbf{where}\ ts : \mathbb{F}\, ThreadInx;\ c : Choice
\end{array}
$$

Figure 5.12: Adding Information to Choices

$$
\begin{array}{l}
orderinx : \mathbb{F}\, \mathbb{N} \longrightarrow \mathrm{seq}\, \mathbb{N} \\
\hline
\mathsf{ran}(orderinx\, s) = s \\
\quad \mathbf{where}\ s : \mathbb{F}\, \mathbb{N}
\end{array}
$$

Figure 5.13: Ordering Index Sets

5.2.3 Execution Step and Selection

The state over which instructions are specified, and a single execution step of the ZAM, are defined in Figure 5.15 (on the next page). The state, called a *selection*, is an enrichment of the ZAM's configuration, *Config*, by the inclusion of those thread's and goal's data fields which are selected for execution. The enrichment allows a direct access to these fields in the specification, supporting Z's sequential specification style.

The *execution step* of the ZAM chooses a thread to execute based on the order given by the *active* sequence of a configuration and of thread priorities. It initializes the data fields from the thread and goal tables, dispatches the instruction (dispatching is defined later on), and stores the data fields contained in the selection back to the configuration.

In Figure 5.15 (on the following page), the signature of the operation *Schedule* is

$$
\begin{array}{lll}
Instruction & ::= & \ldots \\
 & \mid & TRYNEXT \langle\!\langle Value \times \mathrm{seq}\, Value \times \mathrm{seq}\, Intension \rangle\!\rangle \\
 & \mid & SEARCH\ \ \langle\!\langle SearchState \times SearchMode \times \mathrm{seq}\, Intension \rangle\!\rangle \\
SearchState & ::= & Select \mid Resume \mid Backtrack \\
SearchMode & ::= & MuFirst \mid MuNext \langle\!\langle Value \rangle\!\rangle \mid Test \mid TestN
\end{array}
$$

Figure 5.14: ZAM Auxiliary Instructions

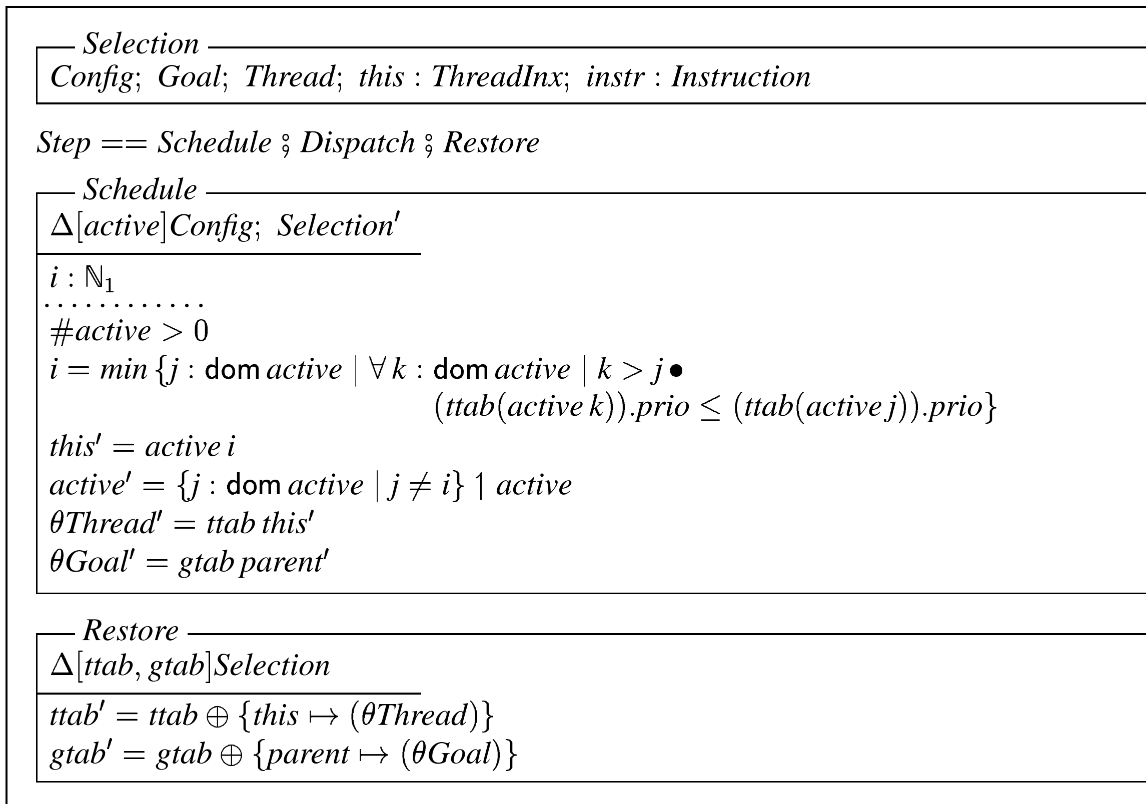


Figure 5.15: ZAM Execution Step

$[\Delta Config;\ Selection']$, the signature of the operation *Dispatch* is $[\Delta Selection]$, and the signature of the operation *Restore* is $[\Delta Config;\ Selection]$. Thus the signature of the overall *Step* is $[\Delta Config]$: one configuration is mapped into the next one, with the help of the intermediate data in the selection.

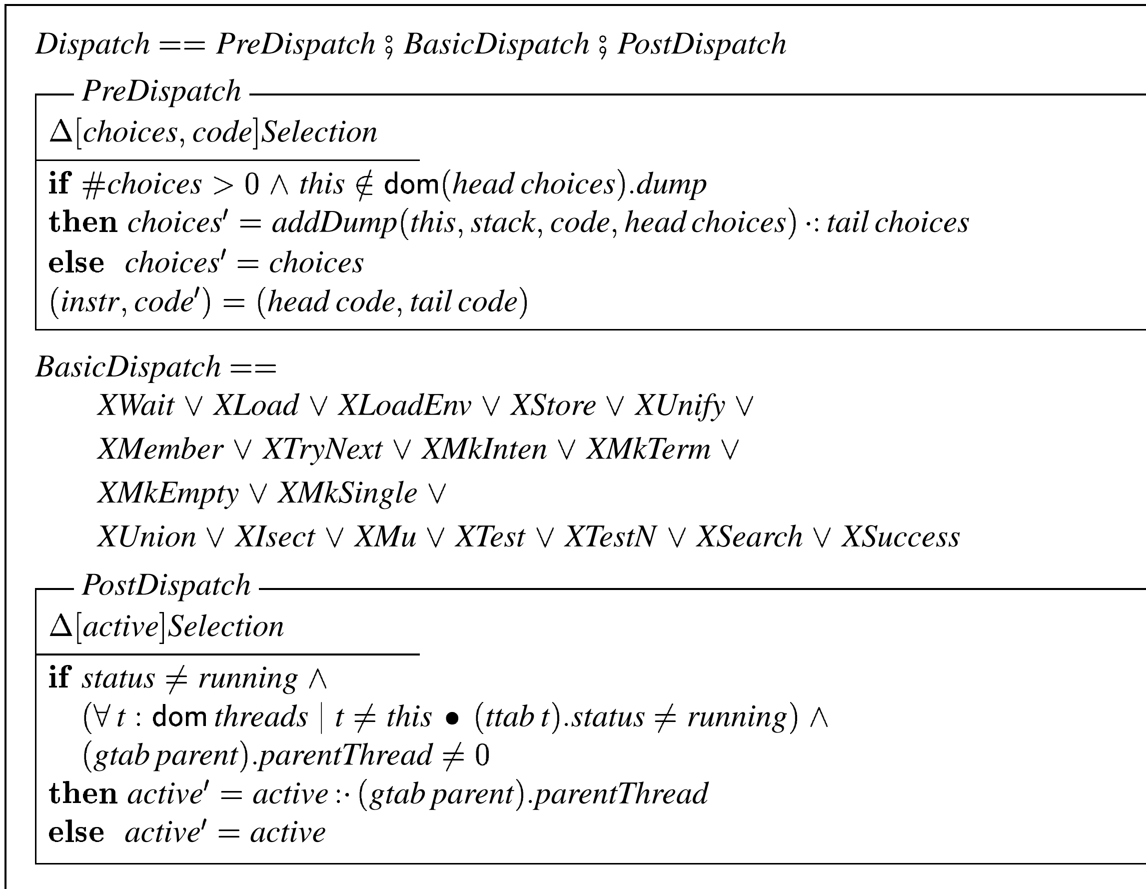
Dispatching is defined in Figure 5.16 (on the next page). The basic dispatch operation is enclosed by two operations, *PreDispatch* and *PostDispatch*, which perform actions common to the execution of every instruction:

- *PreDispatch* dumps the executing thread's state (stack and instruction stream) to the active choice, if one exists and if a dump to this choice has not already been performed (cf. discussion of dumping in Section 5.1.3 (on page 100)). It then loads the next instruction into the *instr* field of the operation, and advances the instruction stream, *code*.

- *PostDispatch* checks whether the threads belonging to the current goal have finished. If this is the case, and the goal has a parent thread, this parent is resumed.

5.2.4 Instructions

The meaning of the instructions of the ZAM is specified in the sequel. The reader is referred to Section 5.1.5 (on page 102) for an informal description of the instructions.



$Dispatch == PreDispatch \mathbin{;} BasicDispatch \mathbin{;} PostDispatch$

PreDispatch

$\Delta[choices, code]Selection$

$\mathbf{if}\ \#choices > 0 \land this \notin \mathsf{dom}(head\ choices).dump$
$\mathbf{then}\ choices' = addDump(this, stack, code, head\ choices) \cdot\!: tail\ choices$
$\mathbf{else}\ \ choices' = choices$
$(instr, code') = (head\ code, tail\ code)$

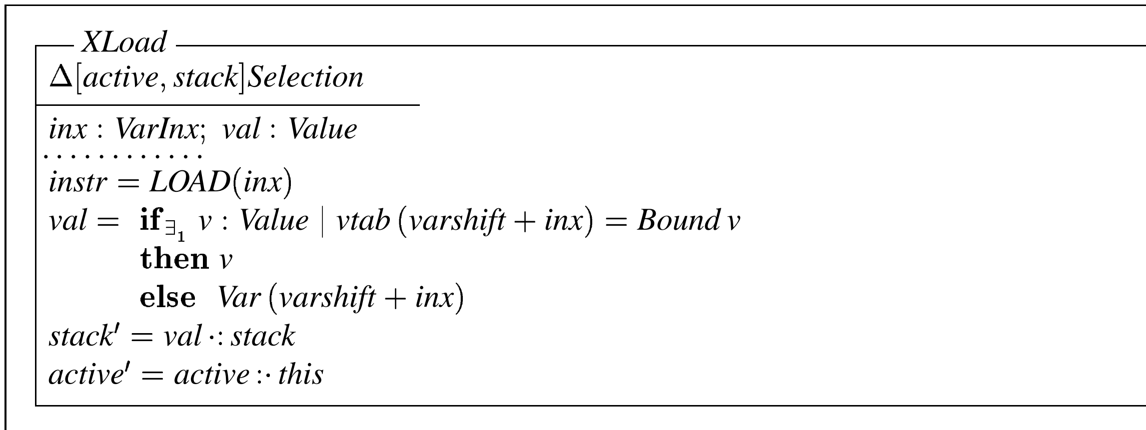
$BasicDispatch ==$
$\quad XWait \lor XLoad \lor XLoadEnv \lor XStore \lor XUnify \lor$
$\quad XMember \lor XTryNext \lor XMkInten \lor XMkTerm \lor$
$\quad XMkEmpty \lor XMkSingle \lor$
$\quad XUnion \lor XIsect \lor XMu \lor XTest \lor XTestN \lor XSearch \lor XSuccess$

PostDispatch

$\Delta[active]Selection$

$\mathbf{if}\ status \neq running \land$
$\quad (\forall t : \mathsf{dom}\ threads \mid t \neq this \bullet (ttab\ t).status \neq running) \land$
$\quad (gtab\ parent).parentThread \neq 0$
$\mathbf{then}\ active' = active \,:\!\cdot\, (gtab\ parent).parentThread$
$\mathbf{else}\ \ active' = active$

Figure 5.16: ZAM Dispatching



XLoad

$\Delta[active, stack]Selection$

$inx : VarInx;\ val : Value$

$instr = LOAD(inx)$
$val = \mathbf{if}_{\exists_1}\ v : Value \mid vtab\,(varshift + inx) = Bound\ v$
$\qquad \mathbf{then}\ v$
$\qquad \mathbf{else}\ \ Var\,(varshift + inx)$
$stack' = val \cdot\!: stack$
$active' = active \,:\!\cdot\, this$

Figure 5.17: The *LOAD* Instruction

The *LOAD* Instruction

The $LOAD(inx)$ (Figure 5.17) instruction loads the variable, addressed by $varshift + inx$, or its binding, to the stack (recall that *varshift* is the offset added for the variable instances of the executing thread).

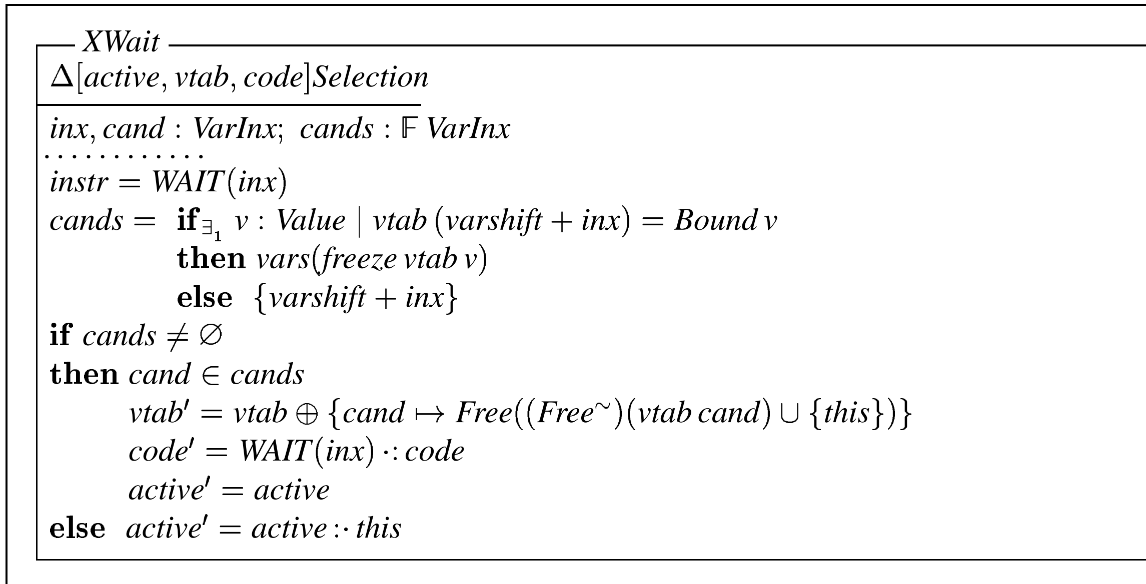


Figure 5.18: The *WAIT* Instruction

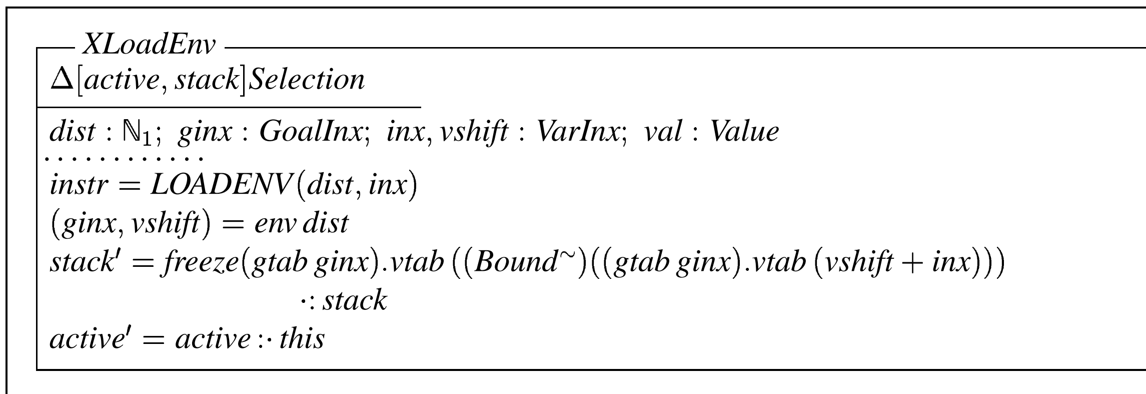


Figure 5.19: The *LOADENV* Instruction

The *WAIT* Instruction

The $WAIT(inx)$ instruction (Figure 5.18) calculates the set of free variables, the index $varhshift + inx$ refers to directly or indirectly. If this set is nonempty, it takes an arbitrary one of it, adds the running thread to the waiting set of this variable, and prepares to execute *WAIT* again when the thread is resumed. Otherwise, the thread just continues.

The *LOADENV* Instruction

The $LOADENV(dist, inx)$ instruction loads the variable *inx* from the goal of the lexical distance *dist* to the stack. The environment variable is expected to be bound, as well as all variables its binding refers to. The loaded value is frozen using the variable table of the goal context. Freezing is essential since the variable indices appearing in the binding of the environment variable are local to the environment's goal, and have no meaning in the current context.

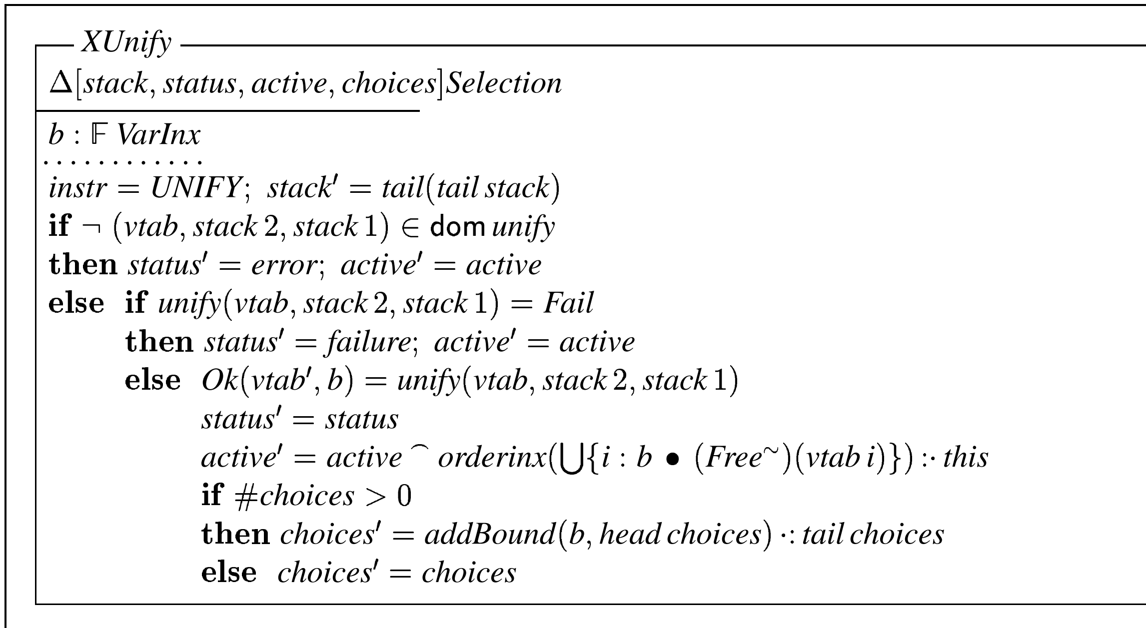


Figure 5.20: The *UNIFY* Instruction

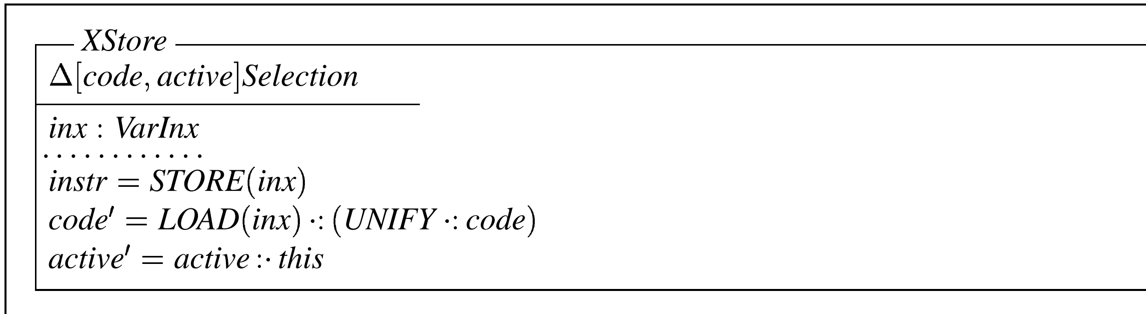


Figure 5.21: The *STORE* Instruction

The *UNIFY* Instruction

The *UNIFY* instruction (Figure 5.20) pops two values from the stack and unifies them. If unification fails, the executing thread fails; if unification is undefined, the thread switches to the *error* status. Otherwise, the variable table is updated, threads which are waiting for the bound variables are resumed and – if the choice stack is not empty – the variables bound by the unification are recorded in the active choice.

The *STORE* Instruction

The $STORE(inx)$ instruction (Figure 5.21) is explained by the code substitution $LOAD(inx);\ UNIFY$.

The *MEMBER* Instruction

The *MEMBER* instruction (Figure 5.22 (on the following page)) is implemented by the auxiliary instruction *TRYNEXT*. On execution of *MEMBER*, first the priority of the running thread is lowered, if it is not already low. Otherwise, a choice point is created, and the instruction *TRYNEXT* is scheduled for execution.

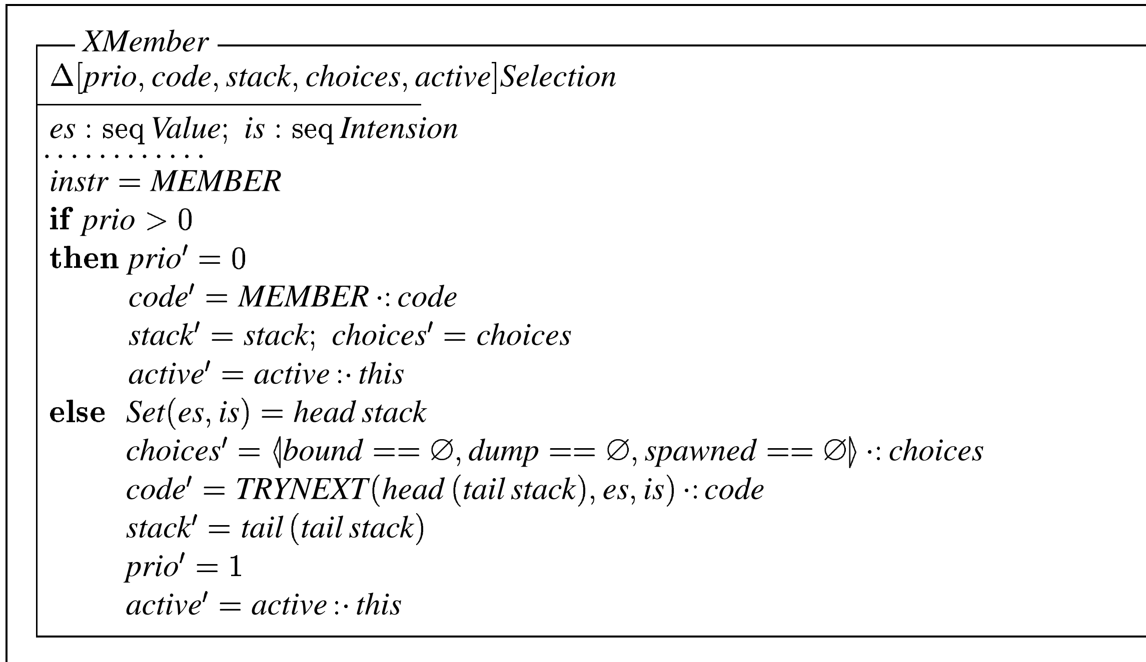


Figure 5.22: The *MEMBER* Instruction

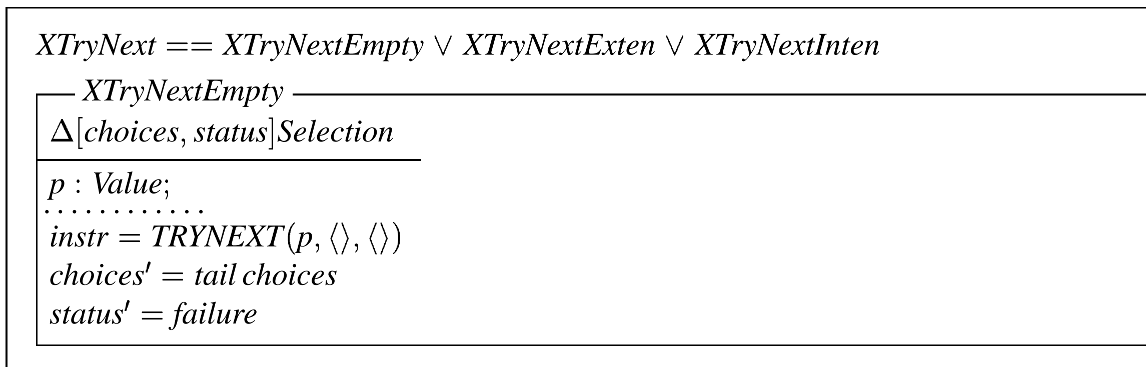
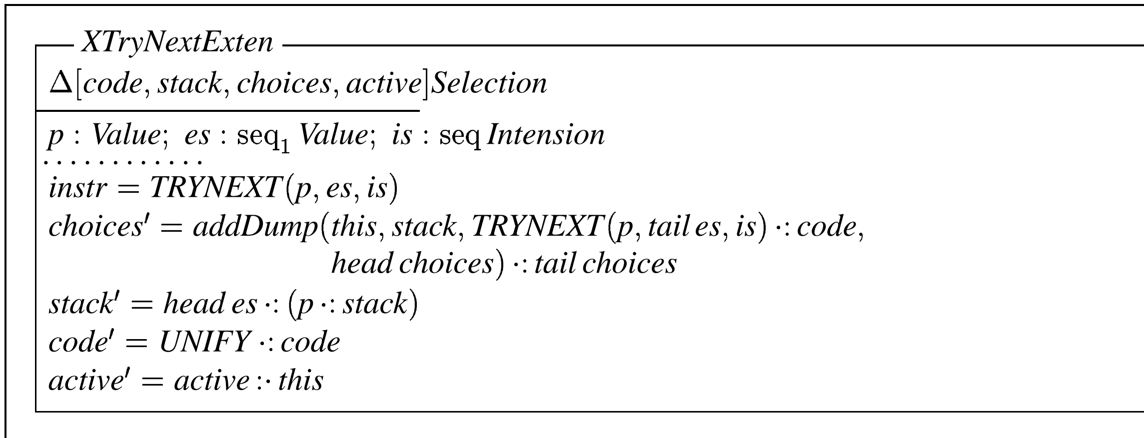


Figure 5.23: The *TRYNEXT* Auxiliary Instruction: The Empty Case

The instruction $TRYNEXT(p, es, is)$ is specified by three cases, depending on the value of es and is. If both es and is are empty, the executing thread fails after removing the corresponding choice from the choice stack (Figure 5.23).

If the extensional part of $TRYNEXT(p, es, is)$ is not empty, one value is extracted, a continuation with the remaining values is dumped to the choice, and p is unified with the extracted value (Figure 5.24 (on the facing page)).

If the extensional part of $TRYNEXT(p, es, is)$ is empty, but the intensional part is not empty, one intension i is extracted and instantiated in the context of the current goal (Figure 5.25 (on the next page)). The variable table is extended by $i.varcnt$ free variables. For each constraint in $i.constrs$, a thread is spawn and initialized with the offset of the allocated variables and the intension's stored environment and parameters. The spawned threads are added to the active choice, and the alternative for this choice is dumped, similar as in the extensional case. The thread continues unifying p with the shifted intension's pattern, *shift vshift inten.pat*.



XTryNextExten

$\Delta[code, stack, choices, active]Selection$

$p : Value;\ es : \mathrm{seq}_1\ Value;\ is : \mathrm{seq}\ Intension$

$instr = TRYNEXT(p, es, is)$

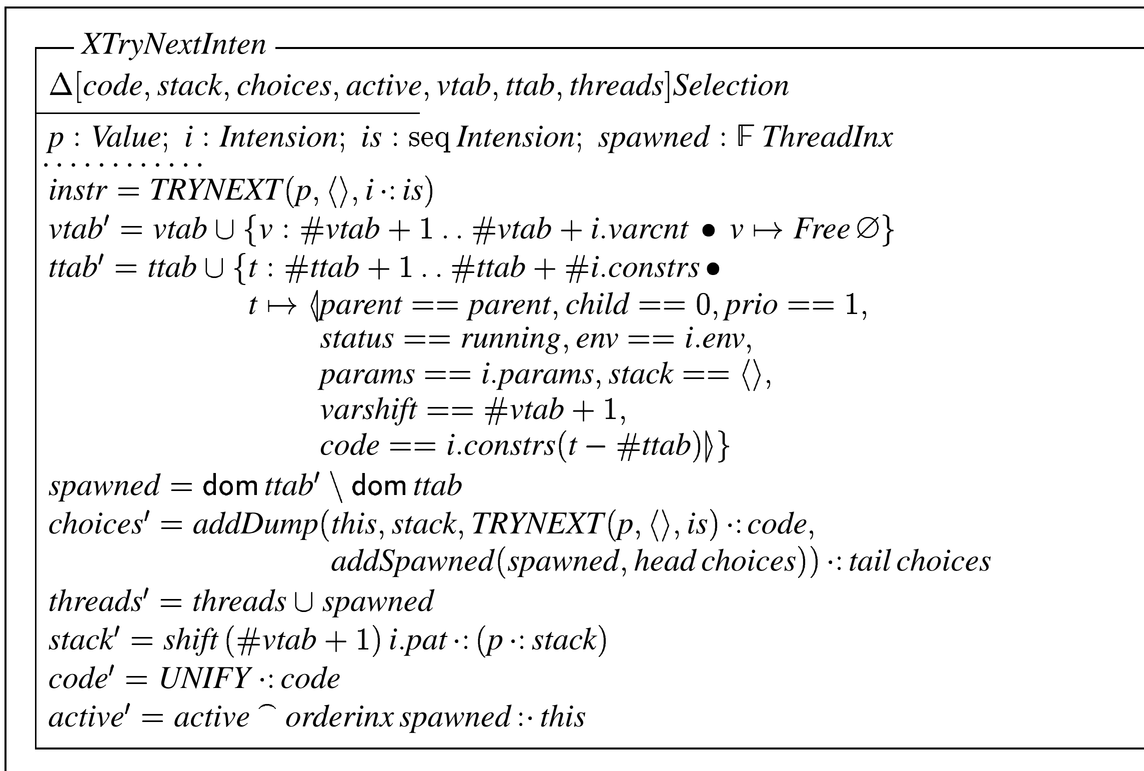
$choices' = addDump(this, stack, TRYNEXT(p, tail\ es, is) \cdot: code, head\ choices) \cdot: tail\ choices$

$stack' = head\ es \cdot: (p \cdot: stack)$

$code' = UNIFY \cdot: code$

$active' = active :\cdot this$

Figure 5.24: The *TRYNEXT* Auxiliary Instruction: The Extensional Case



XTryNextInten

$\Delta[code, stack, choices, active, vtab, ttab, threads]Selection$

$p : Value;\ i : Intension;\ is : \mathrm{seq}\ Intension;\ spawned : \mathbb{F}\ ThreadInx$

$instr = TRYNEXT(p, \langle\rangle, i \cdot: is)$

$vtab' = vtab \cup \{v : \#vtab + 1 \,..\, \#vtab + i.varcnt \bullet v \mapsto Free\ \varnothing\}$

$ttab' = ttab \cup \{t : \#ttab + 1 \,..\, \#ttab + \#i.constrs \bullet t \mapsto \langle\!| parent == parent, child == 0, prio == 1, status == running, env == i.env, params == i.params, stack == \langle\rangle, varshift == \#vtab + 1, code == i.constrs(t - \#ttab) |\!\rangle\}$

$spawned = \mathrm{dom}\ ttab' \setminus \mathrm{dom}\ ttab$

$choices' = addDump(this, stack, TRYNEXT(p, \langle\rangle, is) \cdot: code, addSpawned(spawned, head\ choices)) \cdot: tail\ choices$

$threads' = threads \cup spawned$

$stack' = shift\ (\#vtab + 1)\ i.pat \cdot: (p \cdot: stack)$

$code' = UNIFY \cdot: code$

$active' = active ^\frown orderinx\ spawned :\cdot this$

Figure 5.25: The *TRYNEXT* Auxiliary Instruction: The Intensional Case

The Make Instructions

The *MKEMPTY* instruction pushes an empty set to the stack. The *MKSINGLE* instructions pops a value from the stack and pushes a singleton set containing this value (Figure 5.26 (on the following page)).

The $MKVAR(inx)$ instruction (Figure 5.27 (on the next page)) pushes the value *Var inx* to the stack. The $MKTERM(\rho, n)$ instruction pops n arguments from the stack and pushes the value $Term(\rho, args)$.

The $MKINTEN(parcnt, varcnt, constrs)$ instruction (Figure 5.28) creates a new intension and pushes it to the stack; the environment of the intension is created from the executing threads context.

XMkEmpty

$\Delta[stack, active]Selection$

$instr = MKEMPTY$
$stack' = Set(\langle\rangle, \langle\rangle) \cdot\!: stack$
$active' = active :\!\cdot\ this$

XMkSingle

$\Delta[stack, active]Selection$

$instr = MKSINGLE$
$stack' =$
$\quad Set(\langle head\ stack\rangle, \langle\rangle) \cdot\!: tail\ stack$
$active' = active :\!\cdot\ this$

Figure 5.26: The *MKEMPTY* and *MKSINGLE* Instructions

XMkVar

$\Delta[stack, active]Selection$

$inx : VarInx$

$instr = MKVAR(inx)$
$stack' = Var(inx) \cdot\!: stack$
$active' = active :\!\cdot\ this$

XMkTerm

$\Delta[stack, active]Selection$

$\rho : CONS;\ n : \mathbb{N}$

$instr = MKTERM(\rho, n)$
$stack' = Term(\rho, (1\,..\,n) \upharpoonleft stack)$
$\qquad \cdot\!: ((n+1\,..\,\#stack) \upharpoonleft stack);\ active' = active :\!\cdot\ this$

Figure 5.27: The *MKTERM* Instruction

The *UNION* and *ISECT* Instruction

The *UNION* instruction pops two values from the stack and builds the union of them (Figure 5.29 (on the facing page)). If the union is undefined (because it requires equality on values which cannot be computed) the thread stops in status *error*.

The *ISECT* instruction pops two values from the stack and builds the intersection of

XMkInten

$\Delta[stack, active]Selection$

$parcnt, varcnt : \mathbb{N};\ constrs : \mathrm{seq}\ Code$

$instr = MKINTEN(parcnt, varcnt, constrs)$
$stack' = Set(\langle\rangle, \langle\langle\!| varcnt == varcnt, pat == head\ stack,$
$\qquad params == (1\,..\,parcnt) \upharpoonleft tail\ stack,$
$\qquad env == (parent, varshift) \cdot\!: env, constrs == constrs |\!\rangle\rangle$
$\qquad) \cdot\!: (parcnt + 1\,..\,\#stack) \upharpoonleft tail\ stack$
$active' = active :\!\cdot\ this$

Figure 5.28: The *MKINTEN* Instruction

$$
\begin{array}{l}
\textit{XUnion} \\
\Delta[\textit{active}, \textit{stack}, \textit{status}]\textit{Selection} \\
\hline
es_1, es_2 : \mathrm{seq}\ \textit{Value};\ \ is_1, is_2 : \mathrm{seq}\ \textit{Intension} \\
\hline
\textit{instr} = \textit{UNION} \\
\textit{stack}\,2 = \textit{Set}(es_1, is_1);\ \ \textit{stack}\,1 = \textit{Set}(es_2, is_2) \\
\textbf{if}\ (es_1, es_2) \in \mathsf{dom}(_ \sqcup _) \\
\textbf{then}\ \textit{stack}' = \textit{Set}(es_1 \sqcup es_2, is_1 \frown is_2) \cdot\!: \textit{tail}\,(\textit{tail}\,\textit{stack}) \\
\qquad \textit{active}' = \textit{active} :\!\cdot\ \textit{this};\ \ \textit{status}' = \textit{status} \\
\textbf{else}\ \ \textit{status}' = \textit{error};\ \ \textit{active}' = \textit{active};\ \ \textit{stack}' = \textit{tail}\,(\textit{tail}\,\textit{stack})
\end{array}
$$

Figure 5.29: The *UNION* Instruction

$$
\begin{array}{l}
\textit{XIsect} \\
\Delta[\textit{active}, \textit{stack}, \textit{status}]\textit{Selection} \\
\hline
es_1, es_2 : \mathrm{seq}\ \textit{Value};\ \ is_1, is_2 : \mathrm{seq}\ \textit{Intension} \\
\hline
\textit{instr} = \textit{ISECT};\ \ \textit{stack}\,2 = \textit{Set}(es_1, is_1);\ \ \textit{stack}\,1 = \textit{Set}(es_2, is_2) \\
\textbf{if}\ (es_1, es_2) \in \mathsf{dom}(_ \sqcap _) \\
\textbf{then}\ \textit{stack}' = \textit{Set}(es_1 \sqcap es_2, \textit{join}(\textit{Set}(es_1, \langle\rangle), \textit{Set}(\langle\rangle, is_2)) \frown \\
\qquad\qquad\qquad\qquad \textit{join}(\textit{Set}(es_2, \langle\rangle), \textit{Set}(\langle\rangle, is_1)) \frown \\
\qquad\qquad\qquad\qquad \textit{join}(\textit{Set}(\langle\rangle, is_1), \textit{Set}(\langle\rangle, is_1)) \\
\qquad\qquad\qquad\qquad)\ \cdot\!: \textit{tail}\,(\textit{tail}\,\textit{stack}) \\
\qquad \textit{active}' = \textit{active} :\!\cdot\ \textit{this};\ \ \textit{status}' = \textit{status} \\
\textbf{else}\ \ \textit{status}' = \textit{error};\ \ \textit{active}' = \textit{active};\ \ \textit{stack}' = \textit{tail}\,(\textit{tail}\,\textit{stack})
\end{array}
$$

$$
\begin{array}{l}
\textit{join} : \mathsf{ran}\,\textit{Set} \times \mathsf{ran}\,\textit{Set} \longrightarrow \mathrm{seq}\,\textit{Intension} \\
\hline
\textit{join}\ (\textit{Set}(es_1, is_1), \textit{Set}(es_2, is_2)) \Leftarrow \\
\qquad \textbf{if}\ \#es_1 + \#is_1 > 0 \wedge \#es_2 + \#is_2 > 0 \\
\qquad \textbf{then}\ \langle \lparen\!| \textit{varcnt} == 1, \textit{pat} == \textit{Var}\,0, \textit{env} == \langle\rangle, \\
\qquad\qquad \textit{params} == \langle \textit{Set}(es_1, is_1), \textit{Set}(es_2, is_2)\rangle, \\
\qquad\qquad \textit{constrs} == \\
\qquad\qquad\quad \langle\langle \textit{LOAD}(0), \textit{LOADPAR}(1), \textit{MEMBER}, \textit{SUCCESS}\rangle, \\
\qquad\qquad\quad\ \langle \textit{LOAD}(0), \textit{LOADPAR}(2), \textit{MEMBER}, \textit{SUCCESS}\rangle\rangle |\!\rparen \rangle \\
\qquad \textbf{else}\ \ \langle\rangle \\
\quad \textbf{where}\ es_1, es_2 : \mathrm{seq}\,\textit{Value};\ \ is_1, is_2 : \mathrm{seq}\,\textit{Intension}
\end{array}
$$

Figure 5.30: The *ISECT* Instruction

them (Figure 5.30). If the intersection of the extensional parts undefined (because it requires equality on values which cannot be computed) the thread stops in status *error*. The specification uses an auxiliary function $join(v1, v2)$ that creates an intension, implementing the μZ form $[\![x \mid x \in v_1 \wedge x \in v_2]\!]$ (cf. discussion of intersection, Section 5.1.1 (on page 98)). For realizing the constraints of this intension, we pass the set values v_1 and v_2 as intension parameters.

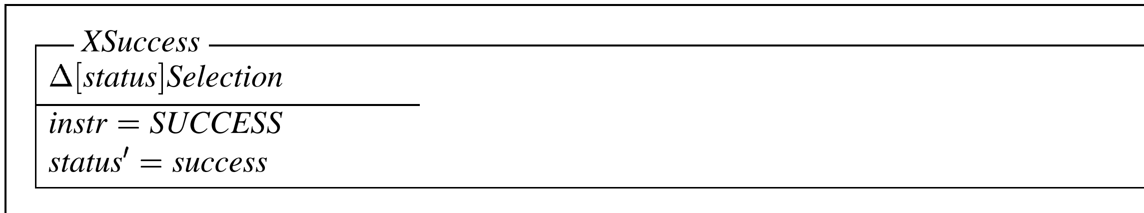


Figure 5.31: The *SUCCESS* Instruction

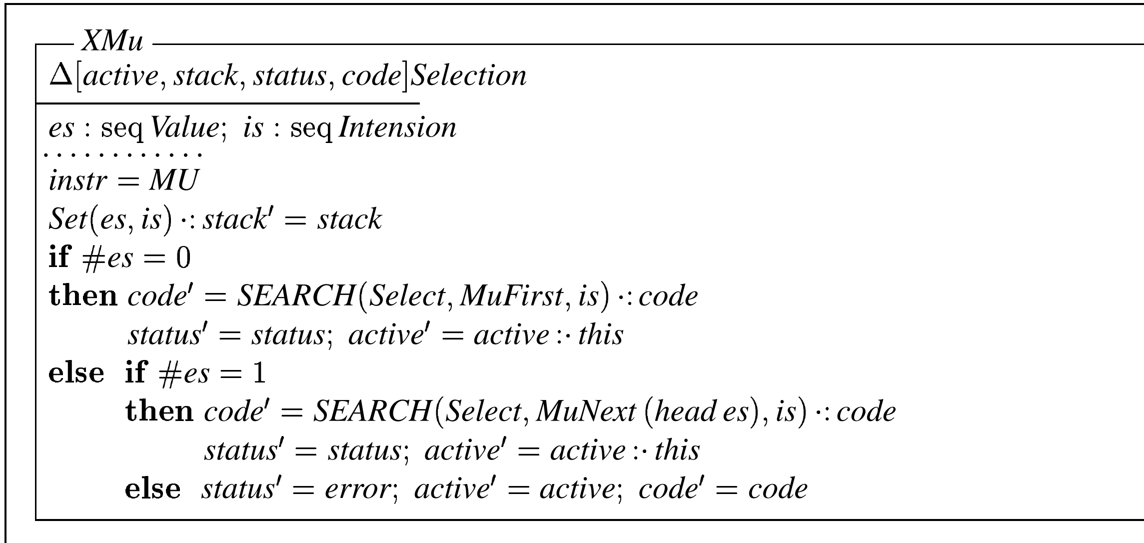


Figure 5.32: The *MU* Instruction

The *SUCCESS* Instruction

The *SUCCESS* instruction stops the running thread and puts into status *success* (Figure 5.31).

The *MU* Instruction

The *MU* instruction (Figure 5.32) delegates its work to the auxiliary instruction $SEARCH(state, mode, is)$ that implements subresolution for a sequence of intensions, *is*. *SEARCH* is started in state *Select* and one of two modes: *MuFirst* indicates that no candidate value is present for unique selection, whereas *MuNext cand* defines a candidate (the details are discussed later on). The candidate is initialized from a singleton extensional part of the set on top of the stack. If the extensional part contains more then one element, the *MU* instruction immediately switches to status *error*.

The *TEST* and *TESTNATIVE* Instructions

As the *MU* instruction, the *TEST* and *TESTNATIVE* instructions (Figure 5.33 (on the next page)) are based on the auxiliary instruction $SEARCH(state, mode, is)$ (defined in the next section). Both instructions are decided immediately, if the extensional part of the set on top of the stack is nonempty: the *TEST* instruction which checks for nonemptiness then succeeds, whereas the *TESTNATIVE* instruction fails.

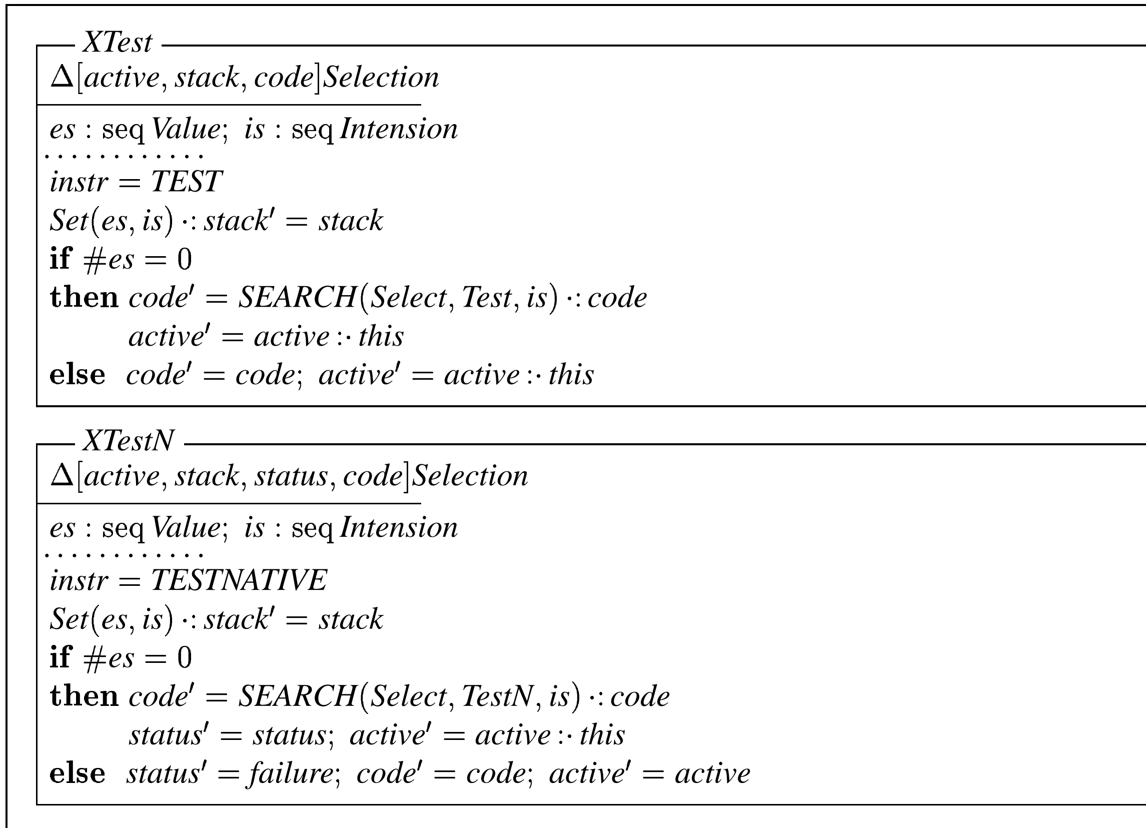


Figure 5.33: The *TEST* and *TESTNATIVE* Instructions

The *SEARCH* Auxiliary Instruction

The auxiliary instruction $SEARCH(state, mode, is)$ implements "encapsulated search" and is used to model the *MU* and *TEST*/*TESTNATIVE* instructions. The *state* is one of the following:

- *Select*: select the next intension from *is* and start a subgoal for its resolution. After selection, the executing thread suspends until the subgoal's execution has finished.
- *Resume*: the state in which the current thread continues when it is resumed since the subgoal's execution has finished. Actions are performed in dependency on the mode of the search: the search may be finished, backtracking may be invoked or the next intension may be tried.
- *Backtrack*: indicates that the current subgoal shall be backtracked

The *mode* describes which functionality is implemented by the search:

- *MuFirst*: search for a μ-value. No candidate is yet known.
- *MuNext cand*: search for a μ-value. A candidate is known, which must equal to each other possible candidate.
- *Test*: test for nonemptiness. The search stops and the initiating thread continues normally as soon as the first resolution is successful.

$$\begin{array}{l}
\mathit{XSearch} == \\
\quad \mathit{XSearchSelectMore} \lor \mathit{XSearchSelectEndMuFirst} \lor \\
\quad \mathit{XSearchSelectEndMuNext} \lor \mathit{XSearchSelectEndTest} \lor \\
\quad \mathit{XSearchSelectEndTestN} \lor \mathit{XSearchResumeSuccessMu} \lor \\
\quad \mathit{XSearchResumeSuccessTest} \lor \mathit{XSearchResumeSuccessTestN} \lor \\
\quad \mathit{XSearchResumeFailure} \lor \mathit{XSearchResumeError} \lor \\
\quad \mathit{XSearchBacktrack}
\end{array}$$

Figure 5.34: The *SEARCH* Auxiliary Instruction: Dispatching

XSearchSelectMore

$$\begin{array}{l}
\Delta[\mathit{child}, \mathit{ttab}, \mathit{gtab}, \mathit{active}, \mathit{code}]\mathit{Selection} \\
\hline
\mathit{mode} : \mathit{SearchMode};\ i : \mathit{Intension};\ \mathit{is} : \mathrm{seq}\,\mathit{Intension} \\
\cdots\cdots\cdots \\
\mathit{instr} = \mathit{SEARCH}(\mathit{Select}, \mathit{mode}, i \cdot\!: \mathit{is}) \\
\mathit{child}' = \#\mathit{gtab} + 1 \\
\mathit{ttab}' = \mathit{ttab} \cup \{t : \#\mathit{ttab} + 1 \mathrel{..} \#\mathit{ttab} + \#i.\mathit{constrs} \bullet \\
\qquad t \mapsto \langle\!|\mathit{parent} == \mathit{child}', \mathit{child} == 0, \mathit{prio} == 1, \\
\qquad\quad \mathit{status} == \mathit{running}, \mathit{env} == i.\mathit{env}, \\
\qquad\quad \mathit{params} == i.\mathit{params}, \mathit{stack} == \langle\rangle, \mathit{varshift} == 1, \\
\qquad\quad \mathit{code} == i.\mathit{constrs}(t - \#\mathit{ttab})|\!\rangle\} \\
\mathit{gtab}' = \mathit{gtab} \cup \{\mathit{child}' \mapsto \langle\!|\mathit{parentThread} == \mathit{this}, \mathit{pat} == i.\mathit{pat}, \\
\qquad \mathit{threads} == \mathsf{dom}\,\mathit{ttab}' \setminus \mathsf{dom}\,\mathit{ttab}, \\
\qquad \mathit{choices} == \langle\rangle, \\
\qquad \mathit{vtab} == \{v : 1 \mathrel{..} i.\mathit{varcnt} \bullet v \mapsto \mathit{Free}\,\emptyset\}|\!\rangle\} \\
\mathit{code}' = \mathit{SEARCH}(\mathit{Resume}, \mathit{mode}, \mathit{is}) \cdot\!: \mathit{code} \\
\mathit{active}' = \mathit{active} ^\frown \mathit{orderinx}(\mathsf{dom}\,\mathit{ttab}' \setminus \mathsf{dom}\,\mathit{ttab})
\end{array}$$

Figure 5.35: The *SEARCH* Auxiliary Instruction: Select, More Intensions

- *TestN*: test for emptiness. The search stops in status *failure* as soon as the first resolution is successful.

The $\mathit{SEARCH}(\mathit{state}, \mathit{mode}, \mathit{is})$ instruction is defined by several schemas that discriminate over *state*, *mode* and the sequence of intensions *is*. The dispatcher is defined in Figure 5.34.

Selection State. Figure 5.35 defines the selection state for the case that the intension sequence is nonempty, which is independent of the *mode* of the search. For the next intension *i*, a new subgoal is created and stored in the *child* field of the executing thread. The spawned threads are activated and the executing thread suspends, after preparing to continue in search state *Resume*.

Figure 5.36 (on the facing page) defines the selection state for the case that the intension sequence is empty. The action performed depends on the *mode*: in mode *MuFirst*, the initiating *MU* instruction produces an error (the set *MU* is executed on is empty). In mode *MuNext cand*, *cand* is the unique result of executing *MU*. In mode *Test*, the tested

XSearchSelectEndMuFirst

$\Delta[status]Selection$

$instr = SEARCH(Select, MuFirst, \langle\rangle)$
$status' = error$

XSearchSelectEndMuNext

$\Delta[active, stack]Selection$

$cand : Value$

$instr = SEARCH(Select, MuNext\ cand, \langle\rangle)$
$stack' = cand \cdot\!: stack$
$active' = active :\!\cdot\ this$

XSearchSelectEndTest

$\Delta[status]Selection$

$instr = SEARCH(Select, Test, \langle\rangle)$
$status' = failure$

XSearchSelectEndTestN

$\Delta[active]Selection$

$instr = SEARCH(Select, TestN, \langle\rangle)$
$active' = active :\!\cdot\ this$

Figure 5.36: The *SEARCH* Auxiliary Instruction: Select: No More Intensions

set is empty which lets the executing thread fail; in mode *TestN*, the test for emptiness has succeeded and the executing thread just continues.

Resume State. Figure 5.37 (on the next page) defines schemas for the *Resume* state in case the subgoal has been successfully resolved. We end here since a thread is automatically resumed by the general dispatch operation *PostDispatch* (Figure 5.16) when the threads of its subgoal have finished. The subgoal is successful if all its threads are in status *success*. The action depends on the mode of search: in case of *Test* or *TestN*, the search is finished, and the executing thread either continues or stops in status *failure*. In case of *MuFirst* or *MuNext cand*, the result of the subresolution is frozen and compared with a possible previous candidate for unique selection. The frozen value must not contain free variables; if it does so, or if a previous candidate does not equal to the frozen result, or if equality is undefined (since the compared values contain intensions), the executing thread stops in status *error*. Otherwise, the search switches either to status *Backtrack* if there are pending choices, or to status *Select* if there are no more choices, trying the next possible intension.

Figure 5.38 (on page 123) defines schemas for the *Resume* state in case the subgoal has failed or is in status *error*. In case of a failure, the next alternative is tried, either by backtracking or by selecting the next intension. An error status is just propagated to the executing thread.

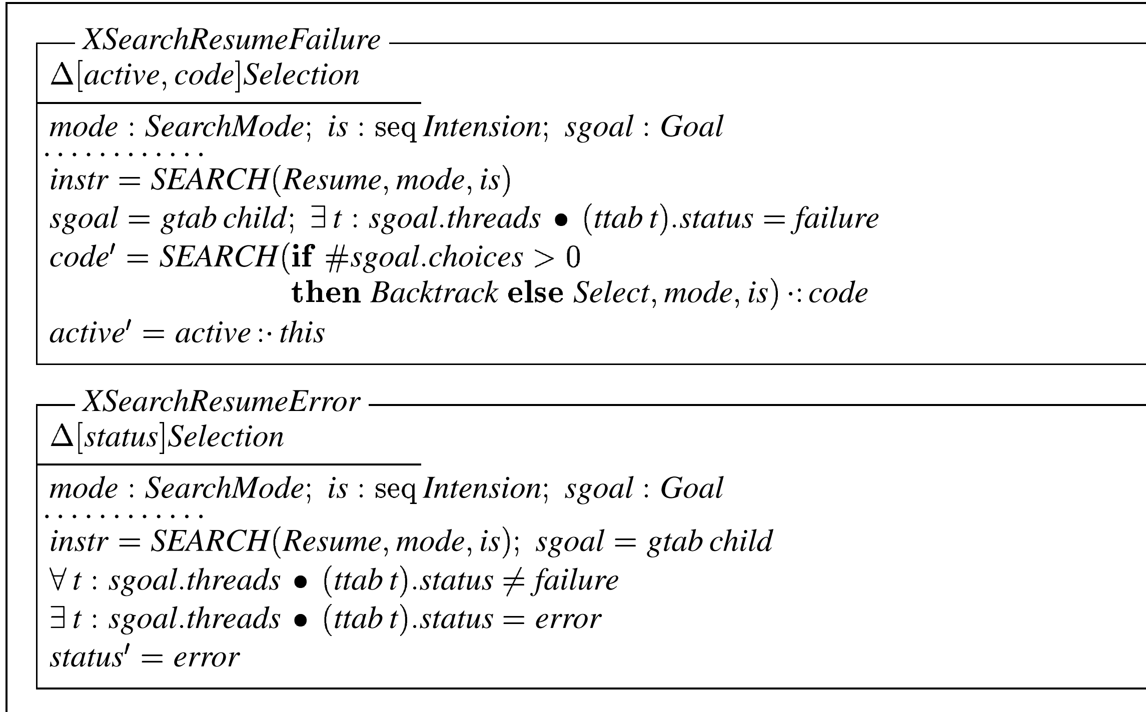
Backtrack State. Figure 5.39 (on page 123) defines the *Backtrack* status of a search. It can be asserted that the choice stack is not empty. In the subgoal, changes as described by the topmost choice entry are undone: dynamically spawned threads are removed, the choice itself is pruned and the variables are reset to be free. The threads which have dumped to the choice are restored. The restored threads are reactivated (they may run or not on backtrack) and the executing thread suspends after preparing to continue in state *Resume* when the backtracked subgoal has finished.

$$
\begin{array}{l}
\textit{XSearchResumeSuccessTest} \\
\Delta[\textit{active}]\textit{Selection} \\
\hline
\textit{is} : \mathrm{seq}\, \textit{Intension};\ \ \textit{sgoal} : \textit{Goal} \\
\cdots\cdots \\
\textit{instr} = \textit{SEARCH}(\textit{Resume}, \textit{Test}, \textit{is}) \\
\textit{sgoal} = \textit{gtab}\ \textit{child};\ \forall\, t : \textit{sgoal.threads} \bullet (\textit{ttab}\ t).\textit{status} = \textit{success} \\
\textit{active}' = \textit{active} :\cdot\ \textit{this};
\end{array}
$$

$$
\begin{array}{l}
\textit{XSearchResumeSuccessTestN} \\
\Delta[\textit{status}]\textit{Selection} \\
\hline
\textit{is} : \mathrm{seq}\, \textit{Intension};\ \ \textit{sgoal} : \textit{Goal} \\
\cdots\cdots \\
\textit{instr} = \textit{SEARCH}(\textit{Resume}, \textit{TestN}, \textit{is}) \\
\textit{sgoal} = \textit{gtab}\ \textit{child};\ \forall\, t : \textit{sgoal.threads} \bullet (\textit{ttab}\ t).\textit{status} = \textit{success} \\
\textit{status}' = \textit{failure}
\end{array}
$$

$$
\begin{array}{l}
\textit{XSearchResumeSuccessMu} \\
\Delta[\textit{active}, \textit{code}, \textit{status}]\textit{Selection} \\
\hline
\textit{mode} : \textit{SearchMode};\ \ \textit{is} : \mathrm{seq}\, \textit{Intension};\ \ \textit{sgoal} : \textit{Goal};\ \ \textit{cand}, \textit{res} : \textit{Value} \\
\cdots\cdots \\
\textit{instr} = \textit{SEARCH}(\textit{Resume}, \textit{mode}, \textit{is}) \\
\textit{mode} = \textit{MuFirst} \lor \textit{mode} = \textit{MuNext}\ \textit{cand} \\
\textit{sgoal} = \textit{gtab}\ \textit{child};\ \forall\, t : \textit{sgoal.threads} \bullet (\textit{ttab}\ t).\textit{status} = \textit{success} \\
\textit{res} = \textit{freeze}\ \textit{sgoal.vtab}\ \textit{sgoal.pat} \\
\mathbf{if}\ \textit{vars}\ \textit{res} = \varnothing \land \\
\quad (\textit{mode} = \textit{MuNext}\ \textit{cand} \Rightarrow (\textit{cand}, \textit{res}) \in \mathrm{dom}\ \textit{equal} \land \textit{equal}(\textit{cand}, \textit{res})) \\
\mathbf{then}\ \textit{code}' = \textit{SEARCH}(\mathbf{if}\ \#\textit{sgoal.choices} > 0 \\
\qquad\qquad\qquad\qquad \mathbf{then}\ \textit{Backtrack}\ \mathbf{else}\ \textit{Select}, \textit{MuNext}\ \textit{res}, \textit{is}) :\cdot\ \textit{code} \\
\qquad \textit{active}' = \textit{active} :\cdot\ \textit{this};\ \ \textit{status}' = \textit{status} \\
\mathbf{else}\ \ \textit{code}' = \textit{code};\ \ \textit{active}' = \textit{active};\ \ \textit{status}' = \textit{error}
\end{array}
$$

Figure 5.37: The *SEARCH* Auxiliary Instruction: Resume: Subgoal Success



XSearchResumeFailure

$\Delta[active, code]Selection$

$mode : SearchMode;\ is : \mathrm{seq}\,Intension;\ sgoal : Goal$

$instr = SEARCH(Resume, mode, is)$
$sgoal = gtab\,child;\ \exists\, t : sgoal.threads \bullet (ttab\,t).status = failure$
$code' = SEARCH(\mathbf{if}\ \#sgoal.choices > 0\ \mathbf{then}\ Backtrack\ \mathbf{else}\ Select, mode, is) \cdot\!: code$
$active' = active :\!\cdot\ this$

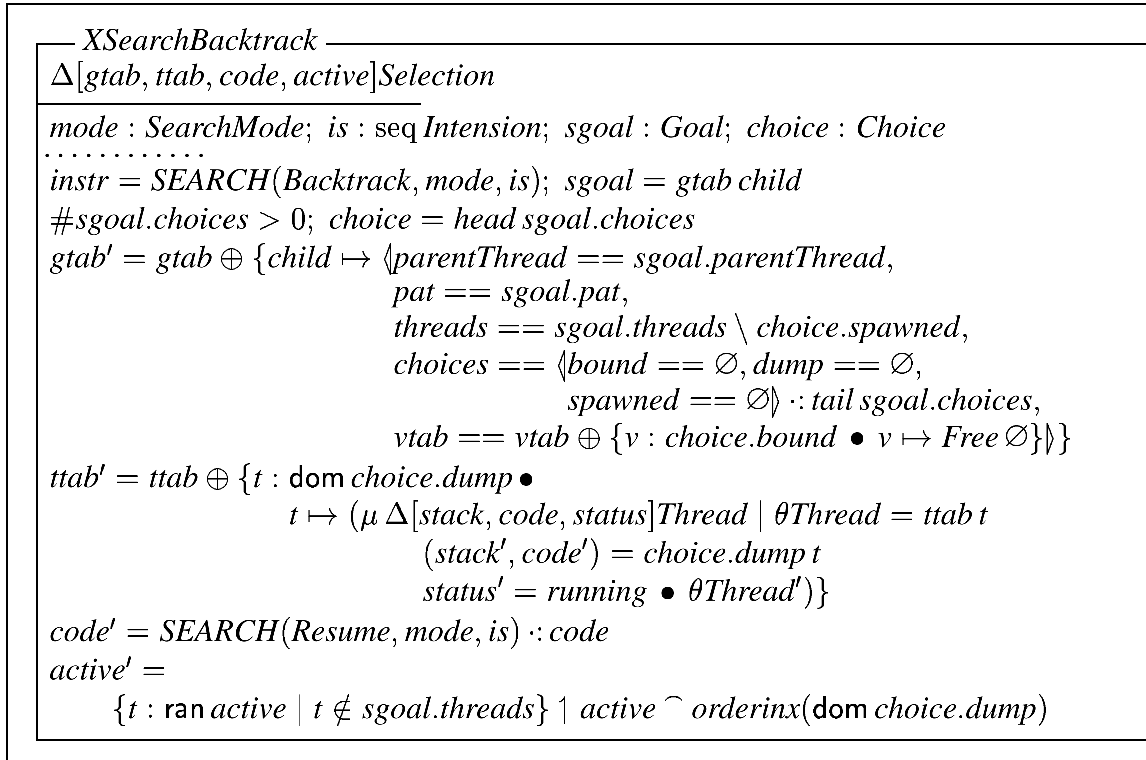
XSearchResumeError

$\Delta[status]Selection$

$mode : SearchMode;\ is : \mathrm{seq}\,Intension;\ sgoal : Goal$

$instr = SEARCH(Resume, mode, is);\ sgoal = gtab\,child$
$\forall\, t : sgoal.threads \bullet (ttab\,t).status \neq failure$
$\exists\, t : sgoal.threads \bullet (ttab\,t).status = error$
$status' = error$

Figure 5.38: The *SEARCH* Auxiliary Instruction: Resume: Subgoal Failure



XSearchBacktrack

$\Delta[gtab, ttab, code, active]Selection$

$mode : SearchMode;\ is : \mathrm{seq}\,Intension;\ sgoal : Goal;\ choice : Choice$

$instr = SEARCH(Backtrack, mode, is);\ sgoal = gtab\,child$
$\#sgoal.choices > 0;\ choice = head\,sgoal.choices$
$gtab' = gtab \oplus \{child \mapsto \langle\!| parentThread == sgoal.parentThread, pat == sgoal.pat, threads == sgoal.threads \setminus choice.spawned, choices == \langle\!| bound == \varnothing, dump == \varnothing, spawned == \varnothing |\!\rangle \cdot\!: tail\,sgoal.choices, vtab == vtab \oplus \{v : choice.bound \bullet v \mapsto Free\,\varnothing\} |\!\rangle\}$
$ttab' = ttab \oplus \{t : \mathrm{dom}\,choice.dump \bullet t \mapsto (\mu\,\Delta[stack, code, status]Thread \mid \theta Thread = ttab\,t\ (stack', code') = choice.dump\,t\ status' = running \bullet \theta Thread')\}$
$code' = SEARCH(Resume, mode, is) \cdot\!: code$
$active' = \{t : \mathrm{ran}\,active \mid t \notin sgoal.threads\} \upharpoonright active \frown orderinx(\mathrm{dom}\,choice.dump)$

Figure 5.39: The *SEARCH* Auxiliary Instruction: Backtracking

$$ENV == (VAR \pfun VarInx) \times (VAR \pfun \mathbb{N}) \times \mathbb{F}\ VAR$$

$$\begin{array}{l}
enter : ENV \times \mathbb{F}\ VAR \longrightarrow ENV \\
index == \lambda\,\gamma : ENV;\ x : VAR \bullet \gamma.1\,x \\
nest == \lambda\,\gamma : ENV;\ x : VAR \bullet \gamma.2\,x \\
mkflex == \lambda\,\gamma : ENV;\ vars : \mathbb{F}\ VAR \bullet (\gamma.1, \gamma.2, \gamma.3 \cup vars) \\
flex == \lambda\,\gamma : ENV \bullet \gamma.3 \\
\hline
enter\ (\gamma, vars) \Leftarrow \\
\quad \mathbf{let}\ order == \mathsf{varorder}\ vars \bullet \\
\quad (\gamma.1 \oplus \{i : \mathsf{dom}\ order \bullet order\ i \mapsto i - 1\}, \\
\quad\ \lambda\,x : VAR \bullet\ \mathbf{if}\ x \in vars\ \mathbf{then}\ 0\ \mathbf{else}\ nest(\gamma, x) + 1, \\
\quad\ flex\,\gamma \setminus vars) \\
\ \ \mathbf{where}\ \gamma : ENV;\ vars : \mathbb{F}\ VAR
\end{array}$$

Figure 5.40: Compilation Environment

5.3 Compilation

In this section, the compilation of μZ to ZAM instructions is described. Compilation works on expressions normalized as described in Section 4.2 (on page 74) in Chapter 4. A top-level expression, constituting a μZ “program”, may, for example, be given as a property with free variables, such as $x \in e$, where x describes the output of the program. Compiling $x \in e$ with an initial environment declaring x yields a sequence of code instructions which can be executed as the constraint of an intension of the ZAM. Backtracking over the goal for this intension yields the possible bindings for x.

5.3.1 Environment

The compilation is based on an environment holding information about variable names in the context of a compiled expression. An environment, $\gamma \in ENV$, is a triple $(indices, nests, flex)$, where *indices* is a mapping from variables to their associated indices, *nests* a mapping from variables to their scoping nest and *flex* a set of variables which are considered as “flexible”. For flexible variables, in contrast to nonflexible, it is not require to emit a *WAIT* instruction before they can be accessed. The specification of environments is given in Figure 5.40.

The $enter(\gamma, vars)$ function extends the environment for entering a scope which declares the variables *vars*. Variables from outer scopes are shadowed; for those which are still visible the scoping nest is incremented by one. The function $index(\gamma, x)$ retrieves the index of the variable x in γ, the function $nest(\gamma, x)$ the scoping nest of x. The function $mkflex(\gamma, vars)$ adds the given variables to the *flex* set, the function $flex\,\gamma$ retrieves the *flex* set.

In the definition of *enter*, we use the function varorder that makes a sequence of some fixed but arbitrary order from a finite set of variables. Note that variable indices in the ZAM start with 0 (whereas Z’s sequences are indexed starting from 1). From the *flex* set, we remove the newly entered variables, since any variables with the same name in this set are shadowed, and the flexible information becomes invalid for them.

$$\begin{array}{|l} \mathcal{C}_{\text{BAS}} : ENV \longrightarrow EXP_B \longrightarrow Code \\ \mathcal{C}_{\text{DIS}} : ENV \longrightarrow EXP_D \longrightarrow Code \\ \mathcal{C}_{\text{CON}} : ENV \longrightarrow EXP_C \longrightarrow Code \\ \mathcal{C}_{\text{LIT}} : ENV \longrightarrow EXP_L \longrightarrow Code \\ \mathcal{C}_{\text{PRO}} : ENV \longrightarrow EXP_A \longrightarrow Code \end{array}$$

Figure 5.41: Declaration of Compilation Functions

$$\begin{array}{lll} \mathcal{C}_{\text{BAS}}\,\gamma\;(\rho(\overline{b})) & \Leftarrow & {}^\frown/(\mathcal{C}_{\text{BAS}}\,\gamma \circ \overline{b}) :\cdot MKTERM(\rho, \#\overline{b}) \\ \quad\mid\quad d & \Leftarrow & \mathcal{C}_{\text{DIS}}\,\gamma\, d \\ \quad\mid\quad c & \Leftarrow & \mathcal{C}_{\text{CON}}\,\gamma\, c \\ \quad\mid\quad l & \Leftarrow & \mathcal{C}_{\text{LIT}}\,\gamma\, l \\ \multicolumn{3}{l}{\quad \textbf{where}\ \gamma : ENV;\ \overline{b} : \text{seq}\, EXP_B;\ \rho : CONS;\ d : EXP_D;\ c : EXP_C;\ l : EXP_L} \end{array}$$

Figure 5.42: Compiling General Expressions

5.3.2 Compilation Functions

For each level of the disjunctive normal form (described in Figure 4.3 (on page 74)), a compilation function is provided: general expressions, disjunctions, conjunctions, literals and properties. These functions constitute a mutual recursive system, which will be specified in the subsections below. The declaration of the compilation functions is given in Figure 5.41.

Compiling General Expressions

The function $\mathcal{C}_{\text{BAS}}\,\gamma b$ takes any expression in normal form and compiles it to a sequence of instructions (Figure 5.42). For constructor application, $\rho(\overline{b})$, code is generated which pushes the arguments, and the *MKTERM* instruction is appended to this code. For other expressions, the appropriate specialized compilation function is called.

Compiling Disjunctions and Conjunctions

The compilation of disjunction (union) and conjunction (intersection) is defined in Figure 5.43 (on the next page). For unions, $\bigcup \overline{c}$, *UNION* instructions are folded into the instructions generated from the operands $\overline{c}$. A simple optimization can prevent the creation of the empty set as a neutral element, which we have omitted. For the compilation of intersection, $\bigcap \overline{l}$, *ISECT* instructions are folded into the instructions generated from the literal operands. The sequence of conjuncted literals can be assumed to be nonempty by the normal form of expressions.

Compiling Literals

The compilation of literals (and atoms, which we subsume here) is defined in Figure 5.44 (on the following page). The following comments:

$$\begin{array}{l}
\mathcal{C}_{\text{DIS}}\,\gamma\,(\bigcup \overline{c}) \Leftarrow (\lambda\, c : \mathit{EXP}_C;\ \mathit{code} : \mathit{Code} \bullet \mathit{code} \frown \mathcal{C}_{\text{CON}}\,\gamma\, c :\cdot \mathit{UNION}, \\
\qquad\qquad \langle \mathit{MKEMPTY} \rangle) \setminus \overline{c} \\
\quad \textbf{where } \gamma : \mathit{ENV};\ \overline{c} : \mathrm{seq}\, \mathit{EXP}_C \\
\mathcal{C}_{\text{CON}}\,\gamma\,(\bigcap \bar{l}) \Leftarrow (\lambda\, l : \mathit{EXP}_L;\ \mathit{code} : \mathit{Code} \bullet \mathit{code} \frown \mathcal{C}_{\text{LIT}}\,\gamma\, l :\cdot \mathit{ISECT}, \\
\qquad\qquad \mathcal{C}_{\text{LIT}}\,\gamma\,(\mathit{head}\,\bar{l})) \setminus \mathit{tail}\,\bar{l} \\
\quad \textbf{where } \gamma : \mathit{ENV};\ \bar{l} : \mathrm{seq}\, \mathit{EXP}_L \mid \#\bar{l} > 0
\end{array}$$

Figure 5.43: Compiling Disjunctions and Conjunctions

$$\begin{array}{lll}
\mathcal{C}_{\text{LIT}}\,\gamma\,(x) & \Leftarrow & \textbf{if } \mathit{nest}(\gamma, x) = 0 \\
& & \quad \textbf{then if } x \notin \mathit{flex}\,\gamma \\
& & \qquad \textbf{then } \langle \mathit{WAIT}(\mathit{index}(\gamma, x)), \mathit{LOAD}(\mathit{index}(\gamma, x)) \rangle \\
& & \qquad \textbf{else } \langle \mathit{LOAD}(\mathit{index}(\gamma, x)) \rangle \\
& & \quad \textbf{else } \langle \mathit{LOADENV}(\mathit{nest}(\gamma, x), \mathit{index}(\gamma, x)) \rangle \\
\quad \mid \ (\{b\}) & \Leftarrow & \mathcal{C}_{\text{BAS}}\,\gamma\, b :\cdot \mathit{MKSINGLE} \\
\quad \mid \ (\mu\, d) & \Leftarrow & \mathcal{C}_{\text{DIS}}\,\gamma\, d :\cdot \mathit{MU} \\
\quad \mid \ (\sim a) & \Leftarrow & \mathcal{C}_{\text{LIT}}\,\gamma\, a :\cdot \mathit{MKVAR}(0) :\cdot \\
& & \mathit{MKINTEN}(1, 1, \langle \langle \mathit{WAIT}(0), \mathit{LOAD}(0), \\
& & \qquad\qquad \mathit{MKSINGLE}, \mathit{LOADPAR}(1), \\
& & \qquad\qquad \mathit{ISECT}, \mathit{TESTNATIVE}, \mathit{SUCCESS} \rangle \rangle) \\
\quad \mid \ (c[p_1 \mapsto p_2]) & \Leftarrow & \textbf{let } \mathit{vars} == \mathsf{vars}\, p_1 \cup \mathsf{vars}\, p_2 \bullet \\
& & \textbf{let } \gamma' == \mathit{mkflex}(\mathit{enter}(\gamma, \mathit{vars}), \mathit{vars}) \bullet \\
& & \mathcal{C}_{\text{CON}}\,\gamma\, c \frown \mathcal{C}_{\text{BAS}}\,\gamma'\, p_2 :\cdot \\
& & \mathit{MKINTEN}(1, \#\mathit{vars}, \\
& & \qquad \langle \mathcal{C}_{\text{BAS}}\,\gamma'\, p_1 \frown \\
& & \qquad\quad \langle \mathit{LOADPAR}(1), \mathit{MEMBER}, \mathit{SUCCESS} \rangle \rangle) \\
\quad \mid \ (\{p \mid \phi\}) & \Leftarrow & \textbf{let } w == \{x : \mathsf{vars}\, \phi \setminus (\mathsf{vars}\, p \cup \mathit{flex}\,\gamma) \mid \mathit{nest}(\gamma, x) = 0\} \bullet \\
& & \textbf{let } \gamma' == \mathit{enter}(\gamma, \mathsf{vars}\ p) \bullet \\
& & (\lambda\, x : w \bullet \mathit{WAIT}(\mathit{index}(\gamma, x))) \circ \mathsf{varorder}\ w \frown \\
& & \mathcal{C}_{\text{BAS}}\,(\mathit{mkflex}(\gamma', \mathsf{vars}\, p))\, p :\cdot \\
& & \mathit{MKINTEN}(0, \#(\mathsf{vars}\, p), \langle \mathcal{C}_{\text{PRO}}\ \gamma'\ \phi \rangle) \\
\quad \mid \ (\mathbf{fix}\, p \lhd b) & \Leftarrow & \textbf{let } w == \{x : \mathsf{vars}\, b \setminus (\mathsf{vars}\, p \cup \mathit{flex}\,\gamma) \mid \mathit{nest}(\gamma, x) = 0\} \bullet \\
& & \textbf{let } \gamma' == \mathit{mkflex}(\mathit{enter}(\gamma, \mathsf{vars}\ p), \mathsf{vars}\ p) \bullet \\
& & (\lambda\, x : w \bullet \mathit{WAIT}(\mathit{index}(\gamma, x))) \circ \mathsf{varorder}\ w \frown \\
& & \mathcal{C}_{\text{BAS}}\,\gamma'\, p :\cdot \\
& & \mathit{MKINTEN}(0, \#(\mathsf{vars}\ p), \langle \mathcal{C}_{\text{PRO}}\ \gamma'\ (p = b) \rangle) :\cdot \mathit{MU} \\
\multicolumn{3}{l}{\quad \textbf{where } \gamma : \mathit{ENV};\ x : \mathit{VAR};\ b : \mathit{EXP}_B;\ d : \mathit{EXP}_D;\ a : \mathit{EXP}_A} \\
\multicolumn{3}{l}{\qquad\qquad c : \mathit{EXP}_C;\ \phi : \mathit{EXP}_P;\ p, p_1, p_2 : \mathit{PAT}}
\end{array}$$

Figure 5.44: Compiling Literals

- For a *variable*, x, we distinguish whether it is local (its lexical nesting is 0) or from the context. For local variables, we generate a *LOAD* instruction, prepending a *WAIT* instruction if the variable is nonflexible. For a context variable, we generate a *LOADENV* instruction.

$$
\begin{array}{lll}
\mathcal{C}_{\text{PRO}}\,\gamma\;(p_1 = p_2) & \Leftarrow & \mathcal{C}_{\text{BAS}}\;(mkflex(\gamma, \mathsf{vars}\,p_1))\;p_1 {}^\frown \\
 & & \mathcal{C}_{\text{BAS}}\;(mkflex(\gamma, \mathsf{vars}\,p_2))\;p_2 :\cdot UNIFY :\cdot SUCCESS \\
\mid \quad (x = b) & \Leftarrow & \mathcal{C}_{\text{BAS}}\,\gamma\,b :\cdot STORE(index(\gamma, x)) :\cdot SUCCESS \\
\mid \quad (p = b) & \Leftarrow & \mathcal{C}_{\text{BAS}}\;(mkflex(\gamma, \mathsf{vars}\,p))\;p {}^\frown \mathcal{C}_{\text{BAS}}\,\gamma\,b :\cdot UNIFY :\cdot SUCCESS \\
\mid \quad (p \in d) & \Leftarrow & \mathcal{C}_{\text{BAS}}\;(mkflex(\gamma, \mathsf{vars}\,p))\;p {}^\frown \mathcal{C}_{\text{DIS}}\,\gamma\,d :\cdot MEMBER :\cdot SUCCESS \\
\mid \quad (?_1 c) & \Leftarrow & \mathcal{C}_{\text{CON}}\,\gamma\,c :\cdot TEST :\cdot SUCCESS \\
\mid \quad (?_0 c) & \Leftarrow & \mathcal{C}_{\text{CON}}\,\gamma\,c :\cdot TESTNATIVE :\cdot SUCCESS \\
\multicolumn{3}{l}{\quad \textbf{where } \gamma : ENV;\; p, p_1, p_2 : PAT;\; x : VAR;\; d : EXP_D;\; c : EXP_C;\; b : EXP_B}
\end{array}
$$

Figure 5.45: Compiling Properties

- For a singleton-set construction, $\{b\}$, code is generated to strictly create the element on the stack, and the *MKSINGLE* instruction is appended.
- For a singleton-set selection, $\mu\, d$, code is generated which pushes the value of d to the stack, and the *MU* instruction is appended.
- For a set complement, $\sim a$, code is generated that realizes the μZ intension $[\![x \mid ?_0(\{x\} \cap v)]\!]$, where v represents the evaluated value of a, which is passed as an intension parameter.
- For a translation, $c[p_1 \mapsto p_2]$, code is generated that realizes the μZ intension $[\![p_2 \mid p_1 \in v]\!]$, where v represents the evaluated value of c, which is passed as an intension parameter.
- For a schema, $\{p \mid \phi\}$, code for creating an intension is generated. Before the creation of the intension, code is inserted which waits for all free variables of the schema which are in the current scope and which are not flexible. This ensures that the newly created intension does not refer to free variables from the environment. It is sufficient to wait for the variables of the current scope (from the viewpoint of the code creating the intension), since, inductively, the variables of the outer scopes *are* bound. Flexible variables are excluded from this treatment.
- A fixed-point, $\mathbf{fix}\, p \lhd b$, is compiled as the form $\mu\,\{p \mid p = b\}$, with the difference that during compilation of $p = b$ we add the variables from p to the set of flexible variables. This allows for the construction of cyclic intensions. Note that we need the temporarily constructed intension that is immediately deconstructed by the *MU* operator in order to get an environment where inner intensions created for b can depend on.

Compiling Properties

The compilation of properties is defined in Figure 5.45. The following comments:

- A pattern equality, $p_1 = p_2$, is mapped to a *UNIFY* instruction where the patterns are compiled such that all variables in them are flexible. Note that, by normalization, the variables appearing in the patterns must be directly bounded by the enclosing schema.

- The next three cases are variants of pattern membership, $p \in c$. For the case of a direct assignment to a variable, $x = b$ (a shortcut for $x \in \{b\}$) we use the *STORE* instruction as an optimization. For the case of an assignment to a pattern, $p = b$, the *UNIFY* instruction is used. In compiling the pattern p, the variables of the pattern are made flexible. Finally, the general case, $p \in c$ is compiled to the *MEMBER* instruction.

- The remaining two cases handle the test for emptiness or nonemptiness of the set c, and are mapped to the *TEST* resp. *TESTNATIVE* instructions.

5.4 Implementation

A prototypical implementation of the ZAM in C++ has been realized, which is discussed in this section. The implementation differs from the specified model in the following respects:

- In order to make dumping of thread states efficient, we use a *persistent single assignment register* machine instead of a stack machine. Since a single assignment register can be never overwritten, it is sufficient to just dump a program counter – a thread backtracking to an execution point always finds a valid temporary state. The usage of a register machine has some influences on the instruction set, as will be seen below.
- Once having the concept of persistent single assignment registers, we may use different techniques for passing parameters to the *MKINTEN* instruction and other instructions. Instead of *copying* the parameters, we just pass a *reference* to a persistent register area (a pointer to a C++ array of values)[4].
- Since we have no problems with creating cyclic data structures in C++, the construction of environments for intensions as in the ZAM's Z specification can be completely avoided, using intension parameters instead. Since the intension parameters are passed by a reference to a register area, it is particularly easy to back-patch fixed-points.

Below, we sketch the data model of the ZAM in C++ and the instruction set of the register version. Some first benchmarks for the prototypical implementation are also given.

5.4.1 Data Model

Figure 5.46 (on the following page) shows the basic types used in C++ to represent ZAM values. Here, we also sketch the realization of native values, used for implementing numbers and other builtin types. Native values are characterized by a virtual method `compare` which implements a total order on them (in the C++ implementation, ordering instead of extensional equality is used).

A set value, `SetData(extens,intens)`, consists of two sorted sequences, represented as STL vectors. C++'s standard template library provides us with algorithms for set union and intersection on sorted sequences.

As discussed, an intension in the C++ implementation does not hold an environment but only an array of parameters. For each free variable of the intension's constraints, the `params` field must contain an entry where a copy of this variable's value is stored. The resulting representation of environment variables is very similar to closures, as yielded by implementing nested functions by λ-lifting in the implementation of functional languages.

A constraint is represented by an abstract class, which provides a method to *spawn* a *thread* executing this constraint. This abstraction allows the easy integration of specialized constraint implementations (for example, the intensions dynamically created for

[4]Pointers to registers seem to be an abuse of notions. We justify this by that the instruction set of the C++ ZAM actually corresponds to a register transfer language. Moreover, storing registers in reentrant memory is also a characteristic feature of RISC like processor architectures.

```
typedef ValueData  const * Value;   typedef Inten-
Data const * Inten;
typedef ConstrData const * Constr;
enum Kind           { TERM, VAR, SET, NATIVE };
struct ValueData    { Kind kind; };
struct TermData     : ValueData { Constr cons; Value args[]; };
struct VarData      : ValueData { int index; };
struct SetData      : ValueData { vector<Value> extens;
                                  vector<Inten> intens; };
struct NativeData   : ValueData { virtual int com-
pare(...) = 0; ... };
struct IntenData    { int varcount; Value pat; Value params[];
                      Constr constrs[]; };
struct ConstrData   { virtual Thread spawn(..., int varshift) = 0; }
```

Figure 5.46: Value Types of the ZAM in C++

```
typedef ThreadData * Thread; typedef GoalData * Goal;
struct ThreadData   { enum Status { RUNNING, SUCCESS, FAIL-
URE,
                                   ER-
ROR, MWAIT, TWAIT } status;
                      Instr * code; int varshift; int dumpDepth;
                      enum Priority { LO, HI } prio;
                      Value params[]; Value registers[];
                      union { <<data for MWAIT and TWAIT>> }; };
struct VarInfo      { Value binding; vector<Thread> wait-
ing; };
struct ChoiceInfo   { Thread initiator; <<data for alterna-
tive>>;
                      vector<int> bound; vec-
tor<ThreaDump> dumps;
                      vector<Thread> spawned; };
struct ThreadDump   { Thread dumper; Instr* code; };
struct GoalData     { Thread parent; vector<Thread> threads;
                      Value pat;
                      vector<VarInfo> vtab;
                      vector<ChoiceInfo> choices; };
```

Figure 5.47: Control Types of the ZAM in C++

realizing set intersection (cf. Figure 5.30 (on page 117)). The spawn method is passed the variable shift of the intension's instance.

The representation of threads and goals in the C++ implementation is given in Figure 5.47. It is very similar to the specification in Z. There are the following differences:

MKEMPTY			$\to r\|v$	*MOVE*	$r\|v\|p$		$\to r\|v$
MKSINGLE	$r\|v\|p$		$\to r\|v$	*UNIFY*	$r\|v\|p,$	$r\|v\|p$	
MKINTEN	s	$r[]$	$\to r\|v$	*MEMBER*	$r\|v\|p,$	$r\|v\|p$	
FIXINTEN	$r\|v$			*MU*	$r\|v\|p$		$\to r\|v$
MKTERM	$c,$	$r[]$	$\to r\|v$	*TEST*	$r\|v\|p$		
UNION	$r\|v\|p,$	$r\|v\|p$	$\to r\|v$	*TESTNATIVE*	$r\|v\|p$		
ISECT	$r\|v\|p,$	$r\|v\|p$	$\to r\|v$	*SUCCESS*			
WAIT	v						

Figure 5.48: Register Transfer Instruction Set of the ZAM

- For reasons of efficiency, auxiliary instructions such as *TRYNEXT*(p, es, is) which carry a data state can not be used. We thus have additional data in choices, representing the state contained in *TRYNEXT*. Furthermore, threads have additional states, *MWAIT* and *TWAIT*, and corresponding data components for execution the *MU* and *TEST*/*TESTNATIVE* instructions; this data is only valid if a thread is in one of these states.

- The need of a thread to dump its state to the active choice is encoded in a more efficient way than in the specification. The field `ThreadData::dumpDepth` contains the maximal choice depth a thread has been dumped to, and monotonically increases during the thread's progress. The test for the need to dump is coded as `goal->choiceTable.size() > thread->dumpDepth`.

As discussed, it is sufficient to just restore the program counter to backtrack threads since the ZAM uses a single assignment register model for storing temporaries (field `ThreadData::registers`):. Thus a thread can be restored to an arbitrary execution point, always finding a valid temporary state at this point[5].

5.4.2 Instruction Set

The ZAM's C++ implementation uses a register transfer instruction language. The instructions are summarized in Figure 5.48. The operands of instructions are the followings:

- r: the index of a register. This addresses the field `ThreadData::registers` of the executing thread.

- v: the index of a variable. This addresses the `GoalData::vtab` field of the goal, the executing thread belongs to. If the variable is bound, its binding is denoted, otherwise a `VarData` term holding the variable's index. The executing thread adds the `ThreadData::varshift` value to the index, which determines the base of the variable frame of on associated instantiation. If a variable is the destination of an instruction, then the value of the instruction's result is unified with the addressed variable's value.

[5] By construction, a thread can never dump in state `MWAIT` or `TWAIT` – since in these states it does not progress. Thus the special temporary control data stored in `ThreadData` for these states needs not to be dumped.

- p: the index of a parameter. This addresses the field `ThreadData::params` of the executing thread.

- $r|v|p$: alternatively, a register, variable or parameter index. Similar as the Java Virtual Machine, the ZAM provides in fact a set of specialized instructions for the different addressing modes, which avoid dynamic encoding at runtime, but from which is abstracted here. Around a hundred concrete instructions are derived from the combinations of addressing modes in the abstract instructions in Figure 5.48 (on the preceding page).

- $r[]$: a reference to a register region. Provided is the starting index of a consecutive region of registers.

- s: the index of an intension. This points to a global table of intensions belonging to the executed unit (the "constant pool" in notions of the JVM).

- c: the index of a constructor. This points to a global table of constructors belonging to the executed unit.

The following remarks on the instructions:

- The *MKEMPTY* instruction stores the empty set in the destination operand, the *MKSINGLE* instruction a singleton set containing the value in the source operand. The value in the source operand must not contain free variables, otherwise the behavior is undefined (the compiled code ensures this with the *WAIT* instruction).

- The *MKINTEN* instruction creates a new intension. The parameters of the intension are found in the register region starting at $r[]$. In the intension's field `IntenData::params`, just a pointer to the register region is stored. The *FIXINTEN* instruction commits the parameters of the addressed intension to be completely initialized. This instruction may make a copy of `IntenData::params`; however, by the assertion of "single assignments" to registers, the *FIXINTEN* instruction may as well do just nothing. (Making a copy avoids space leaks, since the live-time of the created intension may exceed the live-time of the creating thread; in this case, if no copy is made, the entire register array of the thread cannot be reclaimed by the garbage collector unless the intension is.)

- The *MKTERM* instruction creates a new constructor term. The arguments of the constructor are passed in the register region $r[]$. Again, a copy of this region is not required, though omitting it can create space leaks. (To copy or not, or to let this be determined by compiler optimizations, is an open question.)

- The *UNION* and *ISECT* instruction implement set union and intersection, as described in the specification.

- The *WAIT* instruction lets the executing thread suspend until the addressed variable (and all the variables it indirectly refers to by a possible binding) are bound.

- The *MOVE* instruction transfers the source to the destination. If the destination is a variable and is bound, this is similar to unification. The *UNIFY* instruction performs a unification of its operands. If unification fails, the executing thread fails.

- The *MEMBER* instruction tests whether the first operand is member of the second set operand, as described in the specification.

- The *MU*, *TEST* and *TESTNATIVE* instructions create subgoals for resolving their operand, as described in the specification.

- The *SUCCESS* instruction terminates the executing thread. If all threads of a goal have been successfully terminated, then an associated parent thread of the goal is resumed (which is present if the goal is a subgoal).

5.4.3 Memory Management

A Boehm-style conservative garbage collector [Boehm, 1993] is employed for memory retrieval. The collector needs to handle inferior pointers (references "into" a memory cell), because pointers to register regions may be stored in values, but also since the standard template library of C++ is involved, which may use local allocation strategies for containers. Nevertheless, the implementation of the garbage collector is compact (300 lines of code) and portable: this becomes possible since garbage collection can be invoked synchronously inbetween the execution steps of threads, where no hardware stack exists, and the only root to consider is the ZAM's configuration which points to the topmost goal.

5.4.4 Performance

Since the compiler from μZ to ZAM's register transfer code is not yet ready, performance measurements can be only very rudimentary. As a first guess, an "assembler" implementation of the list concatenation function *app* has been used:

$$\begin{aligned} \mathbf{fix}\, app \lhd\; & \{(\langle\rangle, ys, zs) \mid zs = ys\} \cup \\ & \{(x :: xs, ys, zs) \setminus t \mid (xs, ys, t) \in app \cap zs = x :: t\} \end{aligned}$$

All solutions for $app(xs, ys, c)$, where c is a constant list of length 128, have been queried, in a loop executed thousand times (such that garbage collection needs to be invoked). The current implementation requires 16.4 seconds on a Pentium II/400, hence 16.4ms for one evaluation of *app*, backtracking over all possible 128 partitions. Around 2.9 million instruction steps are executed, which amounts to 177000 steps per second.

As a further test, a recursive definition of the *fac* function has been coded:

$$\begin{aligned} \mathbf{fix}\, fac \lhd\; & \{(x, y) \mid x = 0 \cap y = 1\} \cup \\ & \{(x, y) \mid x > 1 \cap y = x * \mu\,((fac \cap \{x - 1\}[x \mapsto (x, _)])[(_, y) \mapsto y])\} \end{aligned}$$

For numbers and arithmetics, a native implementation is used (the ZAM has also some instructions for natives that have not been specified). Executing *fac* 8 thousand times is done in 1.6 seconds, requiring 303000 steps: thus one recursive function application as

implemented with the μ-operator requires around 0.2ms. In contrast, a highly efficient implementation of a functional language such as that of Opal [Schulte and Grieskamp, 1992] requires around 0.0014ms per recursive call – 142 times faster. This is not surprising, since the code the Opal compiler generates for *fac* is identical regarding efficiency to that of an according recursive C function – whereas function application via the μ-operator deals with relations. Nevertheless, more effort has to be invested to get function application in the ZAM more efficient for special cases.

Taking into account that the ZAM is a virtual byte-code machine, these results are encouraging, though real benchmarking is required to get a clearer picture, and comparisons with the performance of other implementations of functional logic languages have to be made.

5.5 Discussion

We defined the abstract machine ZAM, supplied a comprehensive specification of its operation (which also serves as a case study for the Z specifications we wish to execute by our approach), provided a compilation of normalized μZ into the machine's instruction set and outlined the implementation of the machine in C++. Related work is discussed below.

There are a wide variety of abstract machines for functional logic languages. Even five years ago, Hanus [1994] mentioned fourteen different designs, which he considered as the "most important" ones. In general, these machines can be divided into those that have evolved from machines for logic languages, those from functional languages and those that use techniques for concurrent constraint resolution.

Extending Logic Implementation Techniques

Machines based on implementation techniques for logic languages are often extensions of Warren's Abstract Machine [Warren, 1983; Ait-Kaci, 1991]. The WAM provides an efficient implementation of Prolog, which breaks down unification and backtracking to a level of well-designed, specialized low-level instructions. The ZAM does not attempt to do this: its instructions are complex; unification, for example, is contained as "atomic" functionality.

Some extensions of the WAM integrate narrowing (outermost or innermost) into the machine, for example by adding the possibility of term replacement on heap values, as in the A-WAM [Hanus, 1990]. Other extensions such as the K-WAM [Bosco et al., 1989] are based on translating functions into relations, with the result that deterministic program parts are executed using the full indeterministic machinery. This is, in fact, close to the ZAM, where functions are already mapped to relations at the calculus level of μZ (though the ZAM has the advantage of choice point scheduling, which Prolog implementations can only hardly achieve because of the sequential execution model of Prolog.) Conceptually, both approaches are inadequate for implementing μZ, since *concurrent* constraint resolution is essential for our application. Furthermore, these techniques do not treat *higher-orderness* in the sense that the ZAM does, allowing arbitrary set extensions and intensions to be combined at runtime to form new sets, relations and functions.

Extending Functional Implementation Techniques

Machines based on implementation techniques for functional languages use *stack-based reduction* (as first described for the SECD machine, [Landin, 1964]), *continuation-passing style* reduction (CPS, Appel and Jim [1989]), and *graph-based reduction*, the most prominent instance of which is the STG (Spineless Tagless G-machine, [Jones and Salkild, 1988; Jones, 1992]). The stack-based implementations of functional languages have the advantage that they can be mapped to off-the-shelf hardware, making effective use of the target architecture's hardware stack and registers, as is achieved, e.g., in the implementation of the functional language OPAL by compilation of (recursive) OPAL functions to (recursive) C procedures [Schulte and Grieskamp, 1992; Schulte, 1992]. CPS techniques make the reduction stack implicit by passing a continuation to each function application

to be called with the result of evaluation, implying that "stack frames" are actually allocated on the heap. Given a good garbage-collection scheme, this technique can be rather efficient [Appel, 1987], since it allows a natural optimization of tail calls, efficient realization of exceptions (important for the implementation of ML [Milner et al., 1990]), and a stackless implementation of concurrent threads, as used, e.g., for CML [Reppy, 1991].

The STG-based implementations use the graph-reduction paradigm, resulting in compiled code which looks – interestingly enough – rather similar to the final code generated for higher-order functions in stack-based or CPS approaches. A "suspension" in the STG is represented as a closure with an associated evaluation method. The first time the suspension is forced, the evaluation result is memorized in the closure and the evaluation method is updated. The next time the suspension is consulted, the updated method simply retrieves the result from the previous evaluation. In the stack-based implementation of the strict language OPAL, exactly the same effect is achieved by memorization of the evaluation result of 0-ary function abstractions, $\lambda() \bullet e$, in closures, with the only significant difference to the STG that strict stack-based evaluation is the default and lazy closure-based evaluation the special case.

One of the first attempts to extend functional reduction by logic computation is the BAM [Kuchen et al., 1990] and its lazy version, the LBAM [Moreno-Navarro et al., 1990], which are used to implement the functional logic language BABEL [Moreno-Navarro and Rodriguez-Artalejo, 1992]. These machines extend graph reduction by nodes for logic variables and use choice points and trailing as in the WAM (and the ZAM) to deal with backtracking. An optimization by deterministic rule detection at runtime is described by Loogen and Winkler [1991]. If the application of a rule does not bind any logic variable, then – by orthogonality of the functional rule system – choice points for other cases can be discarded. The ZAM, by contrast, cannot rely on orthogonality, since whether a set is a function is not represented in μZ. Hpwever, we have shown how choice point scheduling results in similar effects as the "dynamic cut" of Loogen and Winkler [1991].

Based on the STG model and the above-described closure technique, the JUMP machine [Chakravarty and Lock, 1991; Lock, 1992; Chakravarty and Lock, 1997] provides an interesting approach for integrating logic computation into a functional machine. The notion of a "closure" is extended to an "active object", which is used to represent several kinds of runtime objects – among them suspensions and logical variables. "Jumping" in this machine amounts to calling the evaluation method of an active object, which behaves differently in dependency of kind and state of the object. The JUMP machine uses techniques similar to that of the WAM and the ZAM for dealing with backtracking: choice points are used to hold information for restoring the state of the machine. The machine can avoid the creation of choice points if functions are applied to ground terms, since it can assume orthogonality of the function's rules. One technical problem with the architecture of the JUMP machine is that it does not match well with modern, cache-based processor architectures, since it makes abundant use of indirect code jumps between very small code chunks – on these architectures, the older technique of "tag-based branching" for compiled code might be in fact more appropriate, and a *byte code interpreter* is not much slower.

The machines mentioned are examples of extending functional implementation techniques by logic computation. Again, these techniques, like the discussed extensions of the WAM, are inadequate for implementing μZ. The main problem is the lack of support

for concurrent constraint resolution and the possibility to freely combine extensions and intensions to build sets, relations and functions at runtime.

Concurrent Constraint Resolution

An influential design of a concurrent constraint-resolution machine is the implementation of the Andorra Kernel Language, AKL [Janson and Haridi, 1991; Franzen et al., 1992; Janson, 1994]. Among others, the "Andorra Principle" [Warren, 1990], of which the ZAM's technique of choice scheduling is an instance, is implemented by this machine.

One of the more recent designs of a constraint-resolution machine is the Oz machine, part of the *Mozart* system [Mehl et al., 1995; Scheidhauer, 1998; Mehl, 1999]. The Oz machine uses a so-called *store* to represent Oz's computation spaces, containing bindings, constraints and values. A so-called *worker* executes a thread, which is represented by a stack of so-called *tasks* that are executed until the stack is empty. Tasks are tuples of environments and instruction pointers. On execution of a task, new tasks may be pushed to the stack of a worker, thus realizing dynamically calculated control flow. Since tasks encapsulate their environment, they can be moved between thread workers: hence suspensions of computations that access unbound variables can be handled by moving the related task to a newly created, suspended thread worker and continuing with the next task on the stack of the current worker. One advantage of this model in comparsion with the ZAM is that, initially, only one thread is used to execute a set of constraints, whereas the ZAM spawns a thread for each constraint from the start. On the other hand, the representation of tasks and of context switching to environments attached to each task has to be paid for in the Oz machine, whereas thread creation in the ZAM costs only a few instructions. Moreover, minimizing the number of active threads can also be performed by optimization at compile time: based on a *mode analysis* that evaluates static data dependencies, a set of μZ constraints can be compiled to execute in a single ZAM thread.

The Oz machine does not use trailing and backtracking for implementing choices, but instead copies stores. This allows several search strategies (breadth-first, depth-first) to be easily realized. Copying is closely related to implementing search in functional languages using a "continuation" function that holds the alternative of a choice, the current set of context bindings, stored in a closure. The general problem with copying is that it is speculative: it is not known whether the copy will actually be used during subsequent computations.

A quite recent machine realizing concurrent constraint-resolution is described by Hanus and Sadre [1999] for the implementation of Curry. This machine uses "and parallelism" as usual in constraint resolution, but also "or parallelism" for the execution of disjunctions: independent threads are spawned for the different choices in a disjunction, effectively implementing breadth-first search, as required by Curry's semantics. Choice scheduling, resembling the Andorra principle [Warren, 1990] as in the ZAM, is applied to prune the overhead of choices. To avoid copying, this machine associates with each thread a unique key, and each variable contains a hash table of bindings indexed by thread keys. Though speculative copying is avoided, the overhead of this approach is that each variable access requires selection via a hash table.

The ZAM has a simpler and much more compact design than the other machines mentioned, which it inherits from the minimalistic μZ calculus. But the ZAM is also less

complex because it is restricted to built-in depth-first search, implemented by the classical trailing and backtracking technique. For problems that can be solved by depth-first search, this leads to more efficient execution results than copying of stores (as in the Oz machine) or variable hashing (as in the Curry machine), but problems which require breadth-first search cannot be handled adequately by the ZAM.

"Sie meinen also tatsächlich,
daß ein Kunstwerk kollektiv
hergestellt werden könnte?"
- : "Abärrjá!"
(Arno Schmidt: Die Gelehrtenrepublik)

Chapter 6

The ZeTa System

The concepts presented in this thesis are applied in the broader context of the ZeTa system, which was developed by the author together with Robert Büssow in the course of the ESPRESS project. We conclude the thesis with a review of ZeTa and the application of the μZ implementation, called ZaP, in the context of the ZeTa system.

6.1 Three Dimensions of Integration

The goal of the ZeTa system is to provide an *open* environment for editing, browsing and analyzing integrated specifications, assembled from heterogeneous formalisms – including Statecharts, Z, temporal logic, message-sequence charts and others. These formalisms are processed by different techniques, such as deduction-based analysis, model checking, systematic testing and simulation. Established CASE tools (such as Statemate [Harel et al., 1990]) shall be used in combination with the existing powerful tools for formal methods (such as the deduction system Isabelle [Paulson, 1994] or the model checker FDR [Formal Systems , Europe]), and specialized tools newly developed in the context of ESPRESS – such as the ZAP compiler.

ZeTa's design as an open environment for notations and tools was derived from the experiences in the application oriented research project ESPRESS, in which the industrial partners determined the choice of formalisms and tools. In basic research one can stick to a particular notation or tool over a period of years, carefully tuning the approach. On the other hand, in industrial research, comprehensible results are asked for soon and requirements and fashions change fairly quickly. From the point of view of basic research, it is therefore desirable to have a *flexible* approach with regards to the "front-end" technology, one which can easily cope with new requirements while at the same time allowing scope for *steady enhancement* of the underlying "back-end" technology.

The ZeTa system sets out to achieve this goal by defining three dimensions of integration at the front-end level: *semantic integration*, *document integration* and *tool integration*. This logical integration levels are accompanied by graphical user interfaces. The ultimate goal is to make basic tools, e.g. for computation, once they have been integrated on the semantic and the tool level, reusable for different document formats and input languages.

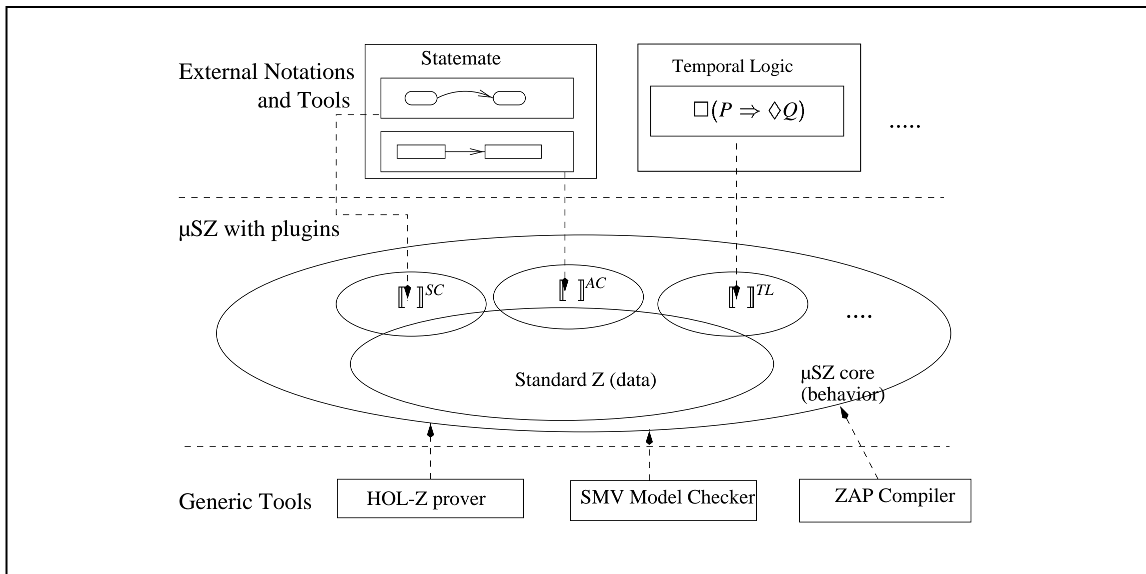


Figure 6.1: Semantic Integration in ZeTa

6.1.1 Semantic Integration

Semantic integration in ZeTa is realized by the $\mu\mathcal{SZ}$ notation [Büssow et al., 1996, 1997a; Büssow and Grieskamp, 1999]. $\mu\mathcal{SZ}$ is based on a *core*, which is a collection of conventions for specifiying *structure* and *behavior* in Standard Z. External notations are integrated by so-called *language plugins*, which are defined by a *shallow syntactic translation* into the $\mu\mathcal{SZ}$ core. This is illustrated in Fig. 6.1.

The $\mu\mathcal{SZ}$ core provides a model for *concurrent components*. Shared and private data can be described, thus constituting the data space. A *set of traces of bindings* of the data space defines the dynamic behavior. The dynamic behavior can be specified in several ways: for example, by giving a data-state-transition relation, which can be executed by ZaP, or by giving axiomatic constraints on the set of traces (which are not as easy to execute – but see, however, future work in Chapter 7). Prior to trace construction, the data space is extended by information to model *concurrent access* to its variables. To this end, each variable of the data space is equipped with a *lock*, which can be acquired by concurrently executing activities. The locks allow us to resolve *racing*, the situation in which two parallel activities write to the same variable during the same execution step. Racing is a prototypical situation of concurrent resource allocation and can be used to model higher-level communication facilities, such as channels and service-access points (*i.e.* remote procedure calls). For further details, see [Büssow and Grieskamp, 1997, 1999].

To integrate a new formalism, it is necessary to define a translation of this formalism into the $\mu\mathcal{SZ}$ core. For tool integration, a so-called *model source adaptor* that does the translation has to be implemented. The adaptor concept is outlined below in Section 6.1.3 (on the next page). Whether integration of a formalism is possible depends on whether a translation can be found. To what extent generic tools such as ZAP can be reused depends on the $\mu\mathcal{SZ}$ core definitions resulting from this translation.

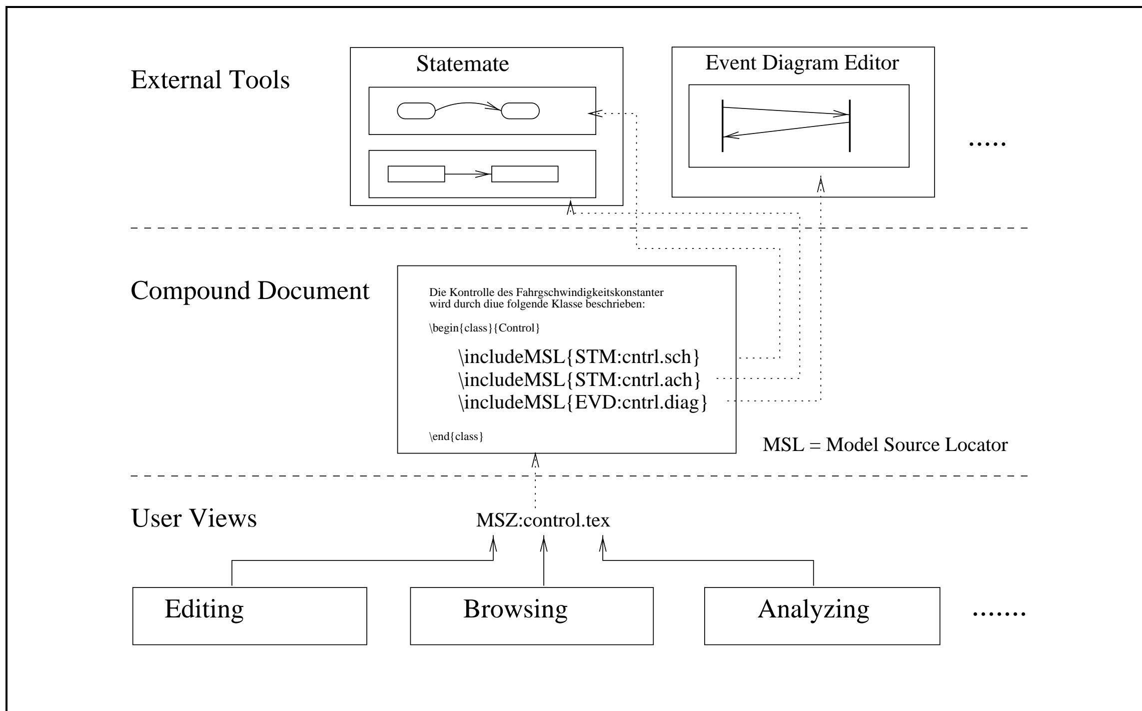


Figure 6.2: Document Integration in ZeTa

6.1.2 Document Integration

The "cement" of integration in the ZeTa environment is an *integrated specification document* (ISD). In most cases, this is a $\mu\mathcal{SZ}$ specification, but in general an ISD needs not to contain formal parts.

An ISD is given in a markup language and contains references to fragments of specification contributed by external tools. The references are described by *model source locators* (MSL), a concept comparable to URLs. An MSL identifies the tool that is to provide the (sub-)model and a tool specific identifier for the desired portion. This is illustrated in Figure 6.2. Besides the external tool Statemate, a tool for editing event diagrams (providing a graphical frontend for temporal logics) is shown as an example.

The major difference between the concept of URLs and that of MSLs is that the latter are interpreted in different ways, depending on the operation to be performed on an ISD. For example, if an ISD is browsed, its MSLs are resolved to encapsulated postscript. If it is analyzed, the MSLs may be resolved to an $\mu\mathcal{SZ}$ core abstract syntax term which is inserted in the enclosing component (depending on the kind of analysis and MSL).

The ISD itself is addressed by an MSL as well: if the ISD is given in the LaTeX-markup, then such an MSL looks like this: `LTX:ICC#CruiseControl.tex`. Here, `LTX` denotes the tool providing this MSL (the LaTeX-System), `ICC` denotes a $\mu\mathcal{SZ}$ component and `CruiseControl.tex` a file where this component is found.

6.1.3 Tool Integration

The basic operation provided by the ZeTa integration environment is *querying* the content of an MSL. This query is parameterized by the *type* of the desired content. Typical types are "gif", "encapsulated postscript", "parsed abstract syntax", "type-checked ab-

stract syntax”, “plugin abstract syntax”, and so on. Types may become more complex containing further selective information, e.g. “the precondition of schema *S*”.

The ZeTa environment behaves as a broker between queries and tools that have registered in the environment the ability to compute content of specific types. Once a tool is contacted to compute a particular content, it may recursively query the environment for further content. For example, when a type checker is contacted to compute the “checked abstract syntax” of an $\mu\mathcal{SZ}$ class, it first queries the “parsed abstract syntax”. If this syntax contains MSLs referring to external notations, it next queries the “plugin abstract syntax”, then merges the syntaxes, and finally performs its type-check operation. This way, tool chains are realized.

The basic model of ZeTa is comparable to a traditional Unix `make`, but it is more powerful and reliable, since it is embedded in the strongly-typed Java programming environment:

- The content exchanged by tools via ZeTa are standardized data formats defined by Java APIs. For example, for the $\mu\mathcal{SZ}$ language (and the Z language embedded in $\mu\mathcal{SZ}$), the abstract syntax is prescribed by ZeTa in a Java representation.
- A tool is represented in a ZeTa session as a Java object whose instances implement the interface of so-called *tool adaptors*, and which can have state and behavior. The adaptor object may implement its functionality completely in Java, or may wrap another Unix process, or may be an RMI (remote method invocation) proxy to a remotely running JVM.
- Tool adaptors may import external models into the ZeTa session from other running tools, not just the file system. For instance, the Statemate adaptor imports initial content for its MSLs, `STM:<chart>`, by connecting to the “dataport” of a running Statemate session.
- ZeTa caches content resulting from computations, thus providing an outdating mechanism. While `make` uses time stamps of files to decide whether an initial content is outdated, in ZeTa adaptors that provide initial content may use arbitrary techniques for this purpose.
- ZeTa realizes a session protocol for computations, which keeps a record of the diagnostics produced by tools, among other things.

The plugins of the $\mu\mathcal{SZ}$ notation directly correspond to certains kind of tool adaptors plugged into the ZeTa environment (in fact, historically, the ZeTa plugins preceded the $\mu\mathcal{SZ}$ plugins). These adaptors, which import an external notation into ZeTa by providing initial content for MSLs, are called *model source adaptors* (MSA).

In the simplest case, an MSA directly performs the translation of the external notation to the $\mu\mathcal{SZ}$ core. Usually, however, there are intermediate steps in the way from an external notation to $\mu\mathcal{SZ}$, and the intermediate content may be used by tools specially tailored for its type.

This is illustrated in Figure 6.3 (on the next page). The Statemate-MSA initially creates an abstract syntax for the Statechart, denoted by the MSL `STM:ICCCONTROL`, which, in a second step, is converted to the $\mu\mathcal{SZ}$ core syntax. In turn, the LaTeX-MSA

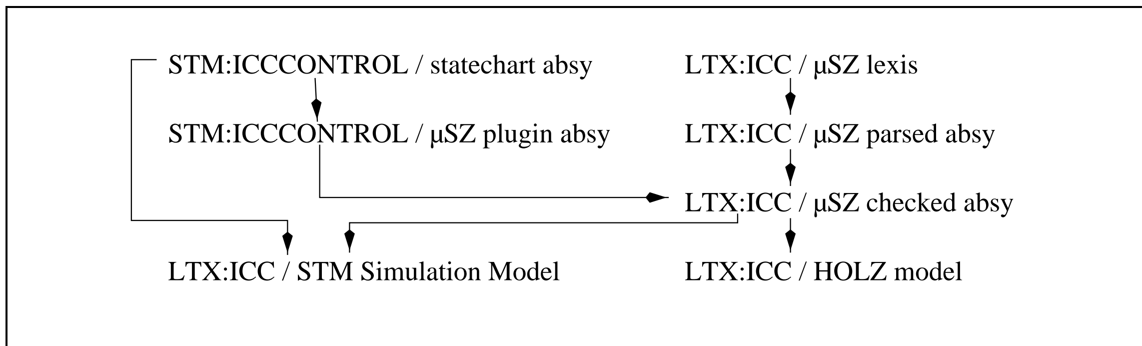


Figure 6.3: Chains of Content Queries

initially creates a $\mu\mathcal{SZ}$ lexis for the MSL `LTX:ICC` (the file name in the MSL has been omitted), which is converted first to a parsed abstract syntax, then to a type-checked abstract syntax. Assuming that the component denoted by `LTX:ICC` refers to the MSL `STM:ICCCONTROL`, the type checker queries the abstract syntax of the $\mu\mathcal{SZ}$ plugin associated with this MSL and includes it in the resulting type-checked abstract syntax (thus checking the type consistency of the plugin within its component context).

Now the generation of a model for the HOLZ prover adaptor [Santen, 1999] is entirely based on the checked abstract syntax, with the plugins expanded. However, in order to generate a model for the Statemate simulator, it is necessary to query the checked $\mu\mathcal{SZ}$ syntax as well as the original representation of the Statechart. Generating a simulation model amounts to compiling code for the Z schemas referred to from transitions and bind them to the Statemate simulator. To generate the binding code, the structure of the Statechart has to be known.

This example shows that in the real world the integration provided by $\mu\mathcal{SZ}$ plugins is not always the only underlying representation for tools. The translation of an external notation to the $\mu\mathcal{SZ}$ core is an abstraction step in which information (in a syntactical sense) is lost.

6.1.4 Graphical User Interfaces

ZeTa realizes a strict partition of model, view and controller. The model is effectively implemented by the individual tool adaptors and the ZeTa content-query broker. The controller is based on a command language that can be also used in batch mode. Two "views" (GUIs) are provided on top of the controller.

XEmacs GUI. This GUI is based on the extendable text editor XEmacs (see Figure 6.4). It is primarily intended for users working under UNIX, who are used to the Emacs editing environment. It maintains a session log, which utilizes for each operation the produced output and supports browsing diagnostics contained in the output. It further provides VTeX, a "what-you-see-is-what-you-mean" editing mode for LaTeX/Z documents. VTeX is a customizable XEmacs mode that incrementally parses and visualizes the LaTeX structures and math symbols entered by the user.

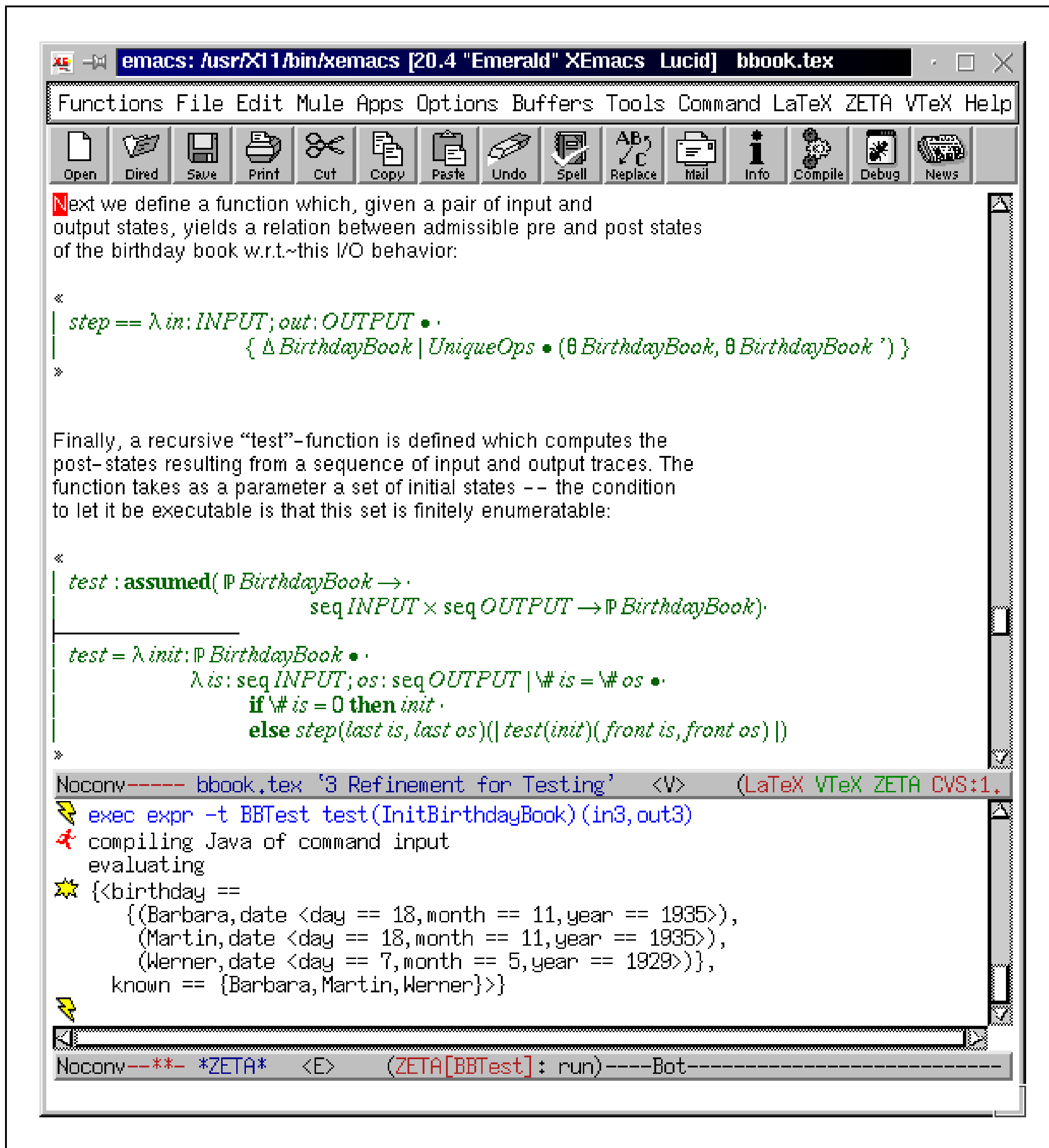


Figure 6.4: The XEmacs GUI

Swing GUI. This GUI is based on Java's Swing system (see Figure 6.5). It allows the activation of ZeTa operations by graphical dialogs and maintains a session log that utilizes for each operation the produced output. Browsing diagnostics is supported by contacting an external editor, which may be Emacs, but other editors (in particular. MS-Windows-specific editors) may be used as well.

6.2 The ZaP Compiler

ZaP (version 1) is an experimental compiler for the Z language, which supports the full higher-order *functional* sublanguage of Z. It is a predecessor of the forthcoming ZaP-

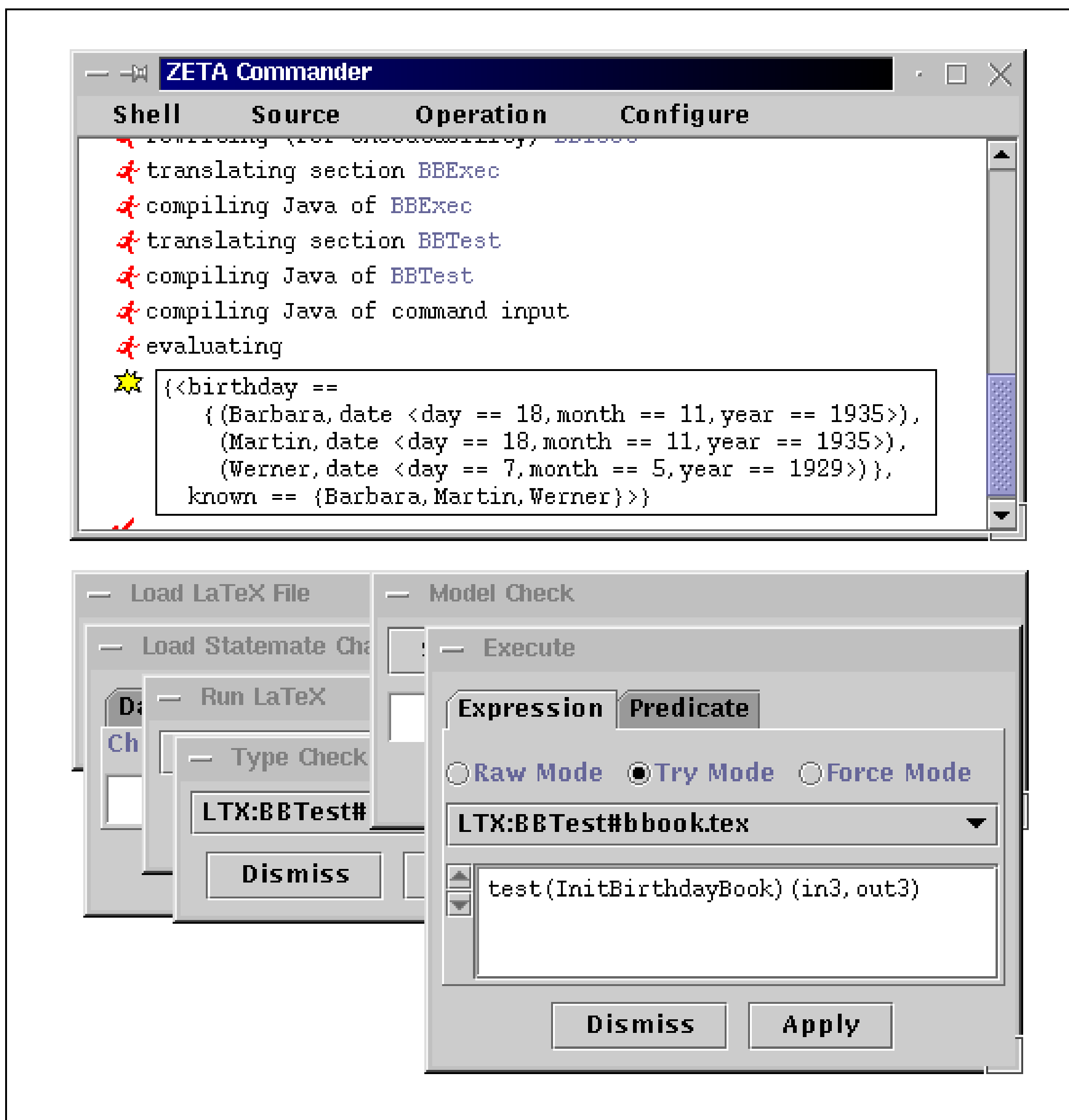


Figure 6.5: The Swing GUI

2, whose underlying concepts are described in this thesis and which will support the *functional logic* sublanguage embedded in Z. Apart from its more powerful computation model, the forthcoming ZaP-2 will not differ from ZaP-1, so we can present the user's view of ZaP based on the old system.

ZaP follows the approach of compiling every Z construct. Nonexecutable constructs of Z will be rejected once their execution has been attempted at runtime. This approach reflects that whether a Z construct is executable depends on how it is used at runtime. For example, a function might be defined in such a way that there is no constructive mapping from input to output and that the application of the function to an input is not executable. However, it might be still possible to test whether a given pair of input and output values is in the function's graph[1].

[1]For future versions of ZaP, it is planned to have an analysis which gives hints for non-executable

Below, the well-known example of the *birthday book* specification is used to illustrate the basic operation of ZaP. Please note that this example *cannot* serve as a systematic outline of ZaP's computation power, but it at least illustrates the ability to execute the schema calculus, to lift schemas to functions and to define recursive higher-order functions and relations – in an application context for test data evaluation. Starting with the usual abstract specification of the birthday book, we refine it for the purpose of executability (mainly fixing given types in the specification), and add a framework for evaluating *test data* described by input/output behavior.

6.2.1 The Basic Specification of the Birthday Book

The specification is partitioned into several Standard Z sections. The first one, *BBSpec*, contains the basic, or "abstract" specification of the birthday book example, as usually found in the literature. The section is opened with the following declaration (all Z paragraphs following this declaration until the next section declaration will belong to *BBSpec*):

section *BBSpec*

We first define the given types *NAME* and *DATE*, and the data state of the birthday book:

$[NAME, DATE]$

BirthdayBook

$known : \mathbb{P}\ NAME$
$birthday : NAME \pfun DATE$

$known = \operatorname{dom} birthday$

An *initial state* of the birthday book is given by the following schema:

$InitBirthdayBook == [BirthdayBook \mid birthday = \emptyset]$

The value of this schema can be evaluated. In order to perform the evaluation the user types in the name *InitBirthdayBook* in the `Execute` panel of the Swing GUI or in the command line of the XEmacs GUI, resulting in the following (conceptual) output:

InitBirthdayBook

```
⇛ {<birthday == {},known == {}>}
```

Note that the field *known* of *InitBirthdayBook* is implicitely defined by the invariant of the schema *BirthdayBook*, $known = \operatorname{dom} birthday$.

Up to now, we have specified the state of the birthday book and a possible initial value, but no state transitions. These are defined by the following Z "operation" schemas:

constructs at compilation time.

AddBirthday

$\Delta BirthdayBook$
$name? : NAME; \; date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

FindBirthday

$\Xi BirthdayBook$
$name? : NAME; \; date! : DATE$

$name? \in known$
$date! = birthday(name?)$

Remind

$\Xi BirthdayBook$
$today? : DATE; \; cards! : \mathbb{P}\, NAME$

$cards! = \{n : known \mid birthday(n) = today?\}$

Remove

$\Delta BirthdayBook$
$name? : NAME$

$birthday' = \{name?\} \lhd birthday$

The schema *AddBirthday* adds a mapping $name? \mapsto date?$ to the birthday book's state, the schema *FindBirthday* lookups the birthday of a given $name?$, and the schema *Remind* yields the set of names whose birthday is $today?$. The schema *Remove* deletes the entry for $name?$ from the birthday book.

We are not yet able to execute any state transitions, since the types *NAME* and *DATE* are given (uninterpreted). In the next section, we will refine these types.

6.2.2 Refining the Birthday Book for Execution

A refinement of the section *BBSpec* for the purpose of execution is provided by the section *BBExec*:

section *BBExec* **parents** *BBSpec*

The purpose of the refinement is to fix the given types *NAME* and *DATE*. The Z of ZeTa allows to refine given types by free types:

$NAME ::= Werner \mid Barbara \mid Martin$
$DATE ::= date\langle\!\langle Date \rangle\!\rangle$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Date}} \\
\hline
day : 1 \mathinner{..} 31 \\
month : 1 \mathinner{..} 12 \\
year : \mathbb{N}_1 \\
\hline
month = 2 \Rightarrow day \leq 29;\ month \in \{4, 6, 7, 9, 11\} \Rightarrow day \leq 30 \\
\hline
\end{array}$$

We can now evaluate expressions by applying the *date* constructor to bindings:

$$date \langle\!| day == 1, month == 1, year == 1999 |\!\rangle$$

```
⇛ date <day == 1,month == 1,year == 1999>
```

The next application of the *date* constructor is undefined, since the passed binding is not in its domain (the month February has no 30th day):

$$date \langle\!| day == 30, month == 2, year == 1999 |\!\rangle$$

```
⇛ ERROR[LTX:bbook.tex(186.3-186.56)]:
     execution failed
      reason:
       application undefined:
        relation: date
        argument: <day == 30,month == 2,year == 1999>
      backtrace:
       at testing membership:
           value: (30,29)
           set: _\leq_
       at testing membership:
           value: <day == 30,month == 2,year == 1999>
           set: LTX:bbook.tex(171.22-171.32)
       at applying relation:
           relation: date
           argument: <day == 30,month == 2,year == 1999>
       at evaluating command input
```

ZaP prints a *backtrace* describing the reason of the undefinedness. The ZeTa commander – either via the Java GUI or the XEmacs GUI – allows to browse locators as contained in the above output by clicking on them.

Next we introduce some convenience functions for describing operations on birthday books. These functions make use of the new features of the Z ISO Standard as regards to the unification of schema expressions and plain expressions. They yield a set of bindings (a schema) which can be referred in schema expressions:

$$\begin{array}{|l}
add == \lambda\, name : NAME;\ date : DATE \bullet \\
\qquad\qquad (\exists\, name? == name;\ date? == date \bullet AddBirthday) \\
find == \lambda\, name : NAME \bullet (\exists\, name? == name \bullet FindBirthday) \\
remind == \lambda\, today : DATE \bullet (\exists\, today? == today \bullet Remind) \\
remove == \lambda\, name : NAME \bullet (\exists\, name? == name \bullet Remove)
\end{array}$$

Note that the existential quantifiers in these definitions are *schema quantifiers*.

We can now execute transitions of the birthday book's state, starting with the initial state:

$$
\begin{array}{l}
\mathit{InitBirthdayBook} \\
\quad \fatsemi \mathit{add}(\mathit{Werner}, \mathit{date}\lblot \mathit{day} == 7, \mathit{month} == 5, \mathit{year} == 1929 \rblot)
\end{array}
$$

```
⇛ {<birthday == {},
     birthday' ==
       {(Werner,date <day == 7,month == 5,year == 1929>)},
     known == {},known' == {Werner}>}
```

$$
\begin{array}{l}
\mathit{InitBirthdayBook} \\
\quad \fatsemi \mathit{add}(\mathit{Werner}, \mathit{date}\lblot \mathit{day} == 7, \mathit{month} == 5, \mathit{year} == 1929 \rblot) \\
\quad \fatsemi \mathit{add}(\mathit{Barbara}, \mathit{date}\lblot \mathit{day} == 18, \mathit{month} == 11, \mathit{year} == 1935 \rblot) \\
\quad \fatsemi \mathit{add}(\mathit{Martin}, \mathit{date}\lblot \mathit{day} == 18, \mathit{month} == 11, \mathit{year} == 1935 \rblot) \\
\quad \fatsemi \mathit{find}(\mathit{Werner}) \\
\quad \fatsemi \mathit{remind}(\mathit{date}\lblot \mathit{day} == 18, \mathit{month} == 11, \mathit{year} == 1935 \rblot) \\
\quad \fatsemi \mathit{remove}(\mathit{Martin})
\end{array}
$$

```
⇛ {<birthday == {},
     birthday' ==
      {(Barbara,date <day == 18,month == 11,year == 1935>),
       (Werner,date <day == 7,month == 5,year == 1929>)},
     cards! == {Barbara,Martin},
     date! == date <day == 7,month == 5,year == 1929>,
     known == {}, known' == {Barbara,Werner}>}
```

The last evaluation deserves some notes on the schema calculus operator, $_ \fatsemi _$. $Op_1 \fatsemi Op_2$ binds primed names x' from the signature of Op_1 to unprimed counterparts x in the signature of Op_2. Variables not bound this way (such as *cards*! and *date*!) or kept as is. In the evaluation result, these variables represent the result of applying the *find* and *remind* operations, respectively.

The application of a unique schema *InitBirthdaybook* (representing by a singleton binding set) to deterministic operations yields a singleton binding set. But ZaP is also capable of handling non-unique initial states and indeterministic operations. Consider the following definition:

$$
\begin{array}{l}
\mathit{InitBirthdayBookUndet} == \\
\quad \mathit{InitBirthdayBook} \lor \\
\quad [\mathit{BirthdayBook} \mid \\
\qquad \mathit{birthday} = \\
\qquad\quad \{\mathit{Werner} \mapsto \mathit{date}\lblot \mathit{day} == 7, \mathit{month} == 5, \mathit{year} == 1929 \rblot\}]
\end{array}
$$

Evaluation yields in:

$\mathit{InitBirthdayBookUndet}$

```
⇛ {<birthday == {},known == {}>,
    <birthday ==
      {(Werner,date <day == 7,month == 5,year == 1929>)},
     known == {Werner}>}
```

$$\mathit{InitBirthdayBookUndet} \\ \quad \fatsemi (\mathit{add}(\mathit{Barbara}, \mathit{date}\lblot \mathit{day} == 18, \mathit{month} == 11, \mathit{year} == 1935 \rblot) \\ \qquad \lor \mathit{add}(\mathit{Martin}, \mathit{date}\lblot \mathit{day} == 18, \mathit{month} == 11, \mathit{year} == 1935 \rblot))$$

```
⇛ {<birthday == {},
     birthday' ==
      {(Barbara,date <day == 18,month == 11,year == 1935>)},
     known == {},known' == {Barbara}>,
    <birthday == {},
     birthday' ==
      {(Martin,date <day == 18,month == 11,year == 1935>)},
     known == {},known' == {Martin}>,
    <birthday ==
      {(Werner,date <day == 7,month == 5,year == 1929>)},
     birthday' ==
      {(Barbara,date <day == 18,month == 11,year == 1935>),
       (Werner,date <day == 7,month == 5,year == 1929>)},
     known == {Werner},
     known' == {Barbara,Werner}>,
    <birthday ==
      {(Werner,date <day == 7,month == 5,year == 1929>)},
     birthday' ==
      {(Martin,date <day == 18,month == 11,year == 1935>),
       (Werner,date <day == 7,month == 5,year == 1929>)},
     known == {Werner},
     known' == {Martin,Werner}>}
```

Of course, such evaluations might easily lead to an explosion of possible states. ZaP-1 will run into efficiency problems if they are hundreds of them – ZaP-2, using the computation model described in this thesis, shall easily cope with several ten-thousands of states.

6.2.3 Refining the Birthday Book for Testing

A refinement of the section *BBExec* for the purpose of evaluation of test data with ZAP is given by the Z section *BBTest*:

section *BBTest* **parents** *BBExec*

The test-data will be specified by sequences of input and output behavior of the birthday book. However, the original specification does not provide unique input and output interfaces. But we can *lift* the original specification without modifying it to such unique

interfaces. First, we define free types for describing the input and output behavior:

$$
\begin{aligned}
&\mathit{INPUT} ::= \\
&\quad \mathit{addI}\langle\!\langle \mathit{NAME} \times \mathit{DATE} \rangle\!\rangle \mid \\
&\quad \mathit{findI}\langle\!\langle \mathit{NAME} \rangle\!\rangle \mid \\
&\quad \mathit{remindI}\langle\!\langle \mathit{DATE} \rangle\!\rangle \mid \\
&\quad \mathit{removeI}\langle\!\langle \mathit{NAME} \rangle\!\rangle \\
&\mathit{OUTPUT} ::= \\
&\quad \mathit{okayO} \mid \\
&\quad \mathit{dateO}\langle\!\langle \mathit{DATE} \rangle\!\rangle \mid \\
&\quad \mathit{namesO}\langle\!\langle \mathbb{P}\, \mathit{NAME} \rangle\!\rangle
\end{aligned}
$$

Next, we lift the operations of the original specification to ones with unique input/output interfaces:

$$
\begin{aligned}
&\mathit{UniqueAdd} == \\
&\quad [\mathit{AddBirthday};\ \mathit{in} : \mathit{INPUT};\ \mathit{out} : \mathit{OUTPUT} \mid \\
&\qquad (\mathit{name}?, \mathit{date}?) \mapsto \mathit{in} \in \mathit{addI};\ \mathit{out} = \mathit{okayO}] \setminus (\mathit{name}?, \mathit{date}?) \\
&\mathit{UniqueFind} == \\
&\quad [\mathit{FindBirthday};\ \mathit{in} : \mathit{INPUT};\ \mathit{out} : \mathit{OUTPUT} \mid \\
&\qquad \mathit{name}? \mapsto \mathit{in} \in \mathit{findI};\ \mathit{date}! \mapsto \mathit{out} \in \mathit{dateO}] \setminus (\mathit{name}?, \mathit{date}!) \\
&\mathit{UniqueRemind} == \\
&\quad [\mathit{Remind};\ \mathit{in} : \mathit{INPUT};\ \mathit{out} : \mathit{OUTPUT} \mid \\
&\qquad \mathit{today}? \mapsto \mathit{in} \in \mathit{remindI};\ \mathit{cards}! \mapsto \mathit{out} \in \mathit{namesO}] \\
&\qquad\qquad \setminus (\mathit{today}?, \mathit{cards}!) \\
&\mathit{UniqueRemove} == \\
&\quad [\mathit{Remove};\ \mathit{in} : \mathit{INPUT};\ \mathit{out} : \mathit{OUTPUT} \mid \\
&\qquad \mathit{name}? \mapsto \mathit{in} \in \mathit{removeI};\ \mathit{out} = \mathit{okayO}] \setminus (\mathit{name}?)
\end{aligned}
$$

ZaP has no problems with the execution of such morphed operations. Instead of illustrating this now, we directly continue by defining a framework which *tests* if sequences of *INPUTS* and *OUTPUTS* do match the specification.

The possible state transitions of the birthday book are defined as the disjunction of all the operations with unique I/O interface:

$$\mathit{UniqueOps} == \mathit{UniqueAdd} \lor \mathit{UniqueFind} \lor \mathit{UniqueRemind} \lor \mathit{UniqueRemove}$$

For test data evaluation, we define a function which, given a pair of input and output states, yields a relation between admissible pre- and post-states of the birthday book w.r.t. the I/O behavior:

$$
\begin{aligned}
&\mathit{step} == \\
&\quad \lambda\, \mathit{in} : \mathit{INPUT};\ \mathit{out} : \mathit{OUTPUT} \bullet \\
&\quad \{\Delta \mathit{BirthdayBook} \mid \mathit{UniqueOps} \bullet \theta \mathit{BirthdayBook} \mapsto \theta \mathit{BirthdayBook}'\}
\end{aligned}
$$

Finally, a recursive “test”-function is defined which computes the post-states resulting

from a sequence of input and output traces. The function takes as a parameter a set of initial states – the condition to let it be executable is that this set is finitely enumeratable:

$$\begin{array}{|l} test : \mathsf{assumed}(\mathbb{P}\, BirthdayBook \longrightarrow \\ \qquad\qquad \mathrm{seq}\, INPUT \times \mathrm{seq}\, OUTPUT \longrightarrow \mathbb{P}\, BirthdayBook) \\ \hline test = \lambda\, init : \mathbb{P}\, BirthdayBook \bullet \\ \qquad \lambda\, is : \mathrm{seq}\, INPUT;\ os : \mathrm{seq}\, OUTPUT \mid \#is = \#os \bullet \\ \qquad\quad \mathbf{if}\ \#is = 0\ \mathbf{then}\ init \\ \qquad\quad \mathbf{else}\ step(last\, is, last\, os)(\!| test(init)(front\, is, front\, os) |\!) \end{array}$$

The $\mathsf{assumed}\ _$ constructor around the type of the function *test* tells ZaP not to try to check whether the set *test* is actually a total function of the specified type. This check would not be executable.

To test "test", some I/O traces representing test data are defined:

$$\begin{array}{|l} date1 == date\langle\!| day == 7, month == 5, year == 1929 |\!\rangle \\ date2 == date\langle\!| day == 18, month == 11, year == 1935 |\!\rangle \end{array}$$

$$\begin{array}{|l} in1 == \langle addI(Werner, date1) \rangle \\ out1 == \langle okayO \rangle \end{array}$$

$$\begin{array}{|l} in2 == \langle addI(Werner, date1), findI(Werner) \rangle \\ out2 == \langle okayO, dateO(date1) \rangle \\ out2a == \langle okayO, dateO(date2) \rangle \end{array}$$

$$\begin{array}{|l} in3 == \langle addI(Werner, date1), addI(Barbara, date2), \\ \qquad addI(Martin, date2), findI(Werner), remindI(date2) \rangle \\ in3a == \langle addI(Werner, date1), addI(Werner, date2), \\ \qquad addI(Barbara, date2), findI(Werner), remindI(date2) \rangle \\ out3 == \langle okayO, okayO, okayO, dateO(date1), \\ \qquad namesO(\{Barbara, Martin\}) \rangle \\ out3a == \langle okayO, okayO, okayO, dateO(date1), namesO(\{Barbara\}) \rangle \end{array}$$

We can now run some evaluations:

$test(InitBirthdayBook)(in1, out1)$

```
⇛ {<birthday ==
     {(Werner,date <day == 7,month == 5,year == 1929>)},
    known == {Werner}>}
```

$test(InitBirthdayBook)(in2, out2)$

```
⇛ {<birthday ==
     {(Werner,date <day == 7,month == 5,year == 1929>)},
    known == {Werner}>}
```

$test(InitBirthdayBook)(in2, out2a)$

```
⇛ {}
```

$test(InitBirthdayBook)(in3, out3)$

```
⇛ {<birthday ==
      {(Barbara,date <day == 18,month == 11,year == 1935>),
       (Martin,date <day == 18,month == 11,year == 1935>),
       (Werner,date <day == 7,month == 5,year == 1929>)},
    known == {Barbara,Martin,Werner}>}
```

$test(InitBirthdayBook)(in3a, out3)$

```
⇛ {}
```

$test(InitBirthdayBook)(in3, out3a)$

```
⇛ {}
```

Failing tests are indicated in that they yield an empty set as the post-state of an execution. A refined version of the function *test* is given as a partial function, which always yields nonempty sets:

$$
\begin{array}{l}
ptest : \mathsf{assumed}(\mathbb{P}\, BirthdayBook \longrightarrow \\
\qquad\qquad \mathrm{seq}\, INPUT \times \mathrm{seq}\, OUTPUT \mathrel{+\!\!\!\rightarrow} \mathbb{P}\, BirthdayBook) \\
\hline
ptest = \lambda\, init : \mathbb{P}\, BirthdayBook \bullet \\
\qquad \lambda\, is : \mathrm{seq}\, INPUT;\ os : \mathrm{seq}\, OUTPUT \mid \#is = \#os \bullet \\
\qquad\quad \mathbf{if}\ \#is = 0\ \mathbf{then}\ init \\
\qquad\quad \mathbf{else}\ (\mu\, post == step(last\, is, last\, os) \\
\qquad\qquad\qquad (\!|ptest(init)(front\, is, front\, os)|\!) \mid \\
\qquad\qquad post \neq \emptyset)
\end{array}
$$

The μ-form lets the executing function be undefined if the *post* states become empty. Evaluating a failing test now results in a detailed backtrace which tells us the reason for the failure (the actual backtrace has been ommited):

$ptest(InitBirthdayBook)(in2, out2a)$

```
⇛ ERROR[LTX:bbook.tex(530.2-530.36)]:
    execution failed
     reason:
      mu-value undefined:
       set: {}
      ...
```

6.3 Discussion

We briefly discuss integration frameworks related to ZeTa and the execution of Z specifications.

Integration Frameworks

Few projects tackle the problem of integrating tools for formal methods and existing commercial CASE tools like the ZeTa system. The approach of the UniForM Workbench [Krieg-Brückner et al., 1999] comes closest to doing so. However, whereas ZeTa's integration language Java is object-oriented, UniForM employs a functional language (Haskell) for this task. Also, the architecture of UniForM is more ambitious (and thus more difficult to implement and maintain) as regards to *version* control. On the other hand, UniForM is less specialized than ZeTa and does not provide conventions for integrating *notations* and *documents*.

Other approaches like the Concurrency Factory [Cleaveland et al., 1994] or AutoFocus [Schätz and Huber, 1999] present more homogeneous environments, with most tools written from scratch and less emphasis on integrating existing tools.

Encoding of Z

Some basic principles of a mapping from Z to μZ have been outlined in Chapter 3, but we have not discussed the intrinsic details. In the ZaP implementation, the mapping of Z to μZ has to convert axioms to definitions where possible, e.g., eliminate quantifiers in axioms of the kind $\forall x : X \bullet f(x) = e$, map recursion to μZ's fixed-point operator, and so on. The mapping must also weaken the specification by turning some axioms into assumptions: if a user, for example, declares a function to be "total" by using Z's total function arrow, $f : A \longrightarrow B$, this assertion cannot be computed if f's definition is non-finite, and need therefore be converted into an assumption (which can be done explicitely by the user or implicitely by ZaP, depending on switches). All this has been implemented in the ZaP tool and will be ported to the new design of functional logic computation in the forthcoming ZaP-2.

Other Approaches Executing Z

Animation of the "imperative" part of Z is provided by the ZANS tool [Jia, 1996], imperative meaning Z's specification style for sequential systems using state transition schemas. This approach is highly restricted, as it cannot even handle the functional sublanguage of Z. An elaborated *functional approach* for executing Z has been described by Valentine [1992, 1995], though no implementation exists today. A translation to Haskell is described in [Goodman, 1995], where monads are used for dealing with the sequential specification style, but no logic resolution is employed. Mappings to Prolog are described, e.g., in [West and Eaglestone, 1992]. Mappings of Z to Mercury are described in [Winikoff et al., 1998]: however, in this approach the effective power of execution is restricted compared with ours, because the data flow has to be determined statically. The approach presented in this thesis goes beyond all the others, since it allows the combination of the functional and logic aspects of Z in a higher-order setting, treating sequential behaviors

as first-order citizins, comparable as to the integration of imperative computation into functional languages using monads.

Chapter 7

Conclusion

> *Dreiundsechzighundert Hektometer*
> *überm Spiegel seiner Wohnung steht er;*
> *sieht die Gasschiff-Flotte der Korona*
> *und erblickt das Mondschaf in Persona.*
>
> (Christian Morgenstern:
> Palmström: Angewandte Wissenschaft)

Motivated by the desire of executing a subset of the set-based specification language Z, we have introduced the set-based calculus μZ, defined a computation model for μZ as well as an abstract machine implementing this model, a compilation of μZ to the machine's instruction set and an efficient implementation strategy for the machine in C++. The application of this work in the ZeTa system has also been outlined. A discussion of the results obtained and of related work has been given in the relevant chapters. In this concluding chapter, we summarize the contributions of the work and discuss some possible directions for the future.

7.1 Contributions

μZ Calculus

The μZ calculus provides a new way of viewing the λ-calculus, predicate calculus, schema/module calculus, set algebra, etc. within a unified framework. The calculus is considered an "invention of the obvious": to the author's knowledge, there is no comparable language though none of the ingredients of the calculus are actually new. The most unusual individual construct of μZ is its operator of set translation, providing a means for the constructive description of morphisms in the category of μZ sets and, in combination with the other operators, supplying μZ's expressive power.

μZ's *semantic model* of partial set algebras, a generalization of three-valued logics, is – at least as a model for Z related languages – a new approach. The author believes that it is actually a more suitable model for Z itself than the one currently proposed by Z's Draft ISO Standard. The standard leaves the treatment of undefinedness and partiality quite vague, in an unfortunate formal method's tradition. In this thesis, we have at least shown that it is possible to define a model of undefinedness for a set-based formalism that is based on simple notions of lattice theory and does not require deep insight into mathematical set theory (which cannot be expected of computer scientists.)

Computation Model

The reference computation model of μZ, defined in the style of natural semantics, provides a new view of computation in the realm of languages that can be encoded in μZ,

leading to a realistic efficient implementation. New is the representation of set values by unions of singleton-set values and of intensions, the latter describing "computation receipts" for sets. Intensions are a generalization of "closures", as used in computation models for functional languages.

Given the minimality of the μZ calculus, its computation model could have been kept minimal as well, and the different embedded models – functional reduction and concurrent constraint resolution – could have been clearly worked out. The interfaces of these two worlds of computation are clearly visible – via the μ-operator that invokes constraint resolution from value reduction and via the $p \in e$ constraint that calls value reduction for e from constraint resolution. The presence of nondeterministic computation and the need to introduce choice points in an implementation is isolated in computing a constraint $p \in v_1 \cup v_2$, where we can choose between $p \in v_1$ and $p \in v_2$. The higher-order nature of the model shows up in the instantiation of an intension for the computation of $p \in [\![p' \mid \sigma \parallel \phi]\!]$, where we "import" the first-order citizen, the constraint $\sigma \parallel \phi$, into a resolution context.

Abstract Machine

The abstract machine, ZAM, and its implementation emerged as a consequence of μZ's computation model. It was possible to keep the machine small, being easy to understand (relative to the inherent complexity of the computation problem in μZ). The overall architecture of the machine is innovative in this respect. Most of the organizational details of the machine can be regarded as "folklore": for example, implementation of residuation using wait sets at variables, or employing the "Andorra Principle" for choice point scheduling.

The implementation of the ZAM in C++ provides an innovative architecture by its persistent single-assignment registers. On the one hand, persistent registers allow an efficient realization of thread dumping, because only the program counter needs to be trailed, and on the other hand, they enable callees to export references to register regions out of their lexical scope. In combination with conservative garbage collection, this leads to an integrated stack/heap model of memory allocation, which can be utilized by high-level optimizations that employ a region analysis.

Formalization in Z

We have given a comprehensive formal specification in strictly type-checked Z of μZ's syntax and semantics, equational theory, computation model and abstract machine. Thus, this work represents a case study for the use of Z as a meta language. For this purpose, it was essential to extend Z. The first category of extensions we made – such as inference-rule systems and recursive partial functions – could be explained as pure syntactic sugar and are actually implemented this way by the *ESZ* macro preprocessor. The second category – such as independence of the textual order of paragraphs – needed real extensions in comparison to Standard Z, which are implemented by the *ESZ* type checker. The experiment shows that – with the extensions used – Z is not only suitable for specifying sequential systems in the style found in the textbooks, but also for the complex axiomatic specifications as found in this thesis.

7.2 Future Work

Apart from minor enhancements of the basic model, its didactic presentation, a full-fledged implementation and case studies applying the implementation, the basic work presented in this thesis provides a starting point for further lines of research. The most important of these are outlined below.

Adding Solvers

For specialized constraint problems (e.g. subset constraints), tailored solving techniques and tools are available. The trend in constraint-solving systems is therefore to use *modular* frameworks, where components for specialized solvers can be easily integrated, on the semantical as well as the tool level. The aim of future work on μZ and its implementation will be to support such modularity, and integrate further resolution techniques.

On the implementation level, the thread-based architecture of the ZAM and its implementation in C++ is in principel prepared for modularity. An extension would need to convert intensions and threads into abstract classes, providing abstract methods for spawning a thread, executing a thread's next step, trailing, backtracking and choice point creation. This may allow third-party constraint solvers to be integrated, thereby inheriting the generic capabilities of the ZAM for representing constraints as first-class values, for choice point management and for variable allocation.

Static Analyses

The μZ calculus provides an optimal setting for investigating and realizing static analyses using the set-based approach. In his diploma thesis, Wieland [1999] has developed initial steps toward a subset constraint solver on μZ terms, which is applied to extended type checking of Z. Another application of this framework might be *determinism and mode analysis* for the optimized compilation of μZ itself. (As Brandes [1997] has shown in his diploma thesis, set-based analysis in principle can be used for this purpose.)

Both the extended type checking and the determinism problem are formulated as abstractions from μZ terms into (specialized) set expressions, which can be directly expressed in μZ itself. For example, for the type analysis, a constant expression 1 is abstracted into the expression $\{1\}$, denoting a singleton type. Future research might, then, focus on a reflective approach: an analysis of a μZ expression (and hence the high-level form it encodes) is described by a translation into a more abstract μZ expression, which is then interpreted by compilation and execution. To obtain the full power of set-based analysis, a solver component for subset constraints must, of course, added to the ZAM.

Computing Interval Logics

In [Büssow and Grieskamp, 1997; Büssow et al., 1997b] we described a modest extension of Z that adds *temporal interval logic* to the language. The addition of *traces* to a set-based model was shown to work well: a temporal formula merely represents a set of traces. A promising line of research would be to add a solver to μZ, implementing a few primitive operators of discrete temporal interval logics. One interesting application area is test data evaluation for reactive and embedded systems; a further is the execution of

watchdogs monitoring the allowed modal actions of safety- or security-critical systems. Since interval logics can be used as a natural model for the constructive description of reactive behavior (Büssow and Grieskamp [1997], for example, provide a shallow encoding of Statecharts in interval logics), the animation and prototyping of reactive systems might also be possible using this approach.

Bibliography

H. Ait-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction*. MIT Press, 1991. reprinted version from the author's WWW page.

H. Ait-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, San Francisco, 1987.

S. Antony. Definitional trees. In *Proc. 3rd International Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

S. Antony, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, 1994.

A. W. Appel. Garbage collection can be faster then stack allocation. *Information Processing Letters*, 25:275–279, 1987.

A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of POPL '89*. ACM, 1989.

P. Arenas-Sanchez and A. Dovier. A minimality study for set unification. *Journal of Functional and Logic Programming*, 1997(7), 1997.

H. Barendregt, editor. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1984. (revised edition).

M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

H. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume 28 of *SIGPLAN Notices*, pages 197–206, June 1993.

P. G. Bosco, E. C. Cecchi, and C. Moiso. An extension of the WAM for K-LEAF: a wam-based compilation of conditional narrowing. In *Proceedings of the Sixth International Conference on Logic Programming (Lisboa)*. MIT Press, 1989.

O. Brandes. Automatische Verfahren zur Ausführbarkeitsanalyse und Übersetzung der mengenbasierten Spezifikationssprache Z. Diplomarbeit, TU Berlin, Mai 1997.

R. Büssow, H. Dörr, R. Geisler, W. Grieskamp, and M. Klar. μSZ – ein Ansatz zur systematischen Verbindung von Z und Statecharts. Technical Report TR 96-32, Technische Universität Berlin, Jan. 1996.

R. Büssow, R. Geisler, W. Grieskamp, and M. Klar. The $\mu\mathcal{SZ}$ Notation Version 1.0. Technical Report 97–26, Technische Universität Berlin, Fachbereich Informatik, Dec. 1997a.

R. Büssow and W. Grieskamp. Combinig Z and Temporal Interval Logics for the Formalization of Properties and Behaviors of Embedded Systems. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science – Asian '97*, LNCS 1345, pages 46–56. Springer-Verlag, 1997.

R. Büssow and W. Grieskamp. A Modular Framework for the Integration of Heterogenous Notations and Tools. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proc. of the 1st Intl. Conference on Integrated Formal Methods – IFM'99*. Springer-Verlag, London, June 1999.

R. Büssow, W. Grieskamp, W. Heicking, and S. Herrmann. An Open Environment for the Integration of Heterogeneous Modelling Techniques and Tools. In *Proc. of Intl. Workshop on Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

R. Büssow, W. Grieskamp, F. Lattemann, and E. Lehmann. The Definition of Dynamic Z. ESPRESS interner Projektbericht. URL: http://uebb.cs.tu-berlin.de/~wg/papers/dynz.ps, May 1997b.

M. M. Chakravarty and H. C. R. Lock. Towards the uniform implementation of declarative languages. *Computer Languages*, 23(2-4), 1997.

M. M. T. Chakravarty, Y. Guo, M. Köhler, and H. C. R. Lock. GOFFIN: Higher-order functions meet concurrent constraints. *Science of Computer Programming*, 30(1-2), 1997.

M. M. T. Chakravarty and H. C. R. Lock. The implementation of lazy narrowing. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

R. Cleaveland, J. Gada, P. Lewis, S. Smolka, O. Sokolsky, and S. Zhang. The concurrency factory — practical tools for specification, simulation, verification, and implementation of concurrent systems. In *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, NJ, May 1994.

C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Language*, 1993.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of POPL '82*, pages 207–212. ACM, 1982.

B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge Mathematical Text Books. Cambridge University Press, 1990.

K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In J. Gutknecht, editor, *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 1994*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer, 1994. URL `http://uebb.cs.tu-berlin.de/papers/published/DesignImplOpal.ps.gz`.

K. Didrich, W. Grieskamp, C. Maeder, and P. Pepper. Programming in the large: the algebraic-functional language opal 2α. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages, St Andrews, Scotland, September 1997 (IFL'97), Selected Papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 323 – 338. Springer, 1998.

A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. $\{log\}$: A language for programming in logic with finite sets. *Journal of Logic Programming*, 28(1), 1996.

H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1985.

A. Fett and W. Grieskamp. Berlin proposals for the Z standard. `http://uebb.cs.tu-berlin.de/~wg/zstan/`, September 1998.

Formal Systems (Europe) Ltd. *Failures Divergence Refinement, FDR2 User Manual*. Formal Systems (Europe) Ltd., 1997.

T. Franzen, S. Haridi, and S. Janson. An overview of the andorra kernel language. In *Proceedings of the 2nd Workshop on Extensions to Logic Programming*. Springer Verlag, 1992.

T. Frauenstein, W. Grieskamp, P. Pepper, and M. Südholt. Communicating functional agents and their application to graphical user interfaces. In *Proceedings of the 2nd International Conference on Perspectives of System Informatics, Novosibirsk*, LNCS. Springer Verlag, Jun 1996a. URL `http://uebb.cs.tu-berlin.de/papers/published/PSI96-EA.ps.gz`.

T. Frauenstein, W. Grieskamp, and M. Südholt. Temporal Semantics of a Concurrency Monad with Choice and Services. In *Proceedings of the 2nd FUJI International Workshop on Functional and Logic Programming*, Nov 1996b. URL `http://uebb.cs.tu-berlin.de/papers/published/FUJI96.ps.gz`.

H. S. Goodman. The Z-into-Haskell tool-kit: An illustrative case study. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, volume 967 of *Lecture Notes in Computer Science*, pages 374–388. Springer-Verlag, 1995. ISBN 3-540-60271-2. URL `http://www.comlab.ox.ac.uk/archive/z/zum95.html`.

M. Gordon and T. Melham, editors. *Introduction to HOL: A theorem proving environment for higher-order logics*. Cambridge University Press, 1993.

W. Grieskamp, M. Heisel, and H. Dörr. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In E. Astesiano, editor, *Proc. of the 1st Intl. Conf. on Fundemantal Approaches to Software Engineering – FASE'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlang, 1998.

M. Hanus. Compiling logic programs with equality. In *Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming*, volume 456 of *Lecture Notes in Computer Sciene*. Springer-Verlag, 1990.

M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19(20), 1994.

M. Hanus. A unified computation model for declarative programming. In *Proc. of the 1997 Joint Conference on Declarative Programming*, Grado (Italy), 1997.

M. Hanus. Curry – an integrated functional logic language. Technical report, Internet, 1999. Language report version 0.5.

M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1), 1999.

M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(6), 1999.

D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16 No. 4, Apr. 1990.

P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, J. Kamperman, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "pseudoknot", a Float-Intensive benchmark. *J. of Functional Programming*, 6(4), 1996.

P. Hudak, S. P. Jones, P. Wadler, et al. Report on the programming language Haskell. *ACM SIGPLAN Notices*, 27(5), March 1992.

D. Jana and B. Jayaraman. Set constructors, finite sets, and logical semantics. *Journal of Functional Programming*, 38(1), 1999.

S. Janson. *AKL – A Multiparadigm Programming Language*. PhD thesis, Computer Science Departement, Uppsala University, Sweden, 1994.

S. Janson and S. 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.

B. Jayaraman and K. Moon. Subset logic programs and their implementation. *Journal of Logic Programming*, 19(20), 1999.

X. Jia. An approach to animating Z specifications. Internet: `http://saturn.cs.depaul.edu/~fm/zans.html`, 1996.

S. L. P. Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), 1992.

S. L. P. Jones and J. Salkild. The spineless tageless G-machine. In *Workshop on Implementation of Lazy Functional Languages*, Aspensas, Schweden, 1988.

G. Kahn. Natural semantics. In *Symposium on Theoretical Computer Science (STACS'97)*, volume 247 of *Lecture Notes in Computer Science*, 1987.

B. Krieg-Brückner, J. Peleska, E. Olderog, and A. Baer. The UniForM workbench, a universal development environment for formal methods. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*. Springer, 1999.

P. Kruchten, E. Schonberg, and J. Schwartz. Software prototyping using the SETL programming language. *IEEE Software*, 1(4), 1984.

H. Kuchen, R. Loogen, J. J. Moreno-Navarro, and M. Redriguez-Artalejo. Graph-based implementation of a functional logic language. In *Proceedings of European Symposium on Programming (ESOP '90)*, volume 432 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, 1964.

P. Larsen and W. Pawlowski. The formal semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5-6), 1995.

H. C. R. Lock. *The Implementation of Functional Logic Languages*. PhD thesis, Technische Universität Berlin, 1992.

R. Loogen and S. Winkler. Dynamic detection of determinism in functional logic languages. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

M. Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1999. submitted.

M. Mehl, R. Scheidhauer, and C. 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.

R. Milner, M. Tofke, and R. Harper. *The Definition of Standard ML*. Mass. Institute of Technology Press, Cambridge, Mass., 1990.

K. Moon. *Implementation of Subset Logic Languages*. PhD thesis, State University of New York at New Buffalo, 1997.

J. J. Moreno-Navarro, H. Kuchen, R. Loogen, , and M. Rodriguez-Artalejo. Lazy narrowing in a graph-machine. In *Proceedings of the Conference on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

J. J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12, 1992.

G. Nadathur and D. Miller. An overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*. MIT Press, 1988.

O. Owe. Partial logic reconsidered: A conservative approach. *Formal Aspects of Computing*, 5:208–223, 1997.

L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer LNCS 828, 1994.

N. Plat and P. Larsen. An overview of the ISO/VDM-SL standard. *SIGPLAN Notices*, 27 (8), 1992.

C. Prehofer. Higher-order narrowing. In *Proc. 9th Annual IEEE Symposium on Logic in Computer Science*. IEEE Press, 1994.

J. Reppy. CML: A higher-order concurrent language. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 293–305. ACM, Jun. 1991.

G. Rossi. The SETLOG programming language. Internet, 1997.

T. Santen. *A Mechanized Logical Model of Z and Object-Oriented Specification*. PhD thesis, Fachbereich Informatik, Technische Universität Berlin, Ger many, 1999.

B. Schätz and F. Huber. Integrating formal description techniques. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*. Springer, 1999.

R. Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine fr Oz*. PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, Dec. 1998.

W. Schulte. *Effiziente und korrekte Übersetzung strikter applikativer Programmiersprachen*. PhD thesis, Technische Universität Berlin, 1992.

W. Schulte and W. Grieskamp. Generating Efficient Portable Code for a Strict Applicative Language. In *Phoenix Seminar and Workshop on Declarative Programming*. Springer Verlag, Berlin, Heidelberg, New York, 1992.

J. Siekmann. Unification theory. *Journal of Symbolic Computation*, 7(1), 1989.

G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, Feb. 1994a.

G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, 7–9 Sept. 1994b. Springer-Verlag.

G. Smolka. Concurrent constraint programming based on functional programming. In C. Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 1381, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.

W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8, 1989.

J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992. ISBN 013-978529-9. URL `http://www.blackwell.co.uk/cgi-bin/bb_item?0139785299`.

L. Spratt. *Seeing the Logic of Programming with Sets*. PhD thesis, University of Kansas, 1996.

F. Stolzenburg. An algorithm for general set unification and its complexity. *Journal of Automated Reasoning*, 22(1), 1999.

I. Toyn. Innovations in standard Z notation. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 193–213. Springer-Verlag, 1998. URL `http://www.fmse.cs.reading.ac.uk/zum98/`.

S. Valentine. The programming language Z^{--}. *Information and Software Technology*, 37 (5–6):293–301, May–June 1995.

S. H. Valentine. Z^{--}, an executable subset of Z. Workshops in Computing, pages 157–187. Springer-Verlag, 1992. ISBN 3-540-19780-X. URL `http://www.dcs.gla.ac.uk/springer-verlag/21.html`.

R. L. Vaught. *Set Theory – An Introduction*. Birkhäuser, 1995.

P. Wadler and S. Blott. How to make ad–hoc polymorphism less ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, 1983.

D. H. D. Warren. The extended andorra model with implicit control. In *ICLP'90 Parallel Logic Programming Workshop*, 1990.

M. M. West and B. M. Eaglestone. Software development: Two approaches to animation of Z specifications using Prolog. *IEE/BCS Software Engineering Journal*, 7(4):264–276, July 1992.

J. Wieland. A resolution algorithm for general subtyping constraints. Master's thesis, Technische Universität Berlin, 1999.

M. Winikoff, P. Dart, and E. Kazmierczak. Rapid prototyping using formal specifications. In *Proceedings of the Australasian Computer Science Conference*, 1998.

M. Wirsing. *Handbook of Theoretical Computer Science*, chapter Algebraic Specification (13), pages 675–788. North-Holland, 1990. edited by J. van Leeuven.

J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996. ISBN 0-13-948472-8. URL `http://www.comlab.ox.ac.uk/usingz.html`.

Z-ISO. Drafts for the Z ISO standard. Ian Toyn (editor). Available via the URL `http://www.cs.york.ac.uk/~ian/zstan`, 1999.

Concept Index

λ-calculus, 39
μZ abbreviation forms, 39, 56
μZ forms, 38

AKL, 140
algebra, 41
Andorra Principle, 140

backtracking, 87, 93, 99, 140
benchmark, 136
binding, 48
boolean
  algebra, 58
  laws, 58
  values, 47
breadth-first search, 141

C++, 132
characteristic function, 43
choice point, 99, 103, 111
closure, 102
closure passing style, 138
committed choice, 70
common subexpression elimination, 95
compilation, 127
complete lattice, 35
configuration, 104
conjunctive normal form, 58
constraint, 41, 89, 101
constructor
  application form, 52
  decomposition laws, 60
continuous function, 36
cpo, 35
Curry, 41, 140
cycles, 102

deep guards, 97
definitional tree, 96
De Morgan's laws, 58, 75
depth-first search, 141
deterministic computation, 87
disjunctive normal form, 58, 75, 95
dispatching, 112

encapsulated search, 97
excluded middle, 44, 58
excluded middle laws, 59
execution step, 112
expression
  assignment, 57
  context, 57
  substitution, 57
  type, 50
expression sharing, 95
extensional
  completeness, 84
  confluence, 74, 81, 84
  equality, 73, 109
  unequality, 73

fixed-point
  construction, 68
  form, 55
  theorem, 36
  unrolling law, 66
flexible, 127
flexible variable, 76
freely generated, 46
free type, 46
free variables, 57
functional logic languages, 41

garbage collection, 136
goal, 102
Goffin, 41
graph reduction, 95

higher-order, 42
  unification, 97

HOL, 69
Horn clause, 41

indeterministic resolution, 87
infimum, 34
instantiation, 101, 106
instruction, 104, 113, 134
intension, 88, 101
interval logic, 70, 164

JUMP machine, 139

lattice, 35, 48
lazy narrowing, 85
logic languages, 41

matching context, 74
meaning function, 51
module, 41
Mozart System, 140

narrowing, 83, 95, 138
natural numbers, 47
needed narrowing, 96
normal form, 75
normalform, 127, 128
normalization, 75, 95
normalization function, 82

Opal, 138
outermost-in narrowing, 85
Oz, 70, 140

parallel and, 87
parallel or, 87
partial
- logic, 44
- order, 34, 48

pattern
- context, 57
- type, 50

persistent, 132
power set, 45
predicate calculus, 40
Prolog, 41

redex selection strategy, 96
reduction step, 89
reflective analysis, 163
register, 132
register transfer language, 134
residuation, 41, 87, 99
resolution step, 89

schema
- calculus, 40
- complement lifting law, 65
- distributivity laws, 64
- elimination law, 64
- form, 55
- membership lifting law, 65, 76
- property substitution law, 66
- translation elimination law, 65

SECD machine, 138
selection, 112
set
- intersection, 100, 109
- operator forms, 53
- operators, 43
- ordering, 48
- representation, 43
- selection elimination law, 59
- translation form, 53
- unification, 97
- union, 100, 109
- value, 100

set-based analysis, 163
SETL, 69
SETLOG, 69
single-assignment, 132
singleton
- set display form, 52
- set selection form, 52

soundness, 80, 84
spineless tagless G-Machine, 138
strict
- narrowing, 86
- reduction step, 89
- semantics, 87

strictness, 97
subset logic programming, 69, 97
supremum, 34
symbolic model checking, 95

temporal logic, 164
termination, 80

thread, 102
three-valued logic, 44
top-level program, 127
translation
    composition law, 60
    distributivity laws, 61
    elimination law, 63
    identity law, 60
    lifting law, 63
    product law, 62
    splitting law, 63
tuple, 48
type
    classes, 34
    relation, 50
    substitution, 45
    terms, 44
    unification, 45

undefinedness, 38, 39, 68
unification, 89, 109
universe, 45

value, 100
    constructor, 46
    form, 88
variable form, 55
variable table, 101

WAM, 138

Z extensions, 28
    ad-hoc overloading, 31
    Delta and Xi, 31
    free type recursion, 31
    if-exists notation, 28
    IF in predicates, 30
    inference rules, 28
    locals in schema boxes, 29
    paragraph order, 31
    recursive partial functions, 29
    redeclarations, 31
    selective delta, 30
    where notation, 28
Z language, 25
    constants, 26
    function abstraction, 26
    genericity, 26
    polymorphism, 26
    quantifiers, 26
    schemas, 25
    schema text, 25
    set comprehension, 26
    types, 25
    undefinedness, 26
    universe, 25
Z toolkit, 32
    Booleans, 33
    order and lattices, 34
    sequences, 32
    set reduction, 32

Symbol Index

LOADENV, 105, 114, 115, 129, 130
MKEMPTY, 105, 107, 118, 129, 134, 135
FIXINTEN, 134, 135
MKINTEN, 105, 107, 119, 130, 132, 134, 135
ISECT, 105, 107, 119, 120, 128–130, 134, 135
LOAD, 105, 113–116, 120, 129, 130
MEMBER, 105, 106, 111, 116, 117, 120, 130, 131, 134, 136
MKVAR, 105, 106, 118, 119, 130
MOVE, 134, 136
MU, 105, 107, 112, 121–123, 129, 130, 134, 136
$_ \vdash _ \cdot _$, 50–55
EXP_{UT}, 50–56
EXP_A, 75, 128, 130
EXP_B, 75, 83, 128, 130, 131
EXP_C, 75, 83, 128–131
EXP_D, 75, 83, 128, 130, 131
EXP_L, 75, 128, 129
EXP_P, 75, 83, 88, 130
EXP_V, 88, 90–93
PAT_{EX}, 50, 55, 60, 61, 79, 80
PAT_{LIN}, 50, 54, 55, 65
PAT_{UT}, 50, 51, 54, 55
$TSUBS_{\varnothing}$, 44, 45, 47, 51, 54
$TYPE_{\varnothing}$, 44, 45, 49
$\langle\!|\, _ \,|\!\rangle$, 48
$_(_)$, 18, 38, 50, 52, 60, 62, 64, 73, 75, 77, 86, 88, 128
$_$, 56, 60, 77
$_ \parallel _$, 88–93, 162
$_ \parallel _$, 88, 89, 92
fix $_ \lhd _$, 18, 38, 39, 41, 42, 48, 49, 55, 66, 68, 75, 83, 90, 102, 129, 130, 136, 137
$[\![_ \mid _]\!]$, 88, 90, 91, 93, 97, 100, 101, 120, 129, 162
$_ \overset{-}{\rightsquigarrow} _$, 54, 61, 62
$_ \rhd^{-} _$, 51, 54, 55, 61, 64, 66
$_ =^{-} _$, 57, 60–62, 64–66, 78, 80, 81, 83
$\{_ \mid _\}$, 18, 19, 37–43, 48, 55, 64–66, 68, 70, 75–81, 83–85, 88, 90, 95, 96, 98, 102, 129, 130, 136, 137
$\{_\}$, 18, 37–39, 41, 52, 56, 57, 59, 60, 62–66, 71, 73, 75–78, 81, 83–86, 88, 91, 93, 95–97, 100, 129, 130, 137
$_[_]$, 44–47
$?_{1}\,_$, 39, 40, 56, 75, 76, 80, 81, 131
$?_{0}\,_$, 56, 75, 91, 129, 131
$?_{_}\,_$, 93, 94
$_/_$, 61
$_[_ \mapsto _]$, 18, 19, 38–40, 42, 53, 54, 56, 57, 60–66, 75–81, 83, 85, 86, 91, 95, 96, 129, 130, 137
$_ \overset{\mapsto}{\rightsquigarrow} _$, 54, 61, 62
$\langle_\rangle$, 47, 48, 64
$\langle\rangle$, 47, 48, 64
$_ =^{-} _$, 45, 52–55
$(_)$, 48, 63, 80, 81
$_ \not\sim_{_} _$, 88, 89, 91
$_ \not\sim^{-}_{_} _$, 88, 89, 93
$_ \sim^{-} _$, 88, 89, 91
$_ \sim^{-}_{_} _$, 88–90, 93
Ω, 51–55, 66
LOADPAR, 105, 106, 120, 130
$\Delta[_]_$, 30, 111, 113–122, 124–126
MKSINGLE, 105, 107, 118, 129, 130, 134, 135
STORE, 105, 106, 115, 116, 130, 131
SUCCESS, 105, 107, 120, 121, 130, 131, 134, 136
MKTERM, 105, 106, 118, 119, 128, 134, 135
TEST, 105, 107, 112, 121, 122, 131, 134, 136
TESTNATIVE, 105, 107, 112, 121, 122, 130, 131, 134, 136
UNIFY, 105, 106, 115–118, 130, 131, 134, 136

UNION, 105, 107, 119, 120, 128, 129, 134, 135
WAIT, 105, 113, 115, 127, 129, 130, 134–136
$\mathbb{B}$, 47, 48, 55
$\bot$, 27, 109
$\bot$, 56
$\diamond$, 18, 19, 39, 44, 47, 48, 56, 60, 62–64, 77
γ, 127–131
$\mathbb{B}$, 33, 43, 45–49, 52, 55, 64, 109
false, 33, 43, 48, 49, 53, 54, 59, 61, 62, 64, 109
ff, 47, 48
true, 33, 43, 48, 49, 52, 53, 55, 59, 61, 62, 64, 109
tr, 54
tt, 47, 48, 55
undef, 45–48, 52, 55–57
$\equiv$, 18, 58–66, 68, 73, 84
$\sqsubseteq$, 34, 35, 48, 49, 56
$\sqsupseteq$, 34, 35
$\vdash$, 50–55
$=$, 39, 42, 48, 55, 56, 62, 63, 65, 75–77, 93, 95, 129–131, 136
$\in$, 19, 39–42, 56, 65, 70, 74–77, 80, 81, 83–85, 87, 91–94, 99, 100, 103, 104, 112, 120, 127, 129–131, 136, 162
0, 18, 38, 39, 53, 56, 58–60, 64–66, 75, 76, 78, 80, 81, 85, 86, 91–94, 96, 129, 131
1, 39, 40, 56, 58, 59, 64, 65, 75, 76, 78–81, 92–94, 131
$\mathbb{N}$, 47, 48
$*$, 33
$/$, 32, 33, 110
E, 57, 58, 80, 83, 85, 86
P, 57
SCTX, 74, 92
Γ, 50–55
$\backslash$, 32, 33, 129
$\cap$, 18, 19, 38–44, 48, 53, 56, 58–64, 66, 68, 70, 75–81, 84–86, 95, 96, 100, 129, 136, 137
$\cdot\!:$, 32, 33, 106, 107, 111, 114–122, 124–126
$\cup$, 18, 37, 38, 41–43, 53, 56, 58, 59, 61, 64, 65, 70, 71, 78–82, 91, 93, 95, 96, 98–100, 103, 136, 137, 162
$\downarrow_{\mathcal{S}}$, 45, 52–55
γ, 57, 60–62, 64–66, 76, 78, 80, 81, 83, 91, 93
ι, 51, 54, 55, 61, 64, 66
$\wedge$, 88, 91, 100, 120
$\vee$, 88, 93
$\mathcal{R}$, 85, 86
$\mathcal{S}$, 74, 92
$\mathcal{S}$, 86, 90
dfd, 27, 29
$\mathcal{C}_{\text{BAS}}$, 128, 130, 131
$\mathcal{C}_{\text{CON}}$, 128–131
$\mathcal{C}_{\text{DIS}}$, 128–131
$\mathcal{C}_{\text{LIT}}$, 128–130
$\mathcal{C}_{\text{PRO}}$, 128, 130, 131
arity, 44, 46
ass, 88, 89, 92
c, 46, 47
ftype, 46–48
$\mathsf{lat}_{\mathsf{C}}$, 35, 36, 49
lat, 35
mksubs, 57, 66, 83
nrm, 82–84, 90
sem, 46, 47, 52
sem, 51–55, 66
sem, 45, 47–49, 51–55, 64, 66
subs, 57, 60–63, 65, 66, 76, 80, 81, 83, 88, 89, 91, 93
subs, 44, 45, 47, 51–55, 66
subs, 51
tinst, 46, 52
type, 46
type, 50–55, 58, 66
varorder, 57, 80, 81, 127, 130
vars, 88, 89, 91, 93
vars, 38, 55, 57, 61–63, 65, 66, 76, 78–81, 83, 90, 91, 93, 130, 131
vars, 44–46, 51
zip, 32, 33, 110
$\not\sim$, 73, 78, 89, 91
$\sim$, 73, 74, 78, 81, 84, 89, 91, 92
μ, 18, 38, 42, 52, 53, 56, 57, 59, 60, 66, 75, 77, 85, 86, 92, 94, 95, 97, 103, 129, 130, 137, 162
$\mu_{\mathcal{E}}$, 92
$\mathbb{P}$, 45
$\mathbb{P}$, 44, 45, 47–49, 52–55
$\triangleright$, 88, 89, 91–93

$\sim$, 18, 19, 38–40, 44, 53, 56, 58, 59, 61, 64, 65, 70, 75, 78, 79, 84–86, 91, 129, 130
$\sqcap$, 35
$\sqcup$, 35
$\uparrow_{\mathcal{S}}$, 45, 46, 49, 52–55
$:\cdot$, 32, 33, 114–122, 124–126, 128–131
$\sqcap$, 34–36, 55
$\sqcup$, 34–36
$\dot{\bigcap}$, 89, 91
$\dot{\bigcap}$, 74, 78, 81–83, 85, 86
$\dot{\bigcup}$, 89, 91
$\dot{\bigcup}$, 74, 78, 85, 86
$\bigcap$, 56, 60, 74–77, 80, 91, 128, 129
$\bigcup$, 56, 73–75, 77, 88, 91, 97, 128, 129
$\bigvee$, 88, 89, 92, 93
$\bigwedge$, 88, 89, 92
$\bot_{-}$, 35, 36
$\sum_{-}^{-}$, 32
$\top_{-}$, 35
$\bigvee$, 33, 34, 54
$\bigwedge$, 33, 34

Binding, 103, 108, 110
Bound, 103, 105, 108, 110, 114, 115

Choice, 103, 111, 126
Code, 101, 103, 105, 111, 119, 128, 129
CONS, 38, 46–48, 50, 52, 60, 73, 77, 100, 105, 108–110, 119, 128
CTR, 88, 90–93

ENV, 127–131
error, 103, 106, 107, 115, 116, 119–121, 124–126
EXP, 18, 38, 50–66, 73–75, 77–83, 85, 88, 90–93, 128–131
EXPASS, 57, 60, 62, 64–66, 78, 80, 81, 83
EXPCTX, 57, 58, 83

Fail, 110, 116
failure, 103, 106, 107, 116, 117, 122–126
Free, 103, 115, 116, 118, 124, 126

Goal, 103, 105, 113, 125, 126
GoalInx, 101–103, 105, 115

INCTX, 74, 78, 81, 83, 89
Instruction, 101, 105, 112, 113
Intension, 100, 101, 109, 112, 117, 118, 120–122, 124–126
IVCTX, 88, 89, 91

Ok, 110, 116
OUTEREDEX, 85

PAT, 18, 38, 50, 51, 54, 55, 57, 60–66, 78–81, 83, 90, 91, 93, 130, 131
PATCTX, 57
Priority, 103

running, 103, 114, 118, 124, 126

SEMCTX, 51–55, 66
Set, 100, 106, 107, 109, 112, 117–122
SREDEX, 90
Status, 103
success, 103, 121, 123, 125

TCONS, 44, 46, 48
Term, 100–102, 106, 108–110, 118, 119
Thread, 103, 105, 113, 126
ThreadInx, 102, 103, 105, 111, 113, 118
TSUBS, 44, 45, 47, 51–55
TVAR, 44–47, 53
TYPASS, 50–55
TYPE, 44–47, 49, 51–55

UNCTX, 74, 78, 89
UNIASS, 51, 54, 55, 64, 66
UniRes, 110
UNIV, 45–47, 49, 51, 52, 54, 55, 61, 62
UVCTX, 88, 89, 91

VALASS, 88, 90–93
Value, 100, 101, 103, 108–112, 114, 115, 117, 118, 120–122, 124, 125
VAR, 38, 50, 51, 55–57, 60, 63, 77, 81, 88, 90–93, 127, 130, 131
Var, 100–102, 105, 106, 108–110, 114, 118–120
VarInx, 100, 101, 103, 105, 108–111, 114–116, 119, 127