Technische Universität Berlin
Fakultät IV (Elektrotechnik und Informatik)
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Übersetzerbau und Programmiersprachen
Franklinstr. 28/29
10587 Berlin

Diplomarbeit

Eine abstrakte Maschine für eine nebenläufige (und parallele) Constraint-funktionale Programmiersprache

Florian Lorenzen

13. November 2006

Gutachter: Prof. Dr. Peter Pepper
Dr. Petra Hofstedt

Betreuer: Dr. Petra Hofstedt
Martin Grabmüller

Zusammenfassung

In dieser Diplomarbeit wird eine nebenläufige constraint-funktionale Sprache und eine abstrakte Maschine als zugehöriges Ausführungsmodell entwickelt. Ebenso wird sowohl ein Compiler, um die Sprache in Code der abstrakten Maschine zu übersetzen, als auch ein Interpreter für die abstrakte Maschine implementiert.

Die Entwicklung nebenläufiger Programme ist schwierig und fehleranfällig. Aus diesem Grunde ist es wünschenswert, ein Programm auf dem höchstmöglichen Abstraktionsniveau formulieren zu können. Deklarative Sprachen eignen sich besonders zur Bearbeitung komplexer nebenläufiger Probleme, weil die Details der Kommunikation und des Prozessmanagements in der Sprachimplementierung verborgen sind. Da funktionale Sprachen insbesondere für transformationelle Algorithmen geeignet sind und sich in constraint-basierten Sprachen Abhängigkeiten und Einschränkungen gut formulieren lassen, besteht die Sprache, die im ersten Teil der Arbeit vorgestellt wird, aus zwei Komponenten: Einer Constraint-Sprache, die die Ausführung nebenläufiger Prozesse koordiniert, und einer nicht-strikten funktionalen Sprache, um die Berechnungen der einzelnen Prozesse zu definieren. Die Sprache unterstützt Abstraktionstechniken wie Funktionen höherer Ordnung, die Möglichkeit aus Koordinationsabstraktionen komplexere Koordinationsschemata zu komponieren und nicht-deterministische Programme. Ein wichtiger Gesichtspunkt beim Entwurf der Sprache, der sich auch in der Gestaltung der abstrakten Maschine niederschlägt, ist die Behandlung von nebenläufigen und parallelen Programmen auf einer gemeinsamen Basis. Parallele Programme werden dabei als nebenläufige Programme betrachtet, bei deren Ausführung gewisse Rahmenbedingungen bzgl. der Prozessorzuordnung garantiert werden. Sowohl Syntax als auch Semantik der Sprache werden vollständig formal spezifiziert sowie anhand von Beispielen erläutert.

Der zweite Teil befasst sich mit dem Entwurf der abstrakten Maschine zur nebenläufigen Abarbeitung constraint-funktionaler Programme, einer Graphreduktionsmaschine, erweitert um Instruktionen und Strukturen zur Ausführung von Prozessen und Behandlung von Constraints. Nach einer kurzen Einführung in Graphreduktion als Implementierungstechnik für nicht-strikte funktionale Sprachen werden die einzelnen Elemente der abstrakten Maschine wie Register, Speicherbereiche und Warteschlangen vorgestellt und die operationale Semantik der Maschine über den Instruktionssatz sowie zusätzliche Hilfsoperationen definiert. Die Maschine ist für Multiprozessor-Systeme entworfen, die nicht über gemeinsamen Speicher verfügen, funktioniert aber ebenso auf Einprozessormaschinen oder Mehrprozessorrechnern mit gemeinsamen Speicher. Dieser Teil schließt mit der Spezifikation von Compilations-Schemata zur Übersetung der constraint-funktionalen Sprache in Maschineninstruktionen.

Die Implementierung des Maschineninterpreters und des Compilers sowie Erfahrungen durch Testläufe auf einem Cluster werden im letzten Teil beschrieben.

Technische Universität Berlin
Fakultät IV (Elektrotechnik und Informatik)
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Übersetzerbau und Programmiersprachen
Franklinstr. 28/29
10587 Berlin

Diploma Thesis

An Abstract Machine for a Concurrent (and Parallel) Constraint Functional Programming Language

Florian Lorenzen

November 13, 2006

Supervisors: Prof. Dr. Peter Pepper
Dr. Petra Hofstedt

Advisors: Dr. Petra Hofstedt
Martin Grabmüller

Abstract

This diploma thesis develops a concurrent constraint functional language and an abstract machine as its execution model. A compiler to translate the language into abstract machine code and an interpreter for the abstract machine is also implemented.

Writing concurrent programs is a difficult and error-prone task. Therefore, it is desirable to use the most abstract formulation available. Declarative programming languages offer a suitable level of abstraction to handle complex concurrent problems because details of communication and process management are hidden in their implementation. As functional languages are very convenient to formulate transformational algorithms, and constraint based languages express dependencies and restrictions very easily, the language presented in the first part consists of two components: a constraint language which coordinates the execution of concurrent processes and a lazy functional language which defines the computations performed by the different processes. The thesis provides a complete formal specification of the concurrent functional language as well as several example programs.

The second part describes an abstract machine to concurrently evaluate constraint functional programs. It is a graph reduction machine extended with instructions and structures to handle constraints and processes. The machine is designed for multi-processor systems but also runs on a single processor. Like the input language, the machine semantics is completely specified as well as compilation schemes to translate the constraint functional language into abstract machine instructions.

The implementation of the machine interpreter and the compiler but also experiences with test runs on a cluster are described in the last part.

Acknowledgements

I have to thank Prof. Dr. Peter Pepper; without his excellent lecture on programming languages and systems I never had tried this joy ride in compiler construction and language design.
Many thanks go to Dr. Petra Hofstedt for giving me a free rein to follow my interest in parallel programming languages as well as numerous suggestions and corrections.
I thank Martin Grabmüller for many practical hints, uncovering several inconsistencies, and countless discussions about HASKELL, monadic-style programming, type systems, and other functional foo.
I also thank Katrin Lang and Fabian Otto for endless coffee-breaks in the i-Café while disussing the world of programming languages in general and of functional languages in particular and for reading parts of the manuscript.
Finally, I thank Janna Hennig for help with the English language.

Contents

1 Introduction **1**
1.1 Concurrency and its pitfalls 1
1.1.1 How to experience the pitfalls... 2
1.1.2 How to experience fewer pitfalls... 2
1.1.3 What this thesis is about 3
1.1.4 A word about parallelism and concurrency 4
1.2 Related work 4
1.2.1 EDEN 4
1.2.2 Concurrent Constraint Programming 5
1.2.3 GOFFIN 6
1.2.4 Others languages 7
1.3 Outline 7
1.4 Notations and general definitions 8

2 The core language FATOM **11**
2.1 Example programs 11
2.1.1 Producer-consumer settings 12
2.1.2 Master-slave setting 15
2.1.3 Divide and conquer setting 16
2.2 Syntax 18
2.2.1 Syntactic structure 19
2.2.2 Lexical structure 22
2.3 Context conditions 22
2.3.1 Scopes 22
2.3.2 Types 24
2.4 Semantics 28
2.4.1 Basic definitions 28
2.4.2 Ask and tell operations 30
2.4.3 Auxiliary functions and definitions 31
2.4.4 Transition relation 36
2.4.5 Example computations 39

3 The abstract machine ATAF **45**
3.1 Review of graph reduction 45
3.1.1 Lazy evaluation 46
3.1.2 Data structures for graph reduction 46
3.1.3 Finding the next redex 48
3.1.4 Instantiating a supercombinator 48

3.1.5 Evaluating built-in operators 49
3.1.6 Graph reduction of **let-** and **case**-expressions 50
3.2 Design of ATAF 51
3.2.1 The territory of processes 51
3.2.2 The machine memory 53
3.2.3 The process infrastructure 54
3.2.4 The constraint store and its periphery 56
3.3 Instruction set and operational semantics 59
3.3.1 Machine state 59
3.3.2 Process management and scheduling 61
3.3.3 Initial and final state 65
3.3.4 Instructions 66
3.4 Compiling FATOM to ATAF 77
3.4.1 Compiling constraint abstractions 79
3.4.2 Compiling functional abstractions 83
3.4.3 Bootstrapping the machine 86
3.4.4 Example compilation 87

4 Implementation **89**
4.1 `ataf` – the ATAF abstract machine interpreter 89
4.1.1 Overview of the implementation 89
4.1.2 The threads in detail 90
4.1.3 MPI and threads 91
4.1.4 Garbage collection 91
4.1.5 Startup and termination 93
4.1.6 Some options concerning performance 93
4.2 `fc` – a compiler for FATOM 93
4.2.1 Testsuite 94
4.3 Measuring performance 94
4.3.1 Speed-up of *pfarm* 95
4.3.2 Quicksort with granularity control 96

5 Conclusions **99**
5.1 Expressiveness and usability 99
5.2 Performance 99

A FATOM prelude **101**
A.1 `PreludeFun.fatom` 101
A.1.1 Function combination 101
A.1.2 Boolean functions 101
A.1.3 Functions on numbers 102
A.1.4 Pairs 103
A.1.5 List functions 103
A.2 `PreludeCoord.fatom` 108

B ATAF instruction set in alphabetical order **109**
B.1 Machine instructions 109
B.2 Internal operations 115

C Inference algorithm for supercombinator types **123**

D Manual of `fc` and `ataf` **125**
D.1 Installation 125
D.1.1 Availabilty 125
D.1.2 Requirements 125
D.1.3 Building and installing 125
D.1.4 Running the testsuite 126
D.2 Compiling FATOM programs 126
D.2.1 Command-line options 127
D.3 Running FATOM programs 128
D.3.1 Starting `ataf` 128
D.3.2 Command-line options 128
D.3.3 Output and result reporting 130
D.3.4 Memory usage 131
D.4 Documentation 131

E The names FATOM and ATAF **133**

Bibliography **135**

List of Figures

2.1 Producer-consumer-buffer communication 13
2.2 Call-tree of active *qsortGc* invocations 17
2.3 Syntax of FATOM programs 20
2.4 Syntax of expressions 20
2.5 Syntax of functional expressions 20
2.6 Syntax of constraint expressions 20
2.7 Syntax of guarded expressions 21
2.8 Lexemes of constants, variables, and numbers. 22

3.1 An overview of the memory architecture of ATAF 52
3.2 Running ATAF in parallel 53
3.3 Operations for queues 55
3.4 Operations for registers, stacks, and heaps 56
3.5 A complete machine instance 56
3.6 LOCK and UNLOCK instructions 58
3.7 Operation allocateVars 58
3.8 Constraint store and periphery 59
3.9 pushProc operation 63
3.10 Helper function copy 64
3.11 Operation set IP 65
3.12 Process state transition diagram 65
3.13 Instruction set of ATAF 68
3.14 Instruction SPAWN 69
3.15 Instruction ISPACK and operation lookup 69
3.16 Instruction ALLOC 69
3.17 Instruction SUSPEND 69
3.18 Instruction TELL 71
3.19 Instructions PUSHFUN, PACK, and PUSHVAR 71
3.20 Instructions MKAP, UPDATE 72
3.21 Instructions POP, SLIDE 73
3.22 Operation suspend 73
3.23 Instructions JUMP, CASEJUMP, SPLIT 73
3.24 Instruction UNWIND 74
3.25 Instruction EVAL 75
3.26 Instruction ADD 76
3.27 Instruction ERROR 76
3.28 Instruction NOPE 77

4.1 Instances and threads of `ataf` 90
4.2 Two `ataf` threads with the communication dispatcher 92
4.3 Speed-up of *pfarm* 95
4.4 Speed-up of *pfarm* 96
4.5 Speed-up of *quicksortGc* 97

List of Programs

2.1 Unbounded buffer communication 12
2.2 Bounded buffer communication 12
2.3 Startup of bounded buffer communication 13
2.4 Bounded buffer 14
2.5 Master-slave parallelisation 15
2.6 Parallel *map* 16
2.7 Parallel Quicksort 17
2.8 Parallel Mergesort 18
3.1 Functional example program to illustrate graph reduction 47
4.1 Calculating square roots 96

Chapter 1

Introduction

1.1 Concurrency and its pitfalls

There are two reasons to write concurrent computer programs, the first one is about convenience, the second one about performance:

1. Many problems and the algorithms to solve them are expressed more easily as independent tasks.

2. To tackle computationally demanding problems the use of parallel machines is an important option.

Both goals can only be fulfilled under the same precondition: a program must consist of parts which can be executed independently. For a purely concurrent program, it is of no importance whether these parts are executed in an interleaving fashion on a single processing element or at the same time on several of them. For parallel programs, it has to be ensured that different parts are executed at the same time by different processing elements to gain speed.

Of course, simple independent computations, called processes from now on, are of no use to help solving a particular problem. The need for some kind of cooperation arises. Cooperation comes in many flavours: several processes read the same set of data, one or more processes wait for data produced by another process or set of processes, many processes write to the same memory location etc.

Unfortunately, cooperation causes a lot of trouble: it implies an order of execution for different parts of the program, thus restricting the independence of execution. The reason is that e. g. a process working on data from another one has to wait until this data is produced. Looking the other way round, the producer has to be executed before any process requiring its outcome. If the program does not obey these restrictions it gets stuck. Another severe problem is that some data has to be manipulated atomically, i. e. processes must not be interrupted while working on this data, otherwise risking inconsistencies that may cause a program crash or incorrect results.

1.1.1 How to experience the pitfalls...

The traditional solution to these problem are variables shared among the processes and explicit synchronisation primitives in the programming language. These primitives come in different flavours like lock variables, semaphores, monitors, or mutexes. They all have in common that the programmer explicitly specifies under which conditions a certain piece of code may be executed, when other processes must be suspended from execution, or activated again. This is very much in the footprints of the imperative programming paradigm, a paradigm where the programmer guides the machine from one to the next state by explicit side effects, i. e. assignments to memory cells.

These very basic techniques are usually extended to message based cooperation, which frees the programmer of the tedious task to explicitly manage synchronisation between processes. In a message passing paradigm, processes may send and receive messages to and from each other, with a message being a certain interesting piece of data like a computational result or a request like a database query. Messages may be sent point-to-point from one process to exactly one recipient, broadcasted from one process to many receivers, or any combination of these. To complicate matters, there are several communication semantics like blocking and non-blocking send and receive operations, rendezvous-principle, message-buffering etc. Despite being a more abstract and thus more powerful programming technique, the developer still has to decide upon every single message and explicitly insert send and receive operations with the proper semantics into the code.

Apart from the method of cooperation, be it message or shared variable based, these techniques have one more disadvantage in common: processes are not anonymous for the simple reason that the programmer must explicitly state which process cooperates with what other process and how. Therefore, processes must be named or numbered. Names are assigned to processes on their creation, and in order for the programmer to grab such a name, processes also have to be created by hand; otherwise, one could not know when to grab such a name. The same manual treatment is true for process termination.

The summarise the above: in a traditional imperative concurrent system, the programming language requires explicit statements for cooperation, communication, and process management.

1.1.2 How to experience fewer pitfalls...

Declarative languages in contrast specify properties of the desired result of a computation, not the sequence of steps how to obtain the result. Functional and logic languages as well as constraint-based languages are examples for declarative programming languages. As they lack side-effects, they are well suited for concurrent execution because subexpressions are independent of each other. Unfortunately, experience shows that it is not feasible to try to exploit the entirety of this independence by concurrency because the overhead is much too large. The evaluation system needs some hints about the structure of the concurrency to achieve a compromise between expressiveness, comfort, and performance. It suggests itself to use the declarative paradigm to formulate the properties of the concurrency just like for the computation. With help of this second ingredient the system itself is able to figure out how processes interaction has to take place.

1.1.3 What this thesis is about

This is just the point where this thesis enters the scene. There are different possibilities how declarative concurrency can be expressed in a programming language, thus a particular choice has to be made. Starting with the next chapter, this thesis will develop a concurrent constraint functional language, called FATOM, an abstract machine ATAF as execution model for this language, and a compiler to translate from FATOM programs to ATAF machine instructions. The decision to chose a functional and constraint-based language is justified as follows:

Functional languages are very suitable for transformational problems, i. e. computations of any kind can be expressed concisely. Whereas constraint-based languages are very adapted to expressing dependencies, restrictions, and boundary conditions – in short, they are very convenient to express the properties of the concurrent structure of the processes. That is, this thesis follows a multi-paradigmatic approach while staying in the purely declarative domain.

There a few restrictions, though: the development and implementation of a complete fully-featured language plus an execution model simply cannot be done in half a year's time. Due to this time constraint, the language developed is to be understood as a core language as conveniently used in compilers as an intermediate representation. Of course, it contains all necessary features to write fully-fledged programs while still being human-readable and -writable. Several example programs and a prelude of commonly used functions are presented. The functional part of the language utilises a lazy reduction strategy, the constraint part consists of primitive built-in constraints, conjunctions of expressions, equality constraints, and guarded expressions. Its syntax and operational semantics is fully formally specified.

The execution model and the target architecture for the compiler is an abstract machine neatly adapted to the requirements of the language, allowing for a compilation as simple as possible, while still considerably narrowing the gap between the fairly abstract constraint-functional source code and real computing machines. Basically, this machine is a graph reduction machine enriched by some special instruction and data structures to handle constraints and multiprocessing. The machine is also fully specified, although in a slightly more semi-formal way. The design of an abstract machine as intermediate architecture is a well-known technique in programming language implementation: it allows for rapid prototyping by implementing an interpreter for the abstract machine to investigate the properties of the language and its execution model in a rather short period of time.

This is exactly the way this thesis heads: after the specification of the language and the machine a prototypical implementation is presented, and conclusions are drawn upon the experiences with this implementation.

The intended real target platform for FATOM and ATAF are parallel systems without shared memory, i. e. systems usually programmed with message passing. These include clusters, networks of workstations, but also tightly coupled multiprocessors. It is furthermore assumed that the program has been assigned a certain fixed set of processors before startup; as customary in most queueing and parallel computer management systems.

1.1.4 A word about parallelism and concurrency

There is one more contribution this thesis tries to make: how parallel and concurrent programs can be implemented very easily on a common footing.
The usual distinction between a concurrent and a parallel program is that concurrency may change the semantics, parallelism does not. Certainly, there is also the following difference: concurrency works on a single processing element, parallelism does not. But concurrency works on a multi-processor system just as good or bad as on a single-processor system. This is the point which offers the possibility to write parallel programs in a concurrent language, provided the following conditions hold:

1. a program must be able to know on how many processing element it is running, and
2. processes must be placed in a balanced way on the different processing elements.

As a parallel programmer, one is usually interested in a working load of 100 % per processing element to utilise maximal computational power, i. e. one process per processing element. If the concurrency properties can be formulated in such a way that as many runnable processes are created as processing elements are available (this can be achieved by help of condition 1) and these processes are placed one on each processing element (condition 2), this exactly is parallelism. The first condition is met by the developed core language FATOM and the second condition is to some approximation implemented in the execution model ATAF.

Before an outline of the remainder of this thesis is given, a brief overview of other approaches to the same problem is presented.

1.2 Related work

Numerous contributions have been made to the topic of concurrent (and parallel) declarative programming languages. This section only presents those that are most closely related to either the core language FATOM and its concepts or the abstract machine ATAF.

1.2.1 EDEN

EDEN [BLOP97, LOP05] is an extension of HASKELL, which introduces explicit process abstraction besides the usual functional abstraction. A process mapping values of type α to values of type β is a value of type $\mathit{Process}\ \alpha\ \beta$ and defined with the *process* function:

$$\mathit{process} :: (\alpha \rightarrow \beta) \rightarrow \mathit{Process}\ \alpha\ \beta$$

Processes are created by the infix operator #, e. g. $\mathit{process}\ (\lambda\ x \rightarrow e_1)\ \#\ e_2$ results in a new process to evaluate the expression $\lambda\ x \rightarrow e_1$ concurrently to the evaluation of e_2 which is done by the parent process.
Processes communicate via unidirectional channels, which are modelled as head-strict lazy lists, and are implicitly managed by the system. Nevertheless it is

possible for a process to create a new channel and send this name to another process, which may pass it on or send a reply via this channel. This feature is called dynamic channels and has been introduced to allow for direct connections between arbitrary processes, which sometimes reduces the communication overhead.

The following example of a Mergesort algorithm creates a binary tree network of processes to sort a list objects (the type constraint *Ord* α expresses that a total order has to be defined on α):

$$
\begin{aligned}
&\mathit{msort} \quad :: (\mathit{Ord}\ \alpha) \Rightarrow [\alpha] \rightarrow [\alpha] \\
&\mathit{msort}\ [] \ = [] \\
&\mathit{msort}\ \mathit{xs} = \mathit{smerge}\ (\mathit{process}\ \mathit{msort}\ \#\ \mathit{xs}_1)\ (\mathit{process}\ \mathit{msort}\ \#\ \mathit{xs}_2) \\
&\qquad \mathbf{where}\ (\mathit{xs}_1, \mathit{xs}_2) = \mathit{unshuffle}\ \mathit{xs}
\end{aligned}
$$

smerge and *unshuffle* are ordinary HASKELL functions to merge two lists preserving the order and to split a list into two halves.

Many-to-one communication is handled separately by a built-in non-deterministic *merge* process, which merges a sequence of input channels into a single output channel. As non-determinism contradicts referential transparency, *merge* may not be instantiated in functions in order to compute their results. Despite this weakness, *merge* is the only possibility to specify reactive and load-balancing systems.

The execution model for EDEN is the abstract machine DREAM (DistRibuted Eden Abstract Machine, [BKL+97]), which is an extension of the Spineless Tagless G-machine. Each Eden process runs in its own DREAM instance that in turn runs several independent threads of evaluation, one for each output channel. The threads use a common heap to enable updates and sharing of expressions. The input channels are also shared among all processes. The input language for DREAM is PEARL (parallel Eden abstract reduction language) that extends the STG-language with primitives for process abstraction, instantiation, and message passing.

1.2.2 Concurrent Constraint Programming

The idea to use constraints to coordinate the interaction of processes originally stems from Concurrent Constraint Programming or CCP [SR89, Sar93] for short. This short description follows the presentation in [HW06].

Concurrent constraint programming is based on the idea of monotonically refining partial knowledge. This knowledge is represented by a constraint store which is a set of constraints, each constraint representing some definite knowledge. Concurrently running processes may ask information from the store and continue execution depending on the answers obtained. They are also allowed to contribute new knowledge as long as the store does not become contradictory. A concurrent constraint program is a sequence of rules, each rule consisting of up to four parts:

$$\mathit{head} :\!-\ \mathit{ask} : \mathit{tell} \mid \mathit{body}.$$

The *head* is a single term identifying a clause, like a function or procedure name. The *ask* part is a sequence of constraints, which have to be fulfilled by the constraint store. If they are not fulfilled the process suspends until the

contents of the store has changed and the *ask* might be satisfied (blocking ask). If the *ask* part is satisfied the *tell* part is added to the store and the *head* is replaced by the *body* but only if it keeps the store consistent. If the *tell* part leads to a contradiction in the store another matching rule is tried.
An example for a CCP predicate is *merge*:

$$\begin{aligned}&\mathit{merge}(\mathit{XS}, \mathit{YS}, \mathit{ZS}) :-\\ &\quad \mathit{XS} = [\_,\_] : \mathit{XS} = [X|\mathit{XS}'], \mathit{ZS} = [X|\mathit{ZS}'] \mid \mathit{merge}(\mathit{XS}', \mathit{YS}, \mathit{ZS}').\\ &\mathit{merge}(\mathit{XS}, \mathit{YS}, \mathit{ZS}) :-\\ &\quad \mathit{YS} = [\_,\_] : \mathit{YS} = [Y|\mathit{YS}'], \mathit{ZS} = [Y|\mathit{ZS}'] \mid \mathit{merge}(\mathit{XS}, \mathit{YS}', \mathit{ZS}').\end{aligned}$$

merge non-deterministically merges two lists XS and YS. If one of the lists contains an element, the second being empty, only one rule is applicable and the first element of the list is inserted into the output variable ZS. The tail of the output variable ZS' is a fresh unbound variable that takes all further merged elements. If both ask parts are fulfilled, i. e. XS and YS contain at least one element, one of the rules is chosen non-deterministically.
merge can be part of a larger process network, it could e. g. join the output of two *produce* predicates:

$$?- \mathit{produce}(\mathit{Dat}_1), \mathit{produce}(\mathit{Dat}_2), \mathit{merge}(\mathit{Dat}_1, \mathit{Dat}_2, \mathit{Buf})$$

In this setting, there are three concurrent processes: two *produce* processes and one *merge* process.

1.2.3 GOFFIN

GOFFIN [Cha97, CGKL98, CGK98] is a constraint functional language built on top of the lazy functional language HASKELL. It is designed around the idea to have a computation and a coordination language that are strictly separated. The functional part of GOFFIN is the entire language HASKELL, the constraint part consists of equality constraints over ranked infinite trees, conjunctions, disjunctions and ask-expressions. GOFFIN makes use of the residuation principle, i. e. function application on logical variables only takes place if the variables are instantiated. As plain HASKELL, it is a strongly typed language and constraints have the type O. Functions with type $\alpha \to O$ are called constraint abstractions and are first-class citizens, so they may be passed around and combined to more complex abstractions.
An example for such a constraint abstraction is *pipe* which takes a list of constraint abstractions and pipes an input value through all of these abstractions:

$$\begin{array}{lll}\mathit{pipe} & :: & [\alpha \to \alpha \to O] \to \alpha \to \alpha \to O\\ \mathit{pipe}\ []\ \mathit{input}\ \mathit{output} & \Rightarrow & \{\mathit{output} \leftarrow \mathit{input}\}\\ \mathit{pipe}\ (c : \mathit{cs})\ \mathit{input}\ \mathit{output} & \Rightarrow & \{\mathit{link}\ \mathbf{in}\ c\ \mathit{input}\ \mathit{link}, \mathit{pipe}\ \mathit{cs}\ \mathit{link}\ \mathit{output}\}\end{array}$$

Constraints are enclosed in {, }, and new logical variables are introduced before the **in** keyword. Equality constraints are expressed by a left-headed arrow $\leftarrow$. Ask-expressions are separated from the body by a bigger arrow $\Rightarrow$ and act as guards: the right-hand-side of the arrow is only evaluated if the guard can be satisfied. Guards also open the possibility for non-determinism by specification of overlapping patterns.

Like in CCP, conjunction, indicated by a comma, is an abstract notion for parallel composition, i. e. the two expressions *c input link* and *pipe cs link output* are evaluated in parallel.

1.2.4 Others languages

More declarative languages based on process coordination by constraints include

- concurrent PROLOG dialects [Sha89] that use shared logical variables as communication and synchronisation objects,
- PARLOG [CG86] which incorporates and- and or-parallel modes into PROLOG, and
- LMNtal [UK05] which uses logical links to represent communication and data structures.

Many functional languages also support concurrent or parallel programming, some examples are

- CONCURRENT HASKELL [PGF96] that supports threads via the IO monad,
- GLASGOW PARALLEL HASKELL [THLP98] which extends HASKELL by strategy combinators for parallel evaluation, and
- parallel OPAL [Nit05], a skeleton-based extension of the strict, purely functional language OPAL [DFG+94].

1.3 Outline

This thesis consists of four additional chapters that shall be described briefly:

Chapter 2 This chapter develops the core language FATOM. It starts with an informal introduction on the basis of several example programs of different complexity. It proceeds with a formal specification of the language's syntax and context-conditions and concludes with the full specification of its operational semantics.

Chapter 3 In this chapter, the abstract machine is developed. Before going into details of the machine, graph reduction as implementation technique is reviewed briefly. The description of the machine and its operational semantics is divided into two parts, the first outlining the structure and different components of the machine, the second defining its instruction set. The last section presents a translation function for FATOM programs into ATAF machine instructions.

Chapter 4 The fourth chapter gives a brief description of the machine's and compiler's prototypical implementation. It ends with a short outlook concerning performance behaviour on a parallel machine.

Chapter 5 This chapter concludes and discusses some points of further interest.

1.4 Notations and general definitions

This section introduces some notations, sets and functions that are used throughout the following chapters. More formalisms are defined more locally at the point they are required.
A partial function f from set A to set B is denoted by $f : A \hookrightarrow B$.
An n-tuple of a set A is written as A^n:

$$A^n := \underbrace{A \times \cdots \times A}_{n \text{ times}}$$

An element of a cartesian product, i. e. a tuple, is enclosed by angular brackets:

$$\langle a_1, \ldots, a_n \rangle \in A_1 \times \cdots \times A_n \quad \text{for } a_1 \in A_1, \ldots, a_n \in A_n$$

The empty typle $a \in A^0$ is written as $\langle\rangle$.
Some definitions in Chapter 2 use the power-set $\mathbb{P}A$ of a set A:

Definition 1.1 *(Power-set)*
The power-set *of a set A is the set of all subsets of A.*

$$\mathbb{P}A :- \{B \mid B \subseteq A\}.$$

The next definitions introduce three more complex sets along with operations on them. These are arrays, lists, and environments, which are very similar to data types of the same name provided by many programming languages.
Arrays are used in the definition of the abstract machine ATAF in Chapter 3.

Definition 1.2 *(Array)*
An array *is a mapping from a finite index set $I \subset \mathbb{N}$ to some set α:*

$$ARRAY_I\, \alpha := I \rightarrow \alpha$$

If the index set is a sequence of natural numbers $\{a, a+1, \ldots, b-1, b\}$ with $a, b \in \mathbb{N}$ the following abbreviation may be used:

$$ARRAY_a^b\, \alpha := ARRAY_{\{a,a+1,\ldots,b-1,b\}}\, \alpha$$

The lookup of a certain field in an array $\mathtt{A} \in ARRAY_I\, \alpha$ is often written with brackets instead of parentheses: $\mathtt{A}[k] := \mathtt{A}(k)$ for $k \in I$.
To create an array $\mathtt{A} : ARRAY_a^b\, \alpha$ with all elements set to to a value $v \in \alpha$, i. e. $\mathtt{A}[k] = v$ for all $k = a, \ldots, b$, the notation

$$[v]_a^b$$

is used.
Arrays can be updated at an index k by an assignment:

$$(\mathtt{A}[k] := v)[i] := \begin{cases} v & \textit{for } k = i \\ \mathtt{A}[i] & \textit{otherwise} \end{cases}$$

Lists are used in the definition of ATAF but also in the compilation schemes of Section 3.4.

Definition 1.3 *(List)*
A list *containing elements of some set α is defined inductively:*

$$[] \in LIST\ \alpha$$
$$a \in \alpha, as \in LIST\ \alpha \Rightarrow a : as \in LIST\ \alpha$$

List construction by : *associates to the right and may be abbreviated:*

$$[a_1, \ldots, a_n] := a_1 : \cdots : a_n : []$$

Concatenation of two lists is written with the $+\!\!+$ *operator:*

$$[a_1, \ldots, a_m] +\!\!+ [b_1, \ldots, b_n] := [a_1, \ldots, a_m, b_1, \ldots, b_n]$$

If as is a list, $|as|$ *is the length of the list:*

$$|[]| :- 0$$
$$|a : as| :- 1 + |as|$$

Environments appear in the compilation schemes as well as in the definition of FATOM's scopes in Section 2.3.

Definition 1.4 *(Environment)*
An environment *is a partial mapping from some set V to some set α:*

$$ENV_V^\alpha := V \hookrightarrow \alpha$$

Two environments Γ_1*,* Γ_2 *can be nested with the dot operator:*

$$(\Gamma_1.\Gamma_2)\,(v) := \begin{cases} a & \text{if } \Gamma_2(v) = a \\ \Gamma_1(v) & \text{otherwise} \end{cases}$$

An initial environment which knows about the values $v_1, \ldots, v_n$ *is set up by*

$$[v_1 \mapsto a_1, \ldots, v_n \mapsto a_n](v) := a_i \quad \text{if } v = v_i,\ i = 1, \ldots, n.$$

If an environment maps to the empty set, i. e. $ENV_V^\emptyset$*, construction may be abbreviated:*

$$[v_1, \ldots, v_n] := [v_1 \mapsto \emptyset, \ldots, v_n \mapsto \emptyset]$$

Transforming the value domain of an environment with a function f is written like this:

$$\Gamma^f := [v_1 \mapsto f(a_1), \ldots, v_n \mapsto f(a_n)] \quad \text{for } \Gamma = [v_1 \mapsto a_1, \ldots, v_n \mapsto a_n]$$

The number of entries in an environment Γ *is written as* $|\Gamma|$*:*

$$|\Gamma| := n \quad \text{for } \Gamma = [v_1 \mapsto a_1, \ldots, v_n \mapsto a_n]$$

Multisets are essentially sets that may contain one element several times. They are used in the specification of the operational semantics of FATOM and are formally defined as follows:

Definition 1.5 *(Multiset)*
A multiset *M is a pair (A, m_A) with A being a set and m_A a function $A \rightarrow \mathbb{N} \setminus \{0\}$.*
An element a is a member of a multiset $M = (A, m_A)$ if $a \in A$, which is also written as $a \in M$.
Two multisets $M = (A, m_A)$ and $N = (B, m_B)$ can be joined with the $\uplus$ operator:

$$M \uplus N := (A \cup B, m_{AB})$$

with

$$m_{AB}(x) := \begin{cases} m_a(x) + m_b(x) & \text{for } x \in A \text{ and } x \in B \\ m_a(x) & \text{for } x \in A \text{ and } x \notin B \\ m_b(x) & \text{for } x \notin A \text{ and } x \in B. \end{cases}$$

Subtraction of two multisets $M = (A, m_A)$ and $N = (B, m_B)$ is defined as

$$M \setminus N := (C, m_c)$$

with

$$C := \{x \mid x \in A \text{ and } (x \notin B \text{ or } m_A(x) > m_B(x))\}$$

and

$$m_C(x) := \begin{cases} m_A(x) & \text{for } x \notin B \\ m_A(x) - m_B(x) & \text{for } m_A(x) > m_B(x). \end{cases}$$

Elements of a multiset may be enumerated in curly brackets:

$$\{ \underbrace{a_1, \ldots, a_1}_{m_A(a_1) \text{ times}}, \ldots, \underbrace{a_n, \ldots, a_n}_{m_A(a_n) \text{ times}} \} := (\{a_1, \ldots, a_n\}, m_A)$$

Chapter 2

The core language FATOM

This chapter develops the language FATOM.
FATOM is intended as target language for high level language compilers. For this reason, FATOM is very simple and the design focus is on implementability and not on elegance or expressive power.
The language is composed of two parts:

- the computational core, which is a weakly typed lazy functional language, and
- the coordination core, which is a simple constraint language including guarded expressions, equality constraints and conjunctions.

The functional part is a subset of the language developed in [LP92], the coordination part resembles some of the aspects of the language GOFFIN [CGKL98, CGK98] and the primitive constraints are inspired by the type constraints of LMNtal [UK05].
The remainder of this chapter is structured as follows:

- Section 2.1 presents a few simple to medium complex example programs along with an intuitive explanation of their meaning.
- The next two Sections 2.2 and 2.3 deal with the language syntax – the first is about the context-free parts and the latter specifies partial context-conditions for types and scopes.
- Finally, Section 2.4 defines FATOM's operational semantics by a transition relation.

2.1 Example programs

The example programs of this section include classical concurrent settings of cooperating processes as well as different parallelisation schemes.
FATOM is not type safe and it is easy to write type incorrect programs. Nevertheless, type correctness is of course required. For documentation purposes, function types are annotated in comments using HASKELL syntax.
Some examples make use of the FATOM prelude defined in Appendix A.

2.1.1 Producer-consumer settings

Producer-consumer communication is a classical interaction scheme of concurrent processes. This section introduces two examples with different complexity.

Unbounded buffer Listing 2.1 shows two processes communicating via an unbounded buffer *buf* (a list). The *consumer* checks if *buf* is bound to a *Cons*, which is expressed by the guarding constraint *pack buf* 2 2. If this is not the case, it is delayed until the *producer* has stored at least one *item* (a 0 in this case) in the buffer. If the buffer is filled its head is dropped (*consume* could equally well do some productive work on it) and the tail is consumed. (This example is taken from [HW06].)

Listing 2.1 Producer-consumer communication via an unbounded buffer.

$$
\begin{array}{l}
--\ \mathit{produce} :: \alpha \rightarrow [\alpha] \rightarrow C \\
\textbf{def}\ \mathit{produce\ item\ buf} = \textbf{with}\ \mathit{buf}'\ \textbf{in}\ \mathit{buf} =:= \mathit{Cons\ item\ buf}'\ \&\ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{produce\ item\ buf}' \\
--\ \mathit{consume} :: [\alpha] \rightarrow C \\
\textbf{def}\ \mathit{consume\ buf} = \mathit{pack\ buf}\ 2\ 2 \Rightarrow \textbf{with}\ \mathit{buf}'\ \textbf{in}\ \mathit{buf}' =:= \mathit{tl\ buf}\ \&\ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{consume\ buf}' \\
--\ \mathit{prodCon} :: C \\
\textbf{def}\ \mathit{prodCon} = \textbf{with}\ \mathit{buf}\ \textbf{in}\ \mathit{produce}\ 0\ \mathit{buf}\ \&\ \mathit{consume\ buf}
\end{array}
$$

Listing 2.2 Producer-consumer communication via a bounded buffer.

$$
\begin{array}{l}
--\ \mathit{produce} :: \alpha \rightarrow \mathit{Msg} \rightarrow \mathit{Msg} \rightarrow [\alpha] \rightarrow C \\
\textbf{def}\ \mathit{produce\ item\ to\ from\ outp} = \\
\quad \textbf{with}\ \mathit{outp}'\ \mathit{from}'\ \textbf{in}\ \mathit{outp} =:= \mathit{Cons\ item\ outp}'\ \&\ \\
\qquad\qquad\qquad\qquad\quad \mathit{from} =:= \mathit{Msg\ from}'\ \&\ \\
\qquad\qquad\qquad\qquad\quad \mathit{produceWait\ item\ to\ from}'\ \mathit{outp}' \\
--\ \mathit{produceWait} :: \alpha \rightarrow \mathit{Msg} \rightarrow \mathit{Msg} \rightarrow [\alpha] \rightarrow C \\
\textbf{def}\ \mathit{produceWait\ item\ to\ from\ outp} = \\
\quad \mathit{pack\ to}\ 1\ 1 \Rightarrow \textbf{with}\ \mathit{to}'\ \textbf{in}\ \mathit{to}' =:= \mathit{msg\ to}\ \&\ \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{produce\ item\ to}'\ \mathit{from\ outp} \\
\\
--\ \mathit{consumeWait} :: \mathit{Msg} \rightarrow \mathit{Msg} \rightarrow [\alpha] \rightarrow C \\
\textbf{def}\ \mathit{consume\ to\ from\ inp} = \\
\quad \textbf{with}\ \mathit{from}'\ \textbf{in}\ \mathit{from} =:= \mathit{Msg\ from}'\ \&\ \\
\qquad\qquad\qquad\quad \mathit{consumeWait\ to\ from}'\ \mathit{inp} \\
--\ \mathit{consumeWait} :: \mathit{Msg} \rightarrow \mathit{Msg} \rightarrow [\alpha] \rightarrow C \\
\textbf{def}\ \mathit{consumeWait\ to\ from\ inp} = \\
\quad \mathit{pack\ to}\ 1\ 1 \Rightarrow \textbf{with}\ \mathit{to}'\ \mathit{inp}'\ \textbf{in}\ \mathit{inp}' =:= \mathit{tl\ inp}\ \&\ \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{to}' =:= \mathit{msg\ to}\ \&\ \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{consume\ to}'\ \mathit{from\ inp}'
\end{array}
$$

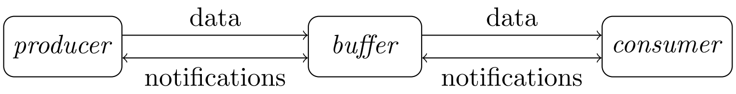


Figure 2.1 Producer-consumer communication via a bounded buffer.

Bounded buffer This example is a more complicated setup than the unbounded buffer program. The key idea is that two processes, a *producer* and a *consumer* again, communicate via a buffer that has a limited capacity. This results not only in delaying consumption on an empty buffer, but also in suspending production if the buffer has reached its capacity limit.
To achieve synchronisation on the limited resource of buffer space, the buffer itself is implemented as a process, which is placed between the *producer* and *consumer* as shown in Figure 2.1 with the following communication structure:

- The *buffer* waits for either the *producer* to deliver an item or the *consumer* to request an item, implemented by the two guarded branches (1) and (2) in Listing 2.4. Delivery and request are communicated via an additional bidirectional notification channel. If the buffer is full insertion of an element into the buffer is delayed until the *consumer* requests an item. Likewise, if the buffer is empty a request of the *consumer* is suspended until an item has been produced.
- The *producer* sends an item to the *buffer* and waits for an acknowledgement in *produceWait* that it has been inserted into the buffer before the next item is produced (Listing 2.2).
- The *consumer* requests an item from the *buffer* and waits for its delivery in *consumeWait*.

The startup of the three processes is shown in Listing 2.3. The capacity limit of the buffer is set to *max* and the buffer is initially empty. The notification channels are *bp*, *pb* for *buffer* $\leftrightarrow$ *producer* and *bc*, *cb* for *buffer* $\leftrightarrow$ *consumer*. The data items are passed via *inp* and *outp*.
In this cooperation, the buffer process acts as a buffering service for two clients. The buffer can also be considered a *Service Access Point* (cf. [PH06]) where one producer offers its service and one consumer request this service. By extensions of the coordination part, the Service Access Point could also handle several producers and consumers and different servicing strategies like first come, first serve or priorities.

Listing 2.3 Startup of bounded buffer communication.

```
-- prodCon :: α → Nat → C
def prodCon item max =
  with full empty bp pb inp bc cb outp in empty =:= True              &
                                          buffer max 0 Nil full empty
                                                 bp pb inp bc cb outp &
                                          produce item bp pb inp      &
                                          consume bc cb outp
```

Listing 2.4 A bounded buffer process.

```
-- buffer :: Nat → Nat → [α] → Bool → Bool → [Msg] → [Msg] →
-- [α] → [Msg] → [Msg] → [α] → C
def buffer max fill buf full empty toP fromP inp toC fromC outp =
      bound fromP & unbound full ⇒      -- (1)
         with empty′ toP′ fromP′ inp′
         in let fill′  = succ fill;
                 buf′  = Cons (hd inp) buf
            in inp′ =:= tl inp           &
               fromP′ =:= msg fromP &
               case fill′ < max of
                  {1} → toP =:= Msg toP′ &
                           (case fill == 0 of
                              {1} → buffer max fill′ buf′ full empty′ toP′
                                          fromP′ inp′ toC fromC outp;
                              {2} → buffer max fill′ buf′ full empty toP′
                                          fromP′ inp′ toC fromC outp);
                  {2} → full =:= True &
                           (case fill == 0 of
                              {1} → buffer max fill′ buf′ full empty′ toP′
                                          fromP′ inp′ toC fromC outp;
                              {2} → buffer max fill′ buf′ full empty toP′
                                          fromP′ inp′ toC fromC outp)
    | bound fromC & unbound empty ⇒ -- (2)
         with full′ toC′ fromC′ outp′
         in let fill′  = pred fill;
                 buf′  = butlast buf;
                 item = last buf
            in outp =:= Cons item outp′ &
               toC =:= Msg toC′          &
               fromC′ =:= msg fromC    &
               case fill′ > 0 of
                  {1} → (case fill == max of
                              {1} → buffer max fill′ buf′ full′ empty toP
                                          fromP inp toC′ fromC outp′;
                              {2} → buffer max fill′ buf′ full empty toP
                                          fromP inp toC′ fromC outp′);
                  {2} → empty =:= True &
                           (case fill == max of
                              {1} → buffer max fill′ buf′ full′ empty toP
                                          fromP inp toC′ fromC outp′;
                              {2} → buffer max fill′ buf′ full empty toP
                                          fromP inp toC′ fromC outp′)


-- data Msg = Msg Msg
-- Msg :: Msg → Msg
def Msg m = Pack {1,1} m


-- msg :: Msg → Msg
def msg m = case m of {1} m′ → m′
```

2.1.2 Master-slave setting

Master-slave cooperation can be considered a generalisation of producer-consumer cooperation described in the previous section.

It is an important parallelisation scheme for data-parallel applications.

In this simple example, two slave processes *allOdds* and *allEvens* produce a list of numbers and send them to a master process *sum*, which sums up the first n numbers of each slave process (see Listing 2.5).

Listing 2.5 Master-slave parallelisation with two slaves.

$$
\begin{array}{l}
-- \text{ Produces the list } [m, m+n, m+2n, m+3n, \ldots]. \\
-- \ \mathit{stepFromBy} :: \mathit{Nat} \to \mathit{Nat} \to [\mathit{Nat}] \\
\mathbf{def}\ \mathit{stepFromBy}\ m\ n = \mathit{Cons}\ m\ (\mathit{stepFromBy}\ (m+n)\ n) \\
\\
-- \ \mathit{allEvens} :: [\mathit{Nat}] \\
\mathbf{def}\ \mathit{allEvens} = \mathit{stepFromBy}\ 0\ 2 \\
\\
-- \ \mathit{allOdds} :: [\mathit{Nat}] \\
\mathbf{def}\ \mathit{allOdds} = \mathit{stepFromBy}\ 1\ 2 \\
\\
-- \ \mathit{sum} :: \mathit{Nat} \to \mathit{Nat} \to C \\
\mathbf{def}\ \mathit{sum}\ n\ s = \mathbf{with}\ \mathit{nums}\ \mathit{evens}\ \mathit{odds} \\
\qquad\qquad\quad \mathbf{in}\ \mathit{evens} =:= \mathit{take}\ n\ \mathit{allEvens}\ \&\ \mathit{odds} =:= \mathit{take}\ n\ \mathit{allOdds}\ \& \\
\qquad\qquad\qquad \mathit{merge}\ \mathit{evens}\ \mathit{odds}\ \mathit{nums}\ \&\ s =:= \mathit{foldl}\ \mathit{add}\ 0\ \mathit{nums} \\
\\
-- \ \mathit{merge} :: [\alpha] \to [\alpha] \to [\alpha] \to C \\
\mathbf{def}\ \mathit{merge}\ l1\ l2\ r = \mathit{pack}\ l1\ 1\ 0\ \& \\
\qquad\qquad\qquad\quad \mathit{pack}\ l2\ 1\ 0 \Rightarrow r =:= \mathit{Nil} \\
\qquad\qquad\qquad \mid \mathit{pack}\ l1\ 2\ 2 \Rightarrow \mathbf{with}\ \mathit{rs}\ l1' \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ r =:= \mathit{Cons}\ (\mathit{hd}\ l1)\ \mathit{rs}\ \& \\
\qquad\qquad\qquad\qquad\qquad\quad l1' =:= \mathit{tl}\ l1\ \&\ \mathit{merge}\ l1'\ l2\ \mathit{rs} \\
\qquad\qquad\qquad \mid \mathit{pack}\ l2\ 2\ 2 \Rightarrow \mathbf{with}\ \mathit{rs}\ l2' \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ r =:= \mathit{Cons}\ (\mathit{hd}\ l2)\ \mathit{rs}\ \& \\
\qquad\qquad\qquad\qquad\qquad\quad l2' =:= \mathit{tl}\ l2\ \&\ \mathit{merge}\ l1\ l2'\ \mathit{rs}
\end{array}
$$

The important point is that *allOdd*'s and *allEvens*'s outputs are merged non-deterministically, which means that the computation smoothes over differences in computation speed of the two slave processes. This cannot be achieved easily by deterministic merging (like the prelude's *amerge*).

The non-determinism results from the two guards *pack l1* 2 2 and *pack l1* 2 2 which wait for *l1* or *l2* to be bound to a *Cons*. If both guards are satisfied one branch is chosen non-deterministically. If none of the guards is fulfilled the *merge* process suspends. Merging terminates as soon as both lists are the empty list *Nil*.

2.1.3 Divide and conquer setting

Divide and conquer algorithms are very suitable for parallelisation. Two examples are given in this paragraph:

- A parallel version of the *map*-functional, with (*pfarm*) and without (*farm*) granularity control.
- Both, Mergesort and Quicksort, are divide and conquer algorithms and can easily be expressed as parallel processes.

farm: a parallel map The *farm* functional spawns one process for each application of the given function f.
It is useful to limit the number of processes to the number of processing elements available ($noPE$), which is done by *pfarm*.
Listing 2.6 shows *farm* and *pfarm*. The functions *partition* and *map* are defined in the FATOM prelude in Appendix A.

Listing 2.6 Parallel *map* functions.

$$
\begin{array}{l}
--\ \mathit{farm} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow C \\
\mathbf{def}\ \mathit{farm}\ f\ l\ r = \\
\quad \mathbf{case}\ l\ \mathbf{of} \\
\qquad \{1\} \qquad\ \ \rightarrow r =:= \mathit{Nil}; \\
\qquad \{2\}\ x\ xs \rightarrow \mathbf{with}\ rs\ \mathbf{in}\ r =:= \mathit{Cons}\ (f\ x)\ rs\ \&\ \mathit{farm}\ f\ xs\ rs
\end{array}
$$

$$
\begin{array}{l}
--\ \mathit{pfarm} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow C \\
\mathbf{def}\ \mathit{pfarm}\ f\ l\ r = \mathit{nfarm}\ \mathit{noPE}\ f\ l\ r
\end{array}
$$

$$
\begin{array}{l}
--\ \mathit{nfarm} :: \mathit{Nat} \rightarrow (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow C \\
\mathbf{def}\ \mathit{nfarm}\ n\ f\ l\ r = \mathbf{let}\ \mathit{parts} = \mathit{partition}\ n\ l; \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{pf} \quad\ = \mathit{map}\ f \\
\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathbf{with}\ rs\ \mathbf{in}\ \mathit{farm}\ \mathit{pf}\ \mathit{parts}\ rs\ \&\ r =:= \mathit{concat}\ rs
\end{array}
$$

Sorting networks Listing 2.7 shows two different versions of the Quicksort algorithm:

- *quicksort* spawns a new sorting process for each recursive invocation.
- *quicksortGc* switches to serial sorting as soon as the number of active invocations of *qsortGc* has reached the number of processing elements. The total number of processes alive during sorting is greater than the number of processing elements but some of them are suspended.

  Each new invocation of *qsortGc* is passed the number of subsequent processes it may spawn. If this is less than two, evaluation continues with a serial sorting function.

  Figure 2.2 shows the invocation of active *qsortGc* instances for eight and five processing elements. The nodes' numbers indicate the argument n

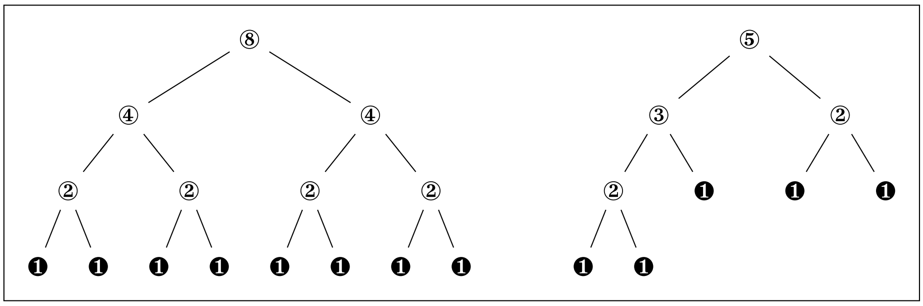


Figure 2.2 Call-tree of active *qsortGc* invocations for eight (left) and five (right) processing elements.

Listing 2.7 Parallel Quicksort.

```
-- quicksort :: [Num] → [Num] → C
def quicksort l r =
  with sl sg
  in case l of
    {1}      → r =:= Nil;
    {2} x xs → let ul = filter (gt x) l;
                   e  = filter (eq x) l;
                   ug = filter (lt x) l
               in quicksort ul sl & quicksort ug sg &
                  r =:= concat (Cons sl (Cons e (Cons sg Nil)))


-- quicksortGc :: [Num] → [Num] → C
def quicksortGc l r = qsortGc noPE l r


-- qsortGc :: Num → [Num] → [Num] → C
def qsortGc n l r =
  with sl sg
  in case l of
    {1}      → r =:= Nil;
    {2} x xs → case n > 1 of
                 {1} → let ul = filter (gt x) l;
                           e  = filter (eq x) l;
                           ug = filter (lt x) l;
                           n1 = n / 2
                       in qsortGc n1 ul sl & qsortGc (n − n1) ug sg &
                          r =:= concat (Cons sl (Cons e (Cons sg Nil)));
                 {2} → r =:= qsort l

-- qsort is a normal serial Quicksort function.
-- As n and 2 both are natural numbers, / performs integer
-- division with truncation of the fraction.
```

passed with the call. Black nodes represent working processes, white ones are suspended. Assuming an averagely unordered input so that *ul* and *ug* are of roughly the same length, load balancing is best if the number of processing elements is a power of two.

Listing 2.8 shows a similar parallelisation of the Mergesort algorithm (without granularity control).
The implementation of *splitAt* makes use of constrained variables f and b to return multiple values.

Listing 2.8 Parallel Mergesort.

$$
\begin{array}{l}
-- \; \mathit{mergesort} :: [\mathit{Num}] \to [\mathit{Num}] \to C \\
\textbf{def} \; \mathit{mergesort} \; l \; r = \textbf{with} \; \mathit{uf} \; \mathit{ub} \; \mathit{sf} \; \mathit{sb} \\
\qquad\qquad\qquad\qquad \textbf{in case} \; \mathit{length} \; l \leqslant 1 \; \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \{1\} \to r =:= l; \\
\qquad\qquad\qquad\qquad\quad \{2\} \to \mathit{split} \; l \; \mathit{uf} \; \mathit{ub} \;\; \& \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{mergesort} \; \mathit{uf} \; \mathit{sf} \; \& \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{mergesort} \; \mathit{ub} \; \mathit{sb} \; \& \\
\qquad\qquad\qquad\qquad\qquad\quad r =:= \mathit{smerge} \; \mathit{sf} \; \mathit{sb} \\
-- \; \mathit{smerge} \text{ is an order-preserving merge function.}
\end{array}
$$

$$
\begin{array}{l}
-- \; \mathit{split} :: [\alpha] \to [\alpha] \to [\alpha] \to C \\
\textbf{def} \; \mathit{split} \; l \; f \; b = \mathit{splitAt} \; ((\mathit{length} \; l) \; / \; 2) \; l \; f \; b
\end{array}
$$

$$
\begin{array}{l}
-- \; \mathit{splitAt} :: \mathit{Nat} \to [\alpha] \to [\alpha] \to [\alpha] \to C \\
\textbf{def} \; \mathit{splitAt} \; n \; l \; f \; b = \textbf{with} \; \mathit{fs} \\
\qquad\qquad\qquad\qquad \textbf{in case} \; n > 0 \; \textbf{of} \\
\qquad\qquad\qquad\qquad\quad \{1\} \to f =:= \mathit{Cons} \; (\mathit{hd} \; l) \; \mathit{fs} \; \& \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{splitAt} \; (n-1) \; (\mathit{tl} \; l) \; \mathit{fs} \; b; \\
\qquad\qquad\qquad\qquad\quad \{2\} \to f =:= \mathit{Nil} \; \& \; b =:= l
\end{array}
$$

2.2 Syntax

This section introduces the syntactic sorts and context-free syntax of FATOM programs.
The context-dependent syntax, i. e. types and scopes, is defined in the following Section 2.3.
Figure 2.3 to Figure 2.7 show the grammar of FATOM programs. Each of them introduces a number of non-terminal symbols, which are typeset in italics and start with a capital letter like *Nonterminal*. Their identifiers are also the names of the syntactic sorts.
All language elements are explained briefly in the following paragraphs. Their meaning is introduced informally; a formal definition of the language's semantics is presented in Section 2.4.

2.2.1 Syntactic structure

Programs A program is a non-empty sequence of supercombinator definitions (see Figure 2.3). A supercombinator is a functional or constraint abstraction and consists of the combinator's name and a possibly empty list of parameters.

Expressions FATOM has three different kinds of expressions (see Figure 2.4):

- functional expressions forming the computational part of the language,
- constraint expressions, and
- guarded expressions forming the coordination part of the language.

Functional expressions The set of functional expressions (see Figure 2.5) is a simple functional language:

- Function application is expressed by juxtaposition of the function name and arguments and associates to the left.
- Local definitions are introduced by the **let** construct and have non-recursive scope; i. e. $x = f\ a$ and $y = g\ x$ has to be written with nested **let**-expressions:

  let $x = f\ a$
  in **let** $y = g\ x$
  in e

- Case analysis and destructuring of compound data is performed by the **case** expression. All compound data is represented by $\mathtt{Pack}\{t, a\}$, which is a constructor with tag t and arity a. A branch in a case analysis consists of a tag number surrounded by {, } and a list of variables to bind the subcomponents to.
- Besides constructor based types, built-in types are natural numbers *Nat* and floating point numbers *Float*.
- Binary arithmetic and logic operators are built-in and applied in infix notation. Operator precedence and associativity is not included in the grammar but instead given in Table 2.1 with higher precedence meaning tighter binding. Logical operators consider $\mathtt{Pack}\{\mathtt{1}, \mathtt{0}\}$ as true and $\mathtt{Pack}\{\mathtt{2}, \mathtt{0}\}$ as false.

Precedence	Associativity	Operator
6	Left	Application
5	Left	$*$, $/$
4	Left	$+$, $-$
3	None	$==$, $\neq$, $<$, $\leqslant$, $>$, $\geqslant$
2	Left	&&
1	Left	$\|$

Table 2.1 Operator precedences.

Prg	$\rightarrow$	*ScDef*$^+$	
ScDef	$\rightarrow$	**def** *Name Parameter** = *Expr*	*(Supercombinator definition)*

Figure 2.3 Syntax of FATOM programs.

Expr	$\rightarrow$	*FExpr* | *CExpr* | *GExpr*	*(Expression)*
FExpr	$\rightarrow$	*App* | *Let* | *Case*	*(Functional expression)*
	|	*Infix* | *SExpr*	
CExpr	$\rightarrow$	*With* | *Conj*	*(Constraint expression)*
GExpr	$\rightarrow$	*GAlt* (❘ *GAlt*)*	*(Guarded expression)*
Name	$\rightarrow$	*Var*	*(Supercombinator name)*
Parameter	$\rightarrow$	*Var*	*(Formal parameter)*

Figure 2.4 Syntax of expressions.

App	$\rightarrow$	*FExpr SExpr*	*(Application)*
Let	$\rightarrow$	**let** *Binding* (; *Binding*)*	*(Non-recursive binding)*
		in *CExpr*	
Case	$\rightarrow$	**case** *FExpr* **of**	*(Case expression)*
		Branch (; *Branch*)*	
Infix	$\rightarrow$	*FExpr Binop FExpr*	*(Infix operators)*
SExpr	$\rightarrow$	*Var* | *Const* | *Pack*	*(Simple expression)*
	|	`(`*FExpr*`)` | *Builtin*	
Pack	$\rightarrow$	`Pack`$\{Nat, Nat\}$	*(Constructor)*
Const	$\rightarrow$	*Float* | *Nat*	*(Constant)*
Binding	$\rightarrow$	*Var* = *FExpr*	*(Binding form)*
Branch	$\rightarrow$	$\{Nat\}$ *Var** $\rightarrow$ *CExpr*	*(Branch of a case expression)*
Binop	$\rightarrow$	$+ \mid - \mid * \mid /$	*(Arithmetic operators)*
	|	$< \mid > \mid \leqslant \mid \geqslant \mid == \mid \neq$	*(Comparison operators)*
	|	&& | ‖	*(Logical connectives)*
Builtin	$\rightarrow$	`not` | `noPE` | `error` | `neg`	*(Builtin function)*
	|	`natToFloat` | `floatToNat`	

Figure 2.5 Syntax of functional expressions.

With	$\rightarrow$	**with** *Var*$^+$ **in** *Conj*	*(Introducing constrained variables)*
Conj	$\rightarrow$	*Atom* (& *Atom*)*	*(Conjunction)*
Atom	$\rightarrow$	*Var* =:= *FExpr*	*(Tell equality)*
	|	*FExpr*	*(Constraint abstraction)*

Figure 2.6 Syntax of constraint expressions.

GAlt	$\rightarrow$	*Guard* $\Rightarrow$ *CExpr*	*(Guarded alternative)*
Guard	$\rightarrow$	*CPrim* (& *CPrim*)*	*(Conjunction of constraint primitives)*
CPrim	$\rightarrow$	`pack` *Var Nat Nat*	*(Constructor term)*
	$\vert$	`bound` *Var*	*(Bound variable)*
	$\vert$	`unbound` *Var*	*(Unbound variable)*

Figure 2.7 Syntax of guarded expressions.

Built-in functions with arity other than two are applied in the usual way. Among them are `noPE` (number of processing elements, a constant) and `error` (the totally undefined function).

Constraint and guarded expressions The basic constraint expression is an equality constraint $v =:= e$ (see Figure 2.6). Invocations of constraint abstractions and equality constraints can built conjunctions. New constrained variables have to be introduced by the **with** keyword.
Constraint expressions can be guarded by conjunctions of primitive constraints (see Figure 2.7). Several guarded expressions may form a disjunction of alternatives (Figure 2.4).
The following primitive constraints are built-in:

- `pack` $v\ t\ a$: Satisfied if v is bound to a constructor of tag t and arity a.
- `bound` v: Satisfied if v is bound to any value.
- `unbound` v: Satisfied if v is not bound to a value.

Variables The syntactic sort of variables *Var* includes constrained variables as well as variables introduced by supercombinator definitions, local definitions, and case expressions.

Variables for syntactic sorts

The following section will, unless indicated differently, use certain variable names for elements of syntactic sorts:

- v for variables *Var*,
- e and b for expressions *Expr*, *FExpr*, *CExpr*, *GExpr*,
- g for guards *Guard*,
- c for constraints *CPrim*,
- $\oplus$ for binary built-in operators *Binop*.

They may appear primed or with indices like x_1, x_2, $\ldots$, x_i, x'.

$$\begin{array}{lcl}
\mathit{Nat} & \rightarrow & \mathit{Digit}^{+} \\
\mathit{Float} & \rightarrow & [\,\mathit{Sign}\,]\ \mathit{Nat}\ [\ \texttt{.}\,\mathit{Nat}\,]\ [\ (\texttt{e} \mid \texttt{E})\ [\,\mathit{Sign}\,]\ \mathit{Nat}\,] \\
\mathit{Var} & \rightarrow & \mathit{Alph}\ (\mathit{Alph} \mid \mathit{Digit})^{*} \\
\mathit{Digit} & \rightarrow & \texttt{0} \mid \ldots \mid \texttt{9} \\
\mathit{Sign} & \rightarrow & \texttt{+} \mid \texttt{-} \\
\mathit{Alph} & \rightarrow & \texttt{a} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \ldots \mid \texttt{Z}
\end{array}$$

Figure 2.8 Lexemes of constants, variables, and numbers.

2.2.2 Lexical structure

All terminal symbols mentioned in grammars 2.3 to 2.7 form their own lexeme. Constant, variable, and number lexemes are defined by the grammar shown in Figure 2.8.

2.3 Context conditions

This section specifies scopes of variables and required context conditions for valid FATOM programs.
Specification is done using annotated syntax trees as in [Pep97].

2.3.1 Scopes

The scope of a variable v is the very piece of code in which v is visible.
FATOM is lexically scoped, which means that variables are bound to values according to the hierarchical structure of the source code.

Definition 2.1 *(Scope)*
A scope is an environment that contains variables:

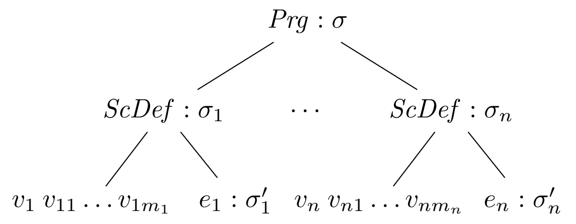
$$SCOPE := ENV^{\emptyset}_{Var}$$

The remainder of this section associates scopes with nodes of abstract syntax trees and equalities that state how scopes evolve out of each other. These annotated syntax trees specify which scope is to be used in a certain part of the program to lookup a variable's attributes like type or value. A scope is defined as $ENV^{\emptyset}$ because these attributes are of no relevance for this section.

Supercombinator definition

$$\begin{array}{lcl}
\mathit{Prg} & \rightarrow & \mathit{ScDef}^{+} \\
\mathit{ScDef} & \rightarrow & \mathit{Name}\ \mathit{Parameter}^{*} = \mathit{Expr}
\end{array}$$

The names of supercombinators are visible in all supercombinator bodies. Additionally, all predefined names of built-in functions in σ are also visible in the supercombinator bodies.
The names of a supercombinator's parameters are visible in its body and may shadow other top-level definitions.

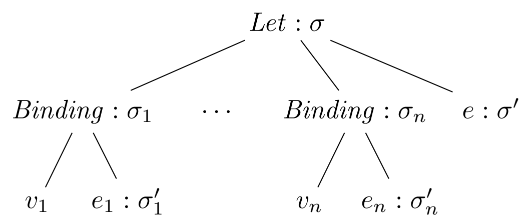


$$\begin{aligned} \sigma_i &= \sigma.[v_1, \ldots, v_n] \\ \sigma'_i &= \sigma_i.[v_{i1}, \ldots, v_{im_i}] \\ i &= 1, \ldots, n \end{aligned}$$

Local definition

$$\begin{aligned} \mathit{Let} &\rightarrow \textbf{let}\ \mathit{Binding}\ \ (\texttt{;}\,\mathit{Binding})^* \\ &\qquad \textbf{in}\ \mathit{CExpr} \\ \mathit{Binding} &\rightarrow \mathit{Var} = \mathit{FExpr} \end{aligned}$$

Local definitions introduce variable names which are visible in the expression following the **in** keyword. Like supercombinator names may shadow built-in functions, local definitions shadow supercombinators or other definitions of the same name. **let**-bindings are not recursive: none of the new names is visible on any right hand side.

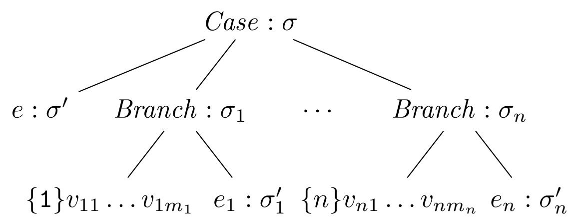


$$\begin{aligned} \sigma_1 = \cdots = \sigma_n &= \sigma \\ \sigma'_1 = \cdots = \sigma'_n &= \sigma \\ \sigma' &= \sigma.[v_1, \ldots, v_n] \end{aligned}$$

Case analysis

$$\begin{aligned} \mathit{Case} &\rightarrow \textbf{case}\ \mathit{FExpr}\ \textbf{of} \\ &\qquad \mathit{Branch}\ \ (\texttt{;}\,\mathit{Branch})^* \\ \mathit{Branch} &\rightarrow \{\mathit{Nat}\}\ \mathit{Var}^* \rightarrow \mathit{CExpr} \end{aligned}$$

Case analysis introduces variables in each branch's expression to which the subexpressions are bound. The expression analysed is evaluated in the environment of the case statement.



$$\begin{aligned} \sigma' &= \sigma \\ \sigma_1 = \cdots = \sigma_n &= \sigma \\ \sigma'_i &= \sigma_i.[v_{i1}, \ldots, v_{im_i}] \\ i &= 1, \ldots, n \end{aligned}$$

Application

$$\begin{aligned} \mathit{App} &\rightarrow \mathit{FExpr}\ \mathit{SExpr} \\ \mathit{SExpr} &\rightarrow \mathit{Var} \mid \mathit{Const} \mid \mathit{Pack} \mid \texttt{(}\mathit{FExpr}\texttt{)} \mid \mathit{Builtin} \end{aligned}$$

Application does not affect scopes. The same is true for infix application, which is just a special case of application.

$$\begin{array}{|c|c|} \hline \begin{array}{c} \mathit{App} : \sigma \\ \diagup \quad \diagdown \\ e_1 : \sigma_1 \qquad e_2 : \sigma_2 \end{array} & \sigma_1 = \sigma_2 = \sigma \\ \hline \end{array}$$

Introduction of constrained variables $\qquad$ $\underline{\mathit{With} \rightarrow \textbf{with } \mathit{Var}^{+} \textbf{ in } \mathit{Conj}}$

with-expressions are similar to **let**-expressions and introduce a set of names into the expression following **in**.

$$\begin{array}{|c|c|} \hline \begin{array}{c} \mathit{With} : \sigma \\ \diagup \quad \diagdown \\ v_1 \ldots v_n \qquad e : \sigma' \end{array} & \sigma' = \sigma.[v_1, \ldots, v_n] \\ \hline \end{array}$$

2.3.2 Types

The following section describes FATOM's type system. Like in the example programs (Section 2.1) HASKELL syntax is used to denote types. The most frequently used notations are shown in Table 2.2.

Built-in types

FATOM has four built-in types shown in Table 2.3.
All built-in arithmetic and comparison functions work on *Nat* as well as *Float* and we define a fifth type *Num* for convenience which is the union of *Nat* and *Float*:

$$\mathit{Num} := \mathit{Nat} \cup \mathit{Float}$$

Pack contains all constructor types. Each type's constructor is represented by a tag t and its arity a. An algebraic data type τ with constructors C_1 to C_n

$$\begin{array}{rcl} \tau & = & C_1\ \tau_{11} \ldots \tau_{1m_1} \\ & \vdots & \\ & | & C_n\ \tau_{n1} \ldots \tau_{nm_n} \end{array}$$

is translated into the following *Pack*-expressions:

$$\texttt{Pack}\{1, m_1\} \quad \ldots \quad \texttt{Pack}\{n, m_n\}$$

As the built-in boolean operators use $\texttt{Pack}\{1, 0\}$ as true and $\texttt{Pack}\{2, 0\}$ as false the following abbreviation suggests itself:

$$\mathit{Bool} = \mathit{True} \mid \mathit{False}$$

$\alpha \rightarrow \beta$	Function type
$[\alpha]$	List
(α, β)	Tuple

Table 2.2 Syntax for types.

Nat	Natural numbers $0, 1, 2, 3, \dots$
Float	Rational numbers $\frac{p}{q}$ with $p, q \in \mathbb{Z}$
C	Constraint
Pack	Constructor type

Table 2.3 Built-in types.

Comparison functions

$\leqslant :: \mathit{Num} \to \mathit{Num} \to \mathit{Bool}$
$< :: \mathit{Num} \to \mathit{Num} \to \mathit{Bool}$
$\geqslant :: \mathit{Num} \to \mathit{Num} \to \mathit{Bool}$
$> :: \mathit{Num} \to \mathit{Num} \to \mathit{Bool}$
$== :: \mathit{Num} \to \mathit{Num} \to \mathit{Bool}$
$\neq :: \mathit{Num} \to \mathit{Num} \to \mathit{Bool}$

Arithmetic functions

$+ :: \mathit{Num} \to \mathit{Num} \to \mathit{Num}$
$- :: \mathit{Num} \to \mathit{Num} \to \mathit{Num}$
$* :: \mathit{Num} \to \mathit{Num} \to \mathit{Num}$
$/ :: \mathit{Num} \to \mathit{Num} \to \mathit{Num}$
$\texttt{neg} :: \mathit{Float} \to \mathit{Float}$

Others

$\texttt{noPE} :: \mathit{Nat}$
$\texttt{error} :: \alpha$

Conversion functions

$\texttt{natToFloat} :: \mathit{Nat} \to \mathit{Float}$
$\texttt{floatToNat} :: \mathit{Float} \to \mathit{Nat}$

Boolean functions

$\&\& :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$
$\Vert :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}$
$\texttt{not} :: \mathit{Bool} \to \mathit{Bool}$

Table 2.4 Types of built-in functions.

with constructors

$$\begin{aligned} \mathit{True} &= \texttt{Pack}\{\texttt{1},\texttt{0}\} \\ \mathit{False} &= \texttt{Pack}\{\texttt{2},\texttt{0}\}. \end{aligned}$$

For the same reason of convenience, the list constructors *Cons* and *Nil* are defined like this:

$$\begin{aligned} \mathit{Nil} &= \texttt{Pack}\{\texttt{1},\texttt{0}\} \\ \mathit{Cons} &= \texttt{Pack}\{\texttt{2},\texttt{2}\} \end{aligned}$$

Types of the built-in functions and constraints

The types of the built-in functions are shown in Table 2.4. The binary arithmetic functions return a *Nat* if all arguments are of type *Nat*, a *Float* otherwise. The exact behaviour is specified by Definition 2.18 in Section 2.4.3.
The result type of all built-in constraints is C while the types of their arguments differ. Their types are shown in Table 2.5, their formal semantics are specified in Section 2.4.2.

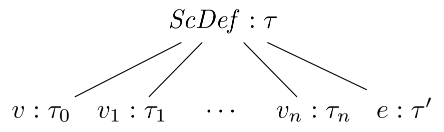
Type equalities

Like scopes, type equalities are defined by annotated syntax trees.

Supercombinator definition The type of a supercombinator definition and its name v is a function type. Its argument types are the types of the parameters and the result type is the type of the body expression.

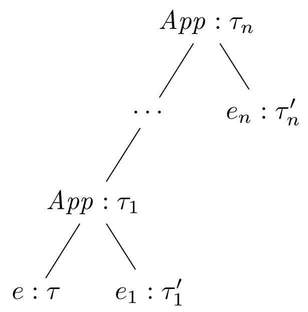
$$
\begin{aligned}
\texttt{pack} &:: Pack \to Nat \to Nat \to C \\
\texttt{bound} &:: \alpha \to C \\
\texttt{unbound} &:: \alpha \to C \\
\& &:: C \to C \to C \\
=:= &:: \alpha \to \alpha \to C
\end{aligned}
$$

Table 2.5 Types of built-in constraints.



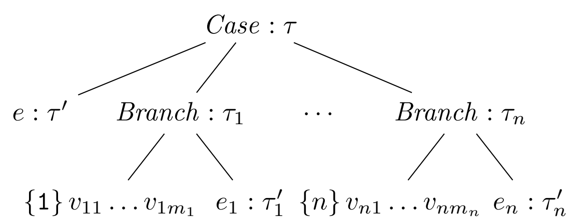
$$
\begin{aligned}
\tau &= \tau_0 \\
\tau_0 &= \tau_1 \to \cdots \to \tau_n \to \tau'
\end{aligned}
$$

Application For a type correct application the arguments have to match the argument types of the applied function (a supercombinator, constructor, or a built-in function). The type of the whole application is the result type of the function.



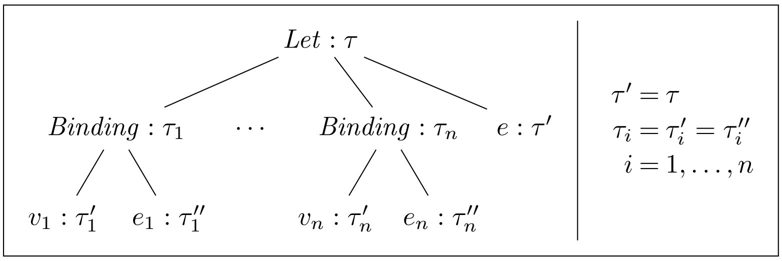
$$
\begin{aligned}
\tau &= \tau'_1 \to \cdots \to \tau'_n \to \tau_n \\
\tau_i &= \tau'_{i+1} \to \cdots \to \tau'_n \to \tau_n \\
i &= 1, \ldots, n-1
\end{aligned}
$$

Case analysis The expression analysed by a case expression has to be of type *Pack*, all branch expressions have to be of the same type, which is also the type of the whole expression.
The types of the variables $v_{ij}, i = 1, \ldots, n, j = 1, \ldots, m_i$ cannot be specified because they depend on the subterms' types of the *Pack*-expressions. Type correctness of a case analysis can only be checked in a type-safe high-level language.

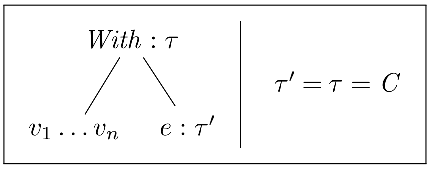


$$
\begin{aligned}
\tau' &= Pack \\
\tau &= \tau_1 = \cdots = \tau_n \\
\tau'_i &= \tau_i \\
i &= 1, \ldots, n
\end{aligned}
$$

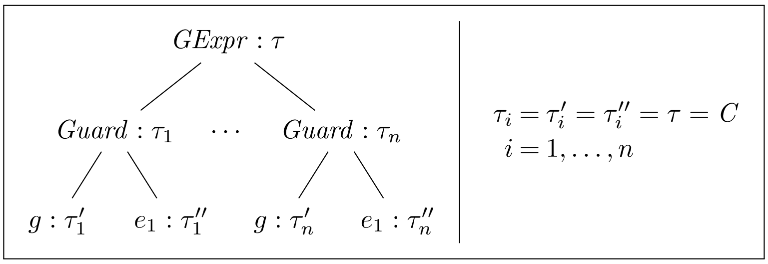
Local definition The type of a local definition is the type of its body expression. Furthermore, the type of a bound variable is determined by the type of the expression bound.



Introduction of constrained variables A **with**-expression as a whole is of type C and so is its body.



Guarded expression The type of a guarded expression is the type of its disjunctive branches which have to be equal. The type of a guard is C.



Tell equalities and conjunctions The types of expressions like $v =:= e$ and $e_1 \,\&\, \ldots \,\&\, e_n$ are already given in Table 2.5.

Shortcomings of the type system

As already stated in Section 2.1, FATOM is not type safe. This is because all constructor types are represented by a single constructor $\texttt{Pack}\{t, a\}$. Each constructor of a type is represented by a type-unique tag t. The constructor's arity is given by a. Thus, the first constructor *Node* of a binary tree

$$\begin{aligned} \mathit{BinTree}\ \alpha = {} & \mathit{Node}\ (\mathit{BinTree}\ \alpha)\ \alpha\ (\mathit{BinTree}\ \alpha) \\ \mid {} & \mathit{Nil} \end{aligned}$$

could be represented as $\texttt{Pack}\{\texttt{1}, \texttt{3}\}$ and cannot be distinguished from a 3-tuple's constructor

$$\mathit{ThreeTuple}\ \alpha\ \beta\ \gamma = (\alpha, \beta, \gamma)$$

which could also be represented by $\texttt{Pack}\{\texttt{1}, \texttt{3}\}$.
However, as FATOM is intended to be a compiler target language, the program will have been type-checked before translation into FATOM and type unsafety is not an issue.

Main supercombinator

Each FATOM-program has to define a `main` supercombinator. This supercombinator is the starting point of program evaluation.
The type of the supercombinator has to be

$$\texttt{main} :: \alpha \rightarrow C.$$

The result of the computation e may be bound to `main`'s parameter by a tell equality: **def** $\texttt{main}\,\texttt{r} = \texttt{r} =:= e$

2.4 Semantics

The following second part of this chapter defines the meaning of FATOM programs by giving structural operational (small-step) semantics similar to the formalism given in [NN92].
This section first introduces essential concepts the semantics is based on and defines certain auxiliary functions required to express the rules of the transition relation.

2.4.1 Basic definitions

Ground values are the primitive values of FATOM. They correspond to the three built-in types *Nat*, *Float*, and *Pack* with *Float* and *Nat* mapped to $\mathbb{Q}$ (see Definition 2.18).

Definition 2.2 *(Ground value, term)*
The set VAL of ground values is defined as

$$VAL := \mathbb{N} \cup \mathbb{Q} \cup \mathbb{T}$$

with $\mathbb{N}$ *being the set of natural numbers,* $\mathbb{Q}$ *being the set of rational numbers, and* $\mathbb{T}$ *being the set of terms defined as follows:*

$$\mathbb{T} := \{w \mid w = Pack\{t, a\}\, e_1 \dots e_a \text{ and } e_1, \dots, e_a \in Expr \text{ and } t, a \geq 0\}$$

Examples of ground values are

- -1, $1.6 \cdot 10^{-19}$, 0,
- *Pack*{2, 2} `map (succ xs) tl ys`, *Pack*{1, 2} `1 0`.

Constraints are simple equality constraints between variables and ground values or primitive constraints.

Definition 2.3 *(Equality constraint, bound/unbound variable)*
The set of equality constraints EQC is formed of bound and unbound variables:

$$EQC := \{v =:= w \mid v \in Var \text{ and } w \in VAL \cup Var \cup \{\Diamond\}\}$$

A variable v_1 *with* $v_1 =:= \Diamond \in EQC$ *is said to be* unbound, *a variable* v_2 *with* $v_2 =:= w$, $w \in VAL \cup Var$, *is said to be* bound.

Definition 2.4 *(Primitive constraint)*
The set of primitive constraints *is defined as*

$$\begin{aligned} PRIMC := \; & \{\texttt{pack}\ v\ t\ a \mid v \in Var \ \textit{and}\ t, a \geq 0\} \cup \\ & \{\texttt{bound}\ v \mid v \in Var\} \cup \\ & \{\texttt{unbound}\ v \mid v \in Var\}. \end{aligned}$$

Definition 2.5 *(Constraint, constraint conjunction)*
The set of constraints *is the union of primitive and equality constraints:*

$$CONSTR := PRIMC \ \cup \ EQC$$

A constraint conjunction is an expression of the form

$$c_1 \ \wedge \ \ldots \ \wedge \ c_n, \ n \geq 1$$

for which holds

$$c_i \in CONSTR, \ 1 \leq i \leq n.$$

The set of constraint conjunctions is denoted by $\Delta CONSTR$*.*

The following definition is very important for the overall understanding of FATOM evaluation.

Definition 2.6 *(Configuration, process, store)*
A configuration γ *is an evaluation's state at a certain time. The set of configurations* $CONF$ *is defined as*

$$CONF := PROC \times STORE.$$

The set of processes $PROC$ *is the set of all multisets of expressions and ground values:*

$$PROC := \{(A, m_A) \mid A \in \mathbb{P}\,(Expr \ \cup \ VAL) \ \textit{and}\ m_A : A \rightarrow \mathbb{N} \setminus \{0\}\}$$

A store $s \in STORE$ *is the power-set of equality constraints:*

$$STORE := \mathbb{P}EQC$$

A process is an expression which is evaluated concurrently with other processes and is similar to a *thread* or *lightweight process* (cf. [Sta05]) – not to be confused with a classical UNIX process.
The reason to use multisets instead of sets for $PROC$ is that it must be possible to evaluate the same expression e several times. A process (multi)set $\{e, e\}$ is different from $\{e\}$ but the former cannot be expressed with sets. Despite being a multiset, the first component of a configuration is called process *set* in the remainder of this chapter.
The next definition puts two configurations into a relation which forms the operational semantics of FATOM.

Definition 2.7 *(Operational semantics)*
The operational semantics of a FATOM *program P is defined by a transition relation*

$$\rhd_P \subseteq CONF \times CONF.$$

For two configurations γ, γ' the infix notation is used:

$$\gamma \rhd_P \gamma' :\Leftrightarrow \langle \gamma, \gamma' \rangle \in \rhd_P$$

The transitive closure of $\rhd_P$ is denoted by $\rhd_P^+$, i. e. the following properties hold:

$$\gamma \rhd_P \gamma' \Rightarrow \gamma \rhd_P^+ \gamma'$$
$$\gamma \rhd_P^+ \gamma' \text{ and } \gamma' \rhd_P \gamma'' \Rightarrow \gamma \rhd_P^+ \gamma''$$

A fully detailed specification of $\rhd_P$ is presented in Section 2.4.4. For the explanation of the transition relation the next definition is useful:

Definition 2.8 *(Normal form)*
A functional expression e is in normal form *if it can not be reduced any further:*

$$\forall e' : e' \neq e \Rightarrow \neg(\langle \{e\}, s\rangle \rhd_P \langle \{e'\}, s'\rangle)$$

Every evaluation of a FATOM program begins with the following configuration:

Definition 2.9 *(Initial configuration)*
The initial configuration γ_0 *of a* FATOM *program P is*

$$\gamma_0 := \langle \{\texttt{main r}\}, \{\texttt{r} =:= \Diamond\}\rangle.$$

An evaluation may end either in a stuck or final configuration:

Definition 2.10 *(Final/stuck configuration)*
A final configuration *is a configuration $\gamma_f = \langle \emptyset, s\rangle$, i. e. there are no processes left for evaluation.*
A stuck configuration *is a configuration $\gamma_s = \langle p, s\rangle$ with $p \neq \emptyset$ and none of the rules forming $\rhd_P$ can be applied.*

Of course, there are many evaluations that never reach a final or stuck configuration; the producer-consumer setting of Section 2.1.1 is an example.

2.4.2 Ask and tell operations

Two functions, $\mathcal{A}$sk and $\mathcal{T}$ell, work on the constraint store and interpret its constraint system.
Operations on the constraint system may succeed or fail, which is expressed by the next definition:

Definition 2.11 *(Result)*
The result *of a constraint operation is either success (ok) or failure (fail):*

$$RESULT := \{ok, fail\}$$

The ask operation is required to extract information from the constraint store.

Definition 2.12 *(Ask)*
The constraint operation $\mathcal{A}$ has the signature

$$\mathcal{A} : \Delta CONSTR \times STORE \rightarrow RESULT$$

and definition:

$$\mathcal{A}[\![\, c_1 \ \wedge \ \ldots \ \wedge \ c_n \,]\!]\, s = \begin{cases} ok & \textit{for } \forall i : 1 \leq i \leq n \textit{ with } entail_s(c_i) = ok \\ fail & \textit{otherwise} \end{cases}$$

$$entail : STORE \times CONSTR \rightarrow RESULT$$

$$entail_s(\texttt{pack}\ v\ t\ a) = \begin{cases} ok & \textit{for } v =:= Pack\{t, a\}\ e_1 \ldots e_a \in s \\ fail & \textit{otherwise} \end{cases}$$

$$entail_s(\texttt{bound}\ v) = \begin{cases} ok & \textit{for } v =:= w \in s \textit{ and } w \neq \Diamond \\ fail & \textit{otherwise} \end{cases}$$

$$entail_s(\texttt{unbound}\ v) = \begin{cases} ok & \textit{for } v =:= \Diamond \in s \\ fail & \textit{otherwise} \end{cases}$$

$$entail_s(v =:= w) = \begin{cases} ok & \textit{for } v =:= w \in s \\ fail & \textit{otherwise} \end{cases}$$

The tell operation is used to augment information to the constraint store while ensuring consistency.

Definition 2.13 *(Tell)*
The tell operation $\mathcal{T}$ has the signature

$$\mathcal{T} : EQC \times STORE \rightarrow \{fail\} \cup STORE.$$

It returns a new constraint store if an equality constraint $v =:= w$ is successfully added to the store and fail otherwise.

$$\mathcal{T}[\![\, v =:= w \,]\!]\, s = \begin{cases} s & \textit{for } lookup_s(v) = w \\ s \cup \{v =:= w\} & \textit{for } lookup_s(v) = \textit{undefined} \\ s \setminus \{v =:= \Diamond\} \cup \{v =:= w\} & \textit{for } lookup_s(v) = \Diamond \\ fail & \textit{otherwise} \end{cases}$$

$$lookup : STORE \times Var \hookrightarrow VAL \cup Var \cup \{\Diamond\}$$
$$lookup_s(v) = w' \quad \textit{for } v =:= w' \in s$$

2.4.3 Auxiliary functions and definitions

The next five definitions are used in the specification of the transition relation $\rhd_P$ presented in the next section.
The substitution function replaces free variables of constraint expressions by other constraint expressions. As constraint expressions *CExpr* contain functional expressions *FExpr* (cf. Section 2.2.1) the substitution function will also be used to define functional reduction in Section 2.4.4.
Before defining substitution, the set of free variables is given:

Definition 2.14 *(Free variables)*
A free variable is a variable not bound by a **let**-*,* **case**-*, or* **with**-*construct. Formally, the set of free variables of a constraint expression $FV(e)$ is defined recursively:*

$$FV : \mathit{CExpr} \to \mathbb{P}\,\mathit{Var}$$

$$FV(\textbf{with } v_1 \dots v_n \textbf{ in } e) = FV(e) \setminus \{v_1, \dots, v_n\}$$

$$FV(e_1 \,\&\, \dots \,\&\, e_n) = \bigcup_{i=1,\dots,n} FV(e_i)$$

$$FV(v =:= e) = FV(e) \cup \{v\}$$

$$FV\begin{pmatrix}\textbf{let} \\ \quad v_1 = e_1\,\textbf{;} \\ \quad \vdots \\ \quad v_n = e_n \\ \textbf{in } e\end{pmatrix} = \bigcup_{i=1,\dots,n} FV(e_i) \cup (FV(e) \setminus \{v_1, \dots, v_n\})$$

$$FV\begin{pmatrix}\textbf{case } e \textbf{ of} \\ \quad \mathit{branch}_1\,\textbf{;} \\ \quad \vdots \\ \quad \mathit{branch}_n\end{pmatrix} = \bigcup_{i=1,\dots,n} FV(\mathit{branch}_i) \cup FV(e)$$

$$FV(\{t\}\, v_1 \dots v_n \to e) = FV(e) \setminus \{v_1, \dots, v_n\}$$

$$FV(e_1 \dots e_n) = \bigcup_{i=1,\dots,n} FV(e_i)$$

$$FV(e_1 \oplus e_2) = FV(e_1) \cup FV(e_2)$$

$$FV((\, e\,)) = FV(e)$$

$$FV(v) = \{v\}$$

All other constraint expressions have no free variables:

$$FV(e) = \emptyset$$

Definition 2.15 *(Substitution)*
The substitution function has the signature

$$\mathit{CExpr} \times \mathit{Var} \times \mathit{CExpr} \to \mathit{CExpr}$$

and is written as $\cdot\,[\,\cdot\,/\,\cdot\,]$.
Composition of substitutions may be abbreviated:

$$(\cdots((e[v_1/e_1])[v_2/e_2])\cdots)[v_n/e_n] = e[v_1/e_1, v_2/e_2, \dots, v_n/e_n]$$

The function's behaviour is defined along the syntax of constraint expressions:

$$(\textbf{with}\ v_1 \ldots v_n\ \textbf{in}\ b)\,[v/e] = \textbf{with}\ v_1 \ldots v_n\ \textbf{in}\ b\,[v/e] \qquad (*)$$
$$\textit{for } v_i \neq v, i = 1, \ldots, n$$

$$(e_1 \,\&\, \ldots \,\&\, e_n)\,[v/e] = e_1[v/e] \,\&\, \ldots \,\&\, e_n[v/e]$$

$$(v' =\!:= b)\,[v/e] = v'\,[v/e] =\!:= b\,[v/e]$$

$$\begin{pmatrix} \textbf{let} \\ \quad binding_1\,; \\ \quad \vdots \\ \quad binding_n \\ \textbf{in}\ b \end{pmatrix}[v/e] = \begin{array}{l} \textbf{let} \\ \quad binding_1[v/e]\,; \\ \quad \vdots \\ \quad binding_n[v/e] \\ \textbf{in}\ b\,[v/e] \end{array} \qquad (*)$$
$$\textit{for } v_i \neq v, i = 1, \ldots, n$$

$$(v = b)\,[v/e] = (v = b[v/e])$$

$$\begin{pmatrix} \textbf{case}\ b\ \textbf{of} \\ \quad branch_1\,; \\ \quad \vdots \\ \quad branch_n \end{pmatrix}[v/e] = \begin{array}{l} \textbf{case}\ b\,[v/e]\ \textbf{of} \\ \quad branch_1[v/e]\,; \\ \quad \vdots \\ \quad branch_n[v/e] \end{array}$$

$$(\{t\}\ v_1 \ldots v_n \rightarrow b)[v/e] = \{t\}\ v_1 \ldots v_n \rightarrow b\,[v/e] \qquad (*)$$
$$\textit{for } v_i \neq v, i = 1, \ldots, n$$

$$(e_1 \ldots e_n)\,[v/e] = e_1[v/e] \ldots e_n[v/e]$$

$$(e_1 \oplus e_2)\,[v/e] = e_1[v/e] \oplus e_2[v/e]$$

$$(\, b\,)\,[v/e] = (\, b[v/e]\,)$$

$$v'\,[v/e] = \begin{cases} e & \textit{for } v = v' \\ v' & \textit{otherwise} \end{cases}$$

For all other constraint expressions, substitution equals identity:

$$b\,[v/e] = b$$

For the binding expressions marked with $(*)$ *an implicit renaming is assumed if any of the bound variables* v_i *is a free variable in* e*, i. e.* $v_i \in FV(e)$*, to prevent bound variable capture.*

The next two functions return information about a given program P and the content of a store s.

Definition 2.16 *(Supercombinator lookup)*
Given a FATOM *program* P*, the partial function* $SC_P(v)$ *returns the supercombinator definition of a supercombinator named* v*:*

$$SC_P : \mathbb{P}ScDef \times Var \hookrightarrow ScDef$$

$$P = \begin{pmatrix} \mathbf{def}\ v_1\, v_{11} \dots v_{1m_1} & = & e_1 \\ & \vdots & \\ \mathbf{def}\ v_n\, v_{n1} \dots v_{nm_n} & = & e_n \end{pmatrix}$$
$$\Rightarrow SC_P(v_i) = (v_i\, v_{i1} \dots v_{im_i} = e_i) \quad \textit{for } 1 \le i \le n$$

Definition 2.17 *(Variable extraction)*
The function vars returns the set of all bound and unbound variables of a store:

$$vars : STORE \to \mathbb{P}\, Var$$

$$vars(s) = \{v_1, \dots, v_n\} \quad \textit{for } s = \{v_1 =:= w_1, \dots, v_n =:= w_n\}$$

The function $\mathcal{B}$ interprets built-in functions and operators, constants (i. e. numbers), and constructors (i. e. `Pack`-expressions). It is a partial function because it only interprets a subset of *FExpr*.

Definition 2.18 *(Interpretation of built-in functions and constants)*
Built-in functions and operators as well as constants are interpreted by the function $\mathcal{B}$. The arguments to built-in functions have to be in normal form.

$$\mathcal{B} : FExpr \hookrightarrow VAL$$

$$\begin{aligned}
\mathcal{B}[\![\, \texttt{error} \,]\!] &= \textit{abort evaluation} \\
\mathcal{B}[\![\, \texttt{noPE} \,]\!] &= \textit{number of available processing elements} \\
\mathcal{B}[\![\, \texttt{Pack}\{t,a\}\, e_1 \dots e_n \,]\!] &= Pack\{t,a\}\, e_1 \dots e_n \\
\mathcal{B}[\![\, x{+}y \,]\!] &= \begin{cases} x +^{\mathbb{N}} y & \textit{for } x, y \in \mathbb{N} \\ x +^{\mathbb{Q}} y & \textit{otherwise} \end{cases} \\
\mathcal{B}[\![\, x{-}y \,]\!] &= \begin{cases} sub(x,y) & \textit{for } x, y \in \mathbb{N} \\ x -^{\mathbb{Q}} y & \textit{otherwise} \end{cases} \\
\mathcal{B}[\![\, x{*}y \,]\!] &= \begin{cases} x \cdot^{\mathbb{N}} y & \textit{for } x, y \in \mathbb{N} \\ x \cdot^{\mathbb{Q}} y & \textit{otherwise} \end{cases} \\
\mathcal{B}[\![\, x/y \,]\!] &= \begin{cases} div(x,y) & \textit{for } x, y \in \mathbb{N} \\ x \div^{\mathbb{Q}} y & \textit{otherwise} \end{cases} \\
\mathcal{B}[\![\, \texttt{neg}\ x \,]\!] &= \begin{cases} 0^{\mathbb{N}} & \textit{for } x \in \mathbb{N} \\ -1^{\mathbb{Q}} \cdot^{\mathbb{Q}} x & \textit{otherwise} \end{cases} \\
\mathcal{B}[\![\, \texttt{natToFloat}\ x \,]\!] &= x \\
\mathcal{B}[\![\, \texttt{floatToNat}\ x \,]\!] &= round(x) \\
\mathcal{B}[\![\, x{<}y \,]\!] &= \begin{cases} Pack\{\texttt{1},\texttt{0}\} & \textit{for } x <^{\mathbb{Q}} y \\ Pack\{\texttt{2},\texttt{0}\} & \textit{otherwise} \end{cases} \\
\mathcal{B}[\![\, x{\leqslant}y \,]\!] &= \begin{cases} Pack\{\texttt{1},\texttt{0}\} & \textit{for } x \le^{\mathbb{Q}} y \\ Pack\{\texttt{2},\texttt{0}\} & \textit{otherwise} \end{cases} \\
\mathcal{B}[\![\, x{==}y \,]\!] &= \begin{cases} Pack\{\texttt{1},\texttt{0}\} & \textit{for } x =^{\mathbb{Q}} y \\ Pack\{\texttt{2},\texttt{0}\} & \textit{otherwise} \end{cases}
\end{aligned}$$

$$\mathcal{B}[\![\, x{\neq}y \,]\!] = \begin{cases} \mathit{Pack}\{\texttt{1},\texttt{0}\} & \mathit{for}\ x \neq^{\mathbb{Q}} y \\ \mathit{Pack}\{\texttt{2},\texttt{0}\} & \mathit{otherwise} \end{cases}$$

$$\mathcal{B}[\![\, x{\geqslant}y \,]\!] = \begin{cases} \mathit{Pack}\{\texttt{1},\texttt{0}\} & \mathit{for}\ x \geq^{\mathbb{Q}} y \\ \mathit{Pack}\{\texttt{2},\texttt{0}\} & \mathit{otherwise} \end{cases}$$

$$\mathcal{B}[\![\, x{>}y \,]\!] = \begin{cases} \mathit{Pack}\{\texttt{1},\texttt{0}\} & \mathit{for}\ x >^{\mathbb{Q}} y \\ \mathit{Pack}\{\texttt{2},\texttt{0}\} & \mathit{otherwise} \end{cases}$$

$$\begin{aligned}
\mathcal{B}[\![\, \texttt{not}\ \mathit{Pack}\{\texttt{2},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{1},\texttt{0}\} \\
\mathcal{B}[\![\, \texttt{not}\ \mathit{Pack}\{\texttt{1},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{2},\texttt{0}\} \\
\mathcal{B}[\![\, \mathit{Pack}\{\texttt{2},\texttt{0}\}\ \&\&\ \mathit{Pack}\{\texttt{2},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{2},\texttt{0}\} \\
\mathcal{B}[\![\, \mathit{Pack}\{\texttt{1},\texttt{0}\}\ \&\&\ \mathit{Pack}\{\texttt{2},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{2},\texttt{0}\} \\
\mathcal{B}[\![\, \mathit{Pack}\{\texttt{2},\texttt{0}\}\ \&\&\ \mathit{Pack}\{\texttt{1},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{2},\texttt{0}\} \\
\mathcal{B}[\![\, \mathit{Pack}\{\texttt{1},\texttt{0}\}\ \&\&\ \mathit{Pack}\{\texttt{1},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{1},\texttt{0}\} \\
\mathcal{B}[\![\, \mathit{Pack}\{\texttt{2},\texttt{0}\} \parallel \mathit{Pack}\{\texttt{2},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{2},\texttt{0}\} \\
\mathcal{B}[\![\, \mathit{Pack}\{\texttt{1},\texttt{0}\} \parallel \mathit{Pack}\{\texttt{2},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{1},\texttt{0}\} \\
\mathcal{B}[\![\, \mathit{Pack}\{\texttt{2},\texttt{0}\} \parallel \mathit{Pack}\{\texttt{1},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{1},\texttt{0}\} \\
\mathcal{B}[\![\, \mathit{Pack}\{\texttt{1},\texttt{0}\} \parallel \mathit{Pack}\{\texttt{1},\texttt{0}\} \,]\!] &= \mathit{Pack}\{\texttt{1},\texttt{0}\}
\end{aligned}$$

$$\mathcal{B}[\![\, \texttt{0} \,]\!] = 0^{\mathbb{N}} \qquad \mathcal{B}[\![\, \texttt{1} \,]\!] = 1^{\mathbb{N}}\ \ldots$$

$$\mathcal{B}[\![\, \texttt{0.0} \,]\!] = 0^{\mathbb{Q}} \qquad \mathcal{B}[\![\, \texttt{0.1} \,]\!] = \tfrac{1}{10}^{\mathbb{Q}} \ldots$$

The two additional functions sub and div are necessary for arithmetic on natural numbers:

$$\mathit{sub}(x,y) = \begin{cases} 0^{\mathbb{N}} & \mathit{for}\ y >^{\mathbb{Q}} x \\ x -^{\mathbb{N}} y & \mathit{otherwise} \end{cases}$$

$$\mathit{div}(x,y) = \begin{cases} \mathcal{B}[\![\, \texttt{error} \,]\!] & \mathit{for}\ y = 0^{\mathbb{N}} \\ n & \mathit{with}\ n \cdot^{\mathbb{N}} y \leq x\ \mathit{and}\ (n+1) \cdot^{\mathbb{N}} y > x \end{cases}$$

The function round rounds a rational number to the next natural number.

$$\mathit{round}(x) = \begin{cases} x & \mathit{for}\ x \in \mathbb{N} \\ 0^{\mathbb{N}} & \mathit{for}\ x <^{\mathbb{Q}} 0^{\mathbb{Q}} \\ a & \mathit{for}\ \frac{p}{q} <^{\mathbb{Q}} \frac{1}{2}^{\mathbb{Q}}\ \mathit{and}\ x = a + \frac{p}{q}\ \mathit{and}\ p <^{\mathbb{Q}} q,\ a,p,q \in \mathbb{N} \\ a+1 & \mathit{for}\ \frac{p}{q}^{\mathbb{Q}} \geq^{\mathbb{Q}} \frac{1}{2}\ \mathit{and}\ x = a + \frac{p}{q}\ \mathit{and}\ p <^{\mathbb{Q}} q,\ a,p,q \in \mathbb{N} \end{cases}$$

The relations $<^{\mathbb{Q}}$, $\leq^{\mathbb{Q}}$, $=^{\mathbb{Q}}$, $\neq^{\mathbb{Q}}$, $\geq^{\mathbb{Q}}$, $>^{\mathbb{Q}}$ *are the usual orderings on rational numbers, which also order natural numbers.*
The functions $\odot^{\mathbb{Q}}$, $\odot^{\mathbb{Q}} \in \{+^{\mathbb{Q}}, -^{\mathbb{Q}}, \cdot^{\mathbb{Q}}, \div^{\mathbb{Q}}\}$, *and* $\odot^{\mathbb{N}}$, $\odot^{\mathbb{N}} \in \{+^{\mathbb{N}}, -^{\mathbb{N}}, \cdot^{\mathbb{N}}\}$, *are the usual arithmetic operations on rational and natural numbers respectively. For any* x, $x \div^{\mathbb{Q}} z$, $z \in \{0^{\mathbb{N}}, 0^{\mathbb{Q}}\}$, *is* $\mathcal{B}[\![\, \texttt{error} \,]\!]$.

The reason for having addition, subtraction, and multiplication once for natural numbers and once for rationals is to preserve the type of the input: the operations $\odot^{\mathbb{N}}$ have result type $\mathbb{N}$ and $\odot^{\mathbb{Q}}$ have $\mathbb{Q}$. This prevents for example that `0.4 + 0.6 - 2` is interpreted by the *sub*-function.

2.4.4 Transition relation

This section defines the transition relation $\rhd_P$ as the least relation closed under the inference rules listed below.

The inference rules follow the syntax of FATOM and state under which premises (presented above the horizontal bar) a certain transition may occur (presented below the horizontal bar).

All transition rules have in common that they operate on one process, i. e. on a particular expression which satisfies the premise. If several rules match for different processes the evaluation can take place in any order or simultaneously. But concerning a particular process in a given configuration, there is at most one rule, which can be applied.

In the following inference rules, p and s are a set of processes and a constraint store respectively, which form the components of a configuration $\gamma = \langle p, s\rangle$.

Local definitions A local definition is evaluated by substituting the bodies of each binding into the body of the whole **let**-expression.

$$(let)\ \frac{\begin{array}{ll} e & \in p \\ e & = \mathbf{let}\ v_1 = e_1\,;\,\ldots\,; v_n = e_n\ \mathbf{in}\ b \\ & n \geq 1 \\ e' & = b\,[v_1/e_1, \ldots, v_n/e_n] \\ p' & = p \setminus \{e\} \uplus \{e'\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s\rangle}$$

Case analysis A case analysis requires reduction of the expression in question to normal form. In a type correct program, this normal form is a `Pack`-expression. The continuation of the **case**-expression is the branch whose tag number equals the tag number of the normal form. Before evaluation is continued the sub-terms of the `Pack`-expression are substituted into the body of the continuation branch, like in (*let*).

$$(case)\ \frac{\begin{array}{ll} e & \in p \\ e & = \mathbf{case}\ b\ \mathbf{of} \\ & \qquad \{1\}\ v_{11}\ \ldots v_{1a_1}\ \rightarrow e_1\,; \\ & \qquad \vdots \\ & \qquad \{m\}\ v_{m1} \ldots v_{ma_m} \rightarrow e_m \\ \multicolumn{2}{l}{\langle \{b\}, s\rangle \rhd_P^+ \langle \{Pack\{t,a\}\ b_1 \ldots b_a\}, s\rangle} \\ & m \geq 1 \\ & 0 \leq a_i,\ i = 1, \ldots, m \\ & 1 \leq t \leq m \\ & a = a_t \\ e' & = e_t[v_{t1}/b_1, \ldots, v_{ta_t}/b_a] \\ p' & = p \setminus \{e\} \uplus \{e'\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s\rangle}$$

Application of a supercombinator On application of a supercombinator, the combinator's arguments are substituted into its body to form the continuation.

This rule implements evaluation of functional expression by normal order reduction.

$$(appSC)\ \frac{\begin{array}{ll} e & \in p \\ e & = v\ e_1 \ldots e_n \\ & \quad n \geq 0 \\ SC_P(v) & = (v\ v_1 \ldots v_n = b) \\ e' & = b\,[v_1/e_1, \ldots, v_n/e_n] \\ p' & = p \setminus \{e\} \uplus \{e'\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s\rangle}$$

Evaluation of a variable An atomic variable expression that is not the name of a supercombinator is looked up in the store. If it is bound it evaluates to this value, otherwise, the rule is not applicable.

$$(var)\ \frac{\begin{array}{ll} e & \in p \\ e & = v \\ SC_P(v) & = \mathit{undefined} \\ \mathit{lookup}_s(v) & = w \in \mathit{VAL} \\ e' & = w \\ p' & = p \setminus \{e\} \uplus \{e'\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s\rangle}$$

Application of built-in functions Like **case**-expressions, built-in functions and operators force reduction of their arguments to normal form. After argument reduction, the built-in operator is interpreted by the $\mathcal{B}$-function.

$$(appOp)\ \frac{\begin{array}{l} e\ \in p \\ e\ = e_1 \oplus e_2 \\ \langle\{e_1\}, s\rangle \rhd_P^+ \langle\{w_1\}, s\rangle \\ \langle\{e_2\}, s\rangle \rhd_P^+ \langle\{w_2\}, s\rangle \\ \quad w_1, w_2 \in \mathit{VAL} \\ e' = \mathcal{B}[\![\, w_1 \oplus w_2 \,]\!] \\ p' = p \setminus \{e\} \uplus \{e'\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s\rangle} \qquad (appBCP)\ \frac{\begin{array}{l} e\ \in p \\ e\ = b\ e_1 \ldots e_n \\ b\ \in \mathit{Builtin} \cup \\ \quad \mathit{Const} \cup \mathit{Pack} \\ \langle\{e_1\}, s\rangle \rhd_P^+ \langle\{w_1\}, s\rangle \\ \quad \vdots \\ \langle\{e_n\}, s\rangle \rhd_P^+ \langle\{w_n\}, s\rangle \\ \quad w_1, \ldots, w_n \in \mathit{VAL} \\ \quad n \geq 0 \\ e' = \mathcal{B}[\![\, b\ w_1 \ldots w_n \,]\!] \\ p' = p \setminus \{e\} \uplus \{e'\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s\rangle}$$

Introduction of constrained variables A **with**-expression introduces a set of fresh unbound variables into the store and substitutes the names of the fresh variables into the body expression.

$$
(\mathit{with})\ \frac{\begin{array}{l} e \ \in p \\ e \ = \textbf{with } v_1 \ldots v_n \textbf{ in } b \\ \quad n \geq 1 \\ \quad v'_1, \ldots, v'_n \notin \mathit{vars}(s) \\ e' = b\,[v_1/v'_1, \ldots, v_n/v'_n] \\ p' = p \setminus \{e\} \uplus \{e'\} \\ s' = s \uplus \{v'_1 =:= \lozenge, \ldots, v'_n =:= \lozenge\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s'\rangle}
$$

Conjunction A conjunction of expressions spawns a set of new processes.

$$
(\mathit{conj})\ \frac{\begin{array}{l} e \ \in p \\ e \ = e_1 \ \& \ldots \& \ e_n \\ \quad n > 1 \\ p' = p \setminus \{e\} \uplus \{e_1, \ldots, e_n\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s\rangle}
$$

Tell A tell expression drives evaluation of the functional expression to normal form. In a type correct program, this normal form is a ground value or a variable.

If it is a number or variable rule (tellV) applies and the equality $r =:= w$ is augmented to the store.

If the normal form is a constructor expression, rule (tellP) applies. In this case, the store is augmented with the constructor expression but with its (possibly unevaluated) subexpressions replaced by fresh variables. These variables are constrained to be equal to the subexpressions of the constructor. This early constructor propagation to the store is done to speed up creation of processes and to increase the level of concurrency.

$$
(\mathit{tellV})\ \frac{\begin{array}{l} e \ \in p \\ e \ = r =:= b \\ \langle \{b\}, s\rangle \rhd_P^{+} \langle \{w\}, s\rangle \\ w \in \mathit{VAL} \setminus \mathbb{T} \cup \mathit{Var} \\ \mathcal{T}[\![\, r =:= w \,]\!]\, s = s' \neq \mathit{fail} \\ p' = p \setminus \{e\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s'\rangle}
$$

$$(\mathit{tellP})\ \frac{\begin{array}{l} e\ \in p \\ e\ = r =:= b \\ \langle \{b\}, s\rangle \rhd_P^+ \langle \{w\}, s\rangle \\ w = \mathit{Pack}\{t,a\}\, e_1 \dots e_a \\ s' = s \cup \{v_1 =:= \Diamond, \dots, v_n =:= \Diamond\} \\ \qquad v_i \notin \mathit{vars}(s),\ i = 1, \dots, n \\ \mathcal{T}[\![\, r =:= \mathit{Pack}\{t,a\}\, v_1 \dots v_n \,]\!]\, s' = s'' \neq \mathit{fail} \\ p' = p \setminus \{e\} \uplus \{v_1 =:= e_1, \dots, v_n =:= e_n\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s''\rangle}$$

Both rules, (tellV) and (tellP), add strictness to the evaluation, especially (tellP) causes constructors to be strict in all arguments. As soon as a new process is created by rule (conj), its evaluation is enforced. If the result of this process is not required in the remainder of the computation, this leads to undesirable resource consumption. Lazy evaluation of the functional part is of course not compromised by this.

Ask An ask operation is performed on evaluation of a guarded expression. As soon as any of the constraint conjunctions of the guard can be successfully asked, evaluation continues with this guard's body expression. By discarding all other alternatives, this rule leads to don't-care non-determinism (cf. [HW06]).

$$(\mathit{ask})\ \frac{\begin{array}{l} e\ \in p \\ e\ = \quad c_{11}\ \&\ \dots\ \&\ c_{1n_1}\ \Rightarrow e_1 \\ \qquad \mid\ \ \vdots \\ \qquad \mid\ c_{m1}\ \&\ \dots\ \&\ c_{mn_m} \Rightarrow e_m \\ \exists k : 1 \leq k \leq m \text{ and } \mathcal{A}[\![\, c_{k1}\ \wedge\ \dots\ \wedge\ c_{kn_k} \,]\!]\, s = \mathit{ok} \\ e' = e_k \\ p' = p \setminus \{e\} \uplus \{e'\} \end{array}}{\langle p, s\rangle \rhd_P \langle p', s\rangle}$$

2.4.5 Example computations

With the transition relation fully defined, a simple arithmetic computation and two more complex computations resulting from evaluating some of the example programs from Section 2.1 are shown and commented on. They are presented as a chain of configurations related by $\rhd_P$ or $\rhd_P^+$. The transition rule applied in each step is noted below the relation symbol. The process or processes the rule or rules apply to are underlined in each step. If several rules are applied in sequential order, they are separated by a ",", if they may be applied in any order, they are separated by "$\|$". If a certain rule is applied n times this is denoted as a superscript. Sometimes repetitive steps or irrelevant details are omitted and indicated by "...". A computation might look like this:

$$\gamma_0 \underset{(\mathit{tr_0})}{\rhd_P} \gamma_1 \underset{(\mathit{tr_{11}}, \mathit{tr_{12}})}{\rhd_P^+} \cdots \underset{(\mathit{tr_{i-1}})}{\rhd_P} \gamma_i \underset{(\mathit{tr_i}^{2})}{\rhd_P^+} \gamma_{i+1} \underset{(\mathit{tr_{i+1,1}} \| \mathit{tr_{i+1,2}})}{\rhd_P^+} \cdots$$

Several rules, like (*tellV*) or (*case*), require subcomputations because expressions must be reduced to normal form to satisfy the rule's premises. The subcomputation is enclosed in $\ulcorner\ \urcorner\ \llcorner\ \lrcorner$ and appears inside the surrounding computation. It is noteworthy that the shown computations are not the only ones possible for a given initial configuration γ_0, because often several rules may be applied in different order or simultaneously.
Furthermore, not all computations start by evaluation of γ_0 as given in Definition 2.9 but by a more specific expression instead to save some indirection steps.
Unlike the example programs of Section 2.1, the following examples use a `teletyper` font for FATOM programs and expressions for a better distinction between syntactic and semantic values.

Arithmetics

A simple arithmetic calculation might look like this:

$$P = \textbf{def}\ \texttt{main}\ \texttt{r} = \texttt{r} =:= \texttt{5} + \texttt{18}$$

The evaluation of this small program is shown below and illustrates the interpretation of built-in operators and numbers and the propagation of ground values to the store.

$$
\begin{array}{ll}
 & \langle\{\underline{\texttt{main r}}\}, \{\texttt{r} =:= \Diamond\}\rangle \\
\underset{(appSC)}{\rhd_P} & \langle\{\underline{\texttt{r} =:= \texttt{5} + \texttt{18}}\}, \{\texttt{r} =:= \Diamond\}\rangle \\
\underset{(tellV)}{\rhd_P} & \ulcorner \qquad\qquad\qquad\qquad\qquad\qquad \urcorner \\
 & \begin{array}{ll}
 & \langle\{\underline{\texttt{5} + \texttt{18}}\}, \ldots\rangle \\
 \underset{(appOp)}{\rhd_P} & \ulcorner \qquad\qquad\qquad\qquad\qquad \urcorner \\
 & \begin{array}{rcl}
 \langle\{\underline{\texttt{5}}\}, \ldots\rangle & \underset{(appBCP)}{\rhd_P} & \langle\{5^{\mathbb{N}}\}, \ldots\rangle \\
 \langle\{\underline{\texttt{18}}\}, \ldots\rangle & \underset{(appBCP)}{\rhd_P} & \langle\{18^{\mathbb{N}}\}, \ldots\rangle
 \end{array} \\
 & \llcorner \qquad\qquad\qquad\qquad\qquad \lrcorner \\
 & \langle\{23^{\mathbb{N}}\}, \ldots\rangle
 \end{array} \\
 & \llcorner \qquad\qquad\qquad\qquad\qquad\qquad \lrcorner \\
 & \langle\emptyset, \{\texttt{r} =:= 23^{\mathbb{N}}\}\rangle
\end{array}
$$

This is a final configuration and the expected result of 23 is bound to the variable `r`, which is passed to the `main`-supercombinator.

Unbounded buffer

The program P in evaluation is the first example shown in Listing 2.1 with two processes `produce` and `consume` communicating via an unbounded buffer `buf`.

$$\langle\{\underline{\mathtt{prodCon}}\},\emptyset\rangle$$

$$\underset{(appSC)}{\triangleright_P}\quad \langle\{\underline{\textbf{with}\ \mathtt{buf}\ \textbf{in}\ \mathtt{produce\ 0\ buf}}\ \&\ \mathtt{consume\ buf}\},\emptyset\rangle$$

$$\underset{(with)}{\triangleright_P}\quad \langle\{\underline{\mathtt{produce\ 0\ buf'}\ \&\ \mathtt{consume\ buf'}}\},\{\mathtt{buf'} =:= \Diamond\}\rangle$$

$$\underset{(conj)}{\triangleright_P}\quad \langle\{\mathtt{produce\ 0\ buf'},\underline{\mathtt{consume\ buf'}}\},\{\mathtt{buf'} =:= \Diamond\}\rangle$$

$$\underset{(appSC)}{\triangleright_P}\quad \begin{array}{l}\langle\{\underline{\mathtt{produce\ 0\ buf'}},\\ \mathtt{pack\ buf'\ 2\ 2}\ \Rightarrow\ \textbf{with}\ \mathtt{rest}\ \textbf{in}\ \mathtt{rest} =:= \mathtt{tl\ buf'}\ \&\\ \mathtt{consume\ rest}\},\{\mathtt{buf'} =:= \Diamond\}\rangle\end{array}$$

$$\underset{(appSC)}{\triangleright_P}\quad \begin{array}{l}\langle\{\underline{\textbf{with}\ \mathtt{items}\ \textbf{in}\ \mathtt{buf'} =:= \mathtt{Cons\ 0\ items}\ \&\ \mathtt{produce\ 0\ items}},\\ \mathtt{pack\ buf'\ 2\ 2}\ \Rightarrow\ \textbf{with}\ \mathtt{rest}\ \textbf{in}\ \mathtt{rest} =:= \mathtt{tl\ buf'}\ \&\\ \mathtt{consume\ rest}\},\{\mathtt{buf'} =:= \Diamond\}\rangle\end{array}$$

$$\underset{(with)}{\triangleright_P}\quad \begin{array}{l}\langle\{\underline{\mathtt{buf'} =:= \mathtt{Cons\ 0\ items'}\ \&\ \mathtt{produce\ 0\ items'}},\\ \mathtt{pack\ buf'\ 2\ 2}\ \Rightarrow\ \textbf{with}\ \mathtt{rest}\ \textbf{in}\ \mathtt{rest} =:= \mathtt{tl\ buf'}\ \&\\ \mathtt{consume\ rest}\},\{\mathtt{buf'} =:= \Diamond,\mathtt{items'} =:= \Diamond\}\rangle\end{array}$$

$$\underset{(conj)}{\triangleright_P}\quad \begin{array}{l}\langle\{\underline{\mathtt{buf'} =:= \mathtt{Cons\ 0\ items'}},\mathtt{produce\ 0\ items'},\\ \mathtt{pack\ buf'\ 2\ 2}\ \Rightarrow\ \textbf{with}\ \mathtt{rest}\ \textbf{in}\ \mathtt{rest} =:= \mathtt{tl\ buf'}\ \&\\ \mathtt{consume\ rest}\},\{\mathtt{buf'} =:= \Diamond,\mathtt{items'} =:= \Diamond\}\rangle\end{array}$$

$$\underset{(tellP)}{\triangleright_P}\quad \begin{array}{l}\ulcorner\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\urcorner\\ \quad\langle\{\underline{\mathtt{Cons\ 0\ items'}}\},\ldots\rangle\ \underset{(appBCP)}{\triangleright_P}\ \langle\{\mathit{Pack}\{2,2\}\ \mathtt{0\ items'}\},\ldots\rangle\\ \llcorner\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\lrcorner\\ \langle\{\mathtt{h'} =:= \mathtt{0},\mathtt{t'} =:= \mathtt{items'},\mathtt{produce\ 0\ items'},\\ \underline{\mathtt{pack\ buf'\ 2\ 2}\ \Rightarrow\ \textbf{with}\ \mathtt{rest}\ \textbf{in}\ \mathtt{rest} =:= \mathtt{tl\ buf'}\ \&}\\ \underline{\mathtt{consume\ rest}}\},\\ \{\mathtt{buf'} =:= \mathit{Pack}\{2,2\}\ \mathtt{h'\ t'},\mathtt{items'} =:= \Diamond,\mathtt{h'} =:= \Diamond,\mathtt{t'} =:= \Diamond\}\rangle\end{array}$$

$$\underset{(ask)}{\triangleright_P}\quad \begin{array}{l}\langle\{\underline{\mathtt{h'} =:= \mathtt{0}},\underline{\mathtt{t'} =:= \mathtt{items'}},\mathtt{produce\ 0\ items'},\\ \textbf{with}\ \mathtt{rest}\ \textbf{in}\ \mathtt{rest} =:= \mathtt{tl\ buf'}\ \&\ \mathtt{consume\ rest}\},\\ \{\mathtt{buf'} =:= \mathit{Pack}\{2,2\}\ \mathtt{h'\ t'},\mathtt{items'} =:= \Diamond,\mathtt{h'} =:= \Diamond,\mathtt{t'} =:= \Diamond\}\rangle\end{array}$$

$$\underset{(tellV^2)}{\triangleright_P^+}\quad \begin{array}{l}\ulcorner\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\urcorner\\ \quad\langle\{\underline{\mathtt{0}}\},\ldots\rangle\ \underset{(appBCP)}{\triangleright_P}\ \langle\{0^{\mathbb{N}}\},\ldots\rangle\\ \llcorner\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\lrcorner\\ \ulcorner\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\urcorner\\ \quad\langle\{\underline{\mathtt{items'}}\},\{\mathtt{items'} =:= \Diamond,\ldots\}\rangle\ \underset{(var)}{\triangleright_P}\ \langle\{\mathtt{items'}\},\ldots\rangle\\ \llcorner\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\lrcorner\\ \langle\{\mathtt{produce\ 0\ items'},\\ \underline{\textbf{with}\ \mathtt{rest}\ \textbf{in}\ \mathtt{rest} =:= \mathtt{tl\ buf'}\ \&\ \mathtt{consume\ rest}}\},\\ \{\mathtt{buf'} =:= \mathit{Pack}\{2,2\}\ \mathtt{h'\ t'},\mathtt{items'} =:= \Diamond,\\ \mathtt{h'} =:= 0^{\mathbb{N}},\mathtt{t'} =:= \mathtt{items'}\}\rangle\end{array}$$

$$\underset{(with,conj)}{\triangleright^{+}_{P}} \quad \langle\{\texttt{produce 0 items}', \underline{\texttt{rest}' =:= \texttt{tl buf}'}, \texttt{consume rest}'\},$$
$$\{\texttt{buf}' =:= \mathit{Pack}\{2,2\}\ \texttt{h}'\ \texttt{t}', \texttt{items}' =:= \lozenge,$$
$$\texttt{h}' =:= 0^{\mathbb{N}}, \texttt{t}' =:= \texttt{items}', \texttt{rest}' =:= \lozenge\}\rangle$$

$$\underset{(tellV)}{\triangleright_{P}} \quad \ulcorner \qquad \urcorner$$

$$\langle\{\underline{\texttt{tl buf}'}\}, \{\texttt{buf}' =:= \mathit{Pack}\{2,2\}\ \texttt{h}'\ \texttt{t}', \ldots\}\rangle$$

$$\underset{(appSC)}{\triangleright_{P}} \quad \langle\{\underline{\textbf{case}\ \texttt{buf}'\ \textbf{of}\ \{2\}\ \texttt{x xs} \rightarrow \texttt{xs}}\},$$
$$\{\texttt{buf}' =:= \mathit{Pack}\{2,2\}\ \texttt{h}'\ \texttt{t}', \ldots\}\rangle$$

$$\underset{(case)}{\triangleright_{P}} \quad \ulcorner \qquad \urcorner$$

$$\langle\{\underline{\texttt{buf}'}\}, \{\texttt{buf}' =:= \mathit{Pack}\{2,2\}\ \texttt{h}'\ \texttt{t}', \ldots\}\rangle$$

$$\underset{(var)}{\triangleright_{P}} \quad \langle\{\mathit{Pack}\{2,2\}\ \texttt{h}'\ \texttt{t}'\}, \ldots\rangle$$

$$\llcorner \qquad \lrcorner$$

$$\langle\{\texttt{t}'\}, \ldots\rangle$$

$$\llcorner \qquad \lrcorner$$

$$\langle\{\texttt{produce 0 items}', \texttt{consume rest}'\},$$
$$\{\texttt{buf}' =:= \mathit{Pack}\{2,2\}\ \texttt{h}'\ \texttt{t}', \texttt{items}' =:= \lozenge,$$
$$\texttt{h}' =:= 0^{\mathbb{N}}, \texttt{t}' =:= \texttt{items}', \texttt{rest}' =:= \texttt{t}'\}\rangle$$

$$\triangleright_{P} \quad \ldots$$

Divide and conquer setting

This example uses the parallel version of `map` as shown in Listing 2.6. It differs from the previous example in that it has a final configuration.
The effect of this example is to compute each element's successor of the list `Cons1 (Cons 2 Nil)`.
This time, the details are left out and only the important parts are shown: the `farm` coordination spawns processes for each application of `succ` that are evaluated in parallel.

$$\langle\{\underline{\textbf{with}\ \texttt{r}\ \textbf{in}\ \texttt{farm succ (Cons 1 (Cons 2 Nil)) r}}\}, \emptyset\rangle$$

$$\triangleright^{+}_{P} \quad \langle\{\texttt{r}' =:= \texttt{Cons (succ 1) rs}', \underline{\texttt{farm succ (Cons 2 Nil) rs}'}\},$$
$$\{\texttt{r}' =:= \lozenge, \texttt{rs}' =:= \lozenge\}\rangle$$

$$\triangleright^{+}_{P} \quad \langle\{\texttt{r}' =:= \texttt{Cons (succ 1) rs}', \texttt{rs}' =:= \texttt{Cons (succ 2) rs}'',$$
$$\underline{\texttt{farm succ Nil rs}''}\},$$
$$\{\texttt{r}' =:= \lozenge, \texttt{rs}' =:= \lozenge, \texttt{rs}'' =:= \lozenge\}\rangle$$

$$\triangleright^{+}_{P} \quad \langle\{\underline{\texttt{r}' =:= \texttt{Cons (succ 1) rs}'}, \underline{\texttt{rs}' =:= \texttt{Cons (succ 2) rs}''}\},$$
$$\{\texttt{r}' =:= \lozenge, \texttt{rs}' =:= \lozenge, \texttt{rs}'' =:= \mathit{Pack}\{1,0\}\}\rangle$$

$$\triangleright^{+}_{P} \quad \langle\{\underline{\texttt{rh}' =:= \texttt{succ 1}}, \texttt{rt}' =:= \texttt{rs}', \underline{\texttt{rsh}' =:= \texttt{succ 2}}, \texttt{rst}' =:= \texttt{rs}''\},$$
$$\{\texttt{r}' =:= \mathit{Pack}\{2,2\}\ \texttt{rh}'\ \texttt{rt}', \texttt{rs}' =:= \mathit{Pack}\{2,2\}\ \texttt{rsh}'\ \texttt{rst}',$$
$$\texttt{rs}'' =:= \mathit{Pack}\{1,0\}, \texttt{rh}' =:= \lozenge, \texttt{rt}' =:= \lozenge, \texttt{rsh}' =:= \lozenge, \texttt{rst}' =:= \lozenge\}\rangle$$

$$\begin{array}{ll}\rhd^+_P & \langle\{\underline{\mathtt{rt}' =:= \mathtt{rs}'}, \underline{\mathtt{rst}' =:= \mathtt{rs}''}\},\\ & \{\mathtt{r}' =:= \mathit{Pack}\{\mathtt{2},\mathtt{2}\}\ \mathtt{rh}'\ \mathtt{rt}', \mathtt{rs}' =:= \mathit{Pack}\{\mathtt{2},\mathtt{2}\}\ \mathtt{rsh}'\ \mathtt{rst}',\\ & \mathtt{rs}'' =:= \mathit{Pack}\{\mathtt{1},\mathtt{0}\}, \mathtt{rh}' =:= 2^{\mathbb{N}}, \mathtt{rt}' =:= \Diamond, \mathtt{rsh}' =:= 3^{\mathbb{N}}, \mathtt{rst}' =:= \Diamond\}\rangle\\ \rhd^+_P & \langle\emptyset, \{\mathtt{r}' =:= \mathit{Pack}\{\mathtt{2},\mathtt{2}\}\ \mathtt{rh}'\ \mathtt{rt}', \mathtt{rs}' =:= \mathit{Pack}\{\mathtt{2},\mathtt{2}\}\ \mathtt{rsh}'\ \mathtt{rst}',\\ & \mathtt{rs}'' =:= \mathit{Pack}\{\mathtt{1},\mathtt{0}\}, \mathtt{rh}' =:= 2^{\mathbb{N}}, \mathtt{rt}' =:= \mathtt{rs}', \mathtt{rsh}' =:= 3^{\mathbb{N}}, \mathtt{rst}' =:= \mathtt{rs}''\}\rangle\end{array}$$

The result of the computation is bound to $\mathtt{r}'$ which was introduced by the **with**-construct. Following the terms in the store,

$$\mathtt{r}' =:= \mathit{Pack}\{\mathtt{2},\mathtt{2}\}\ 2^{\mathbb{N}}\ (\mathit{Pack}\{\mathtt{2},\mathtt{2}\}\ 3^{\mathbb{N}}\ \mathit{Pack}\{\mathtt{1},\mathtt{0}\})$$

is found, which is the list `Cons 1 (Cons 2 Nil)` with each element incremented by one.

Chapter 3

The abstract machine ATAF

This chapter describes the target machine for FATOM programs called ATAF. It is an abstract machine that supports lazy evaluation, processes and coordination/ cooperation by constraints.

The main goal of the machine's design is to offer a suitable infrastructure for FATOM's operational semantics to be easily implemented on a real machine. A straightforward implementation will not be the most efficient one but serves well as a proof of concept. It is of course possible to gain performance by a more sophisticated implementation and a longer development process.

This chapter is composed of four sections:

- Lazy evaluation of functional expressions will be done by graph reduction, which is reviewed in Section 3.1.
- Sections 3.2 and 3.3 describe the ATAF abstract machine itself. The first section gives an overview of its components and their purpose, whereas the second section defines the details of the machine instructions, i. e. the operational semantics.
- The last section explains how FATOM programs are compiled into ATAF machine code by a series of compilation schemes.

3.1 Review of graph reduction

One half of the ATAF machine is concerned with lazy evaluation of functional expressions. An efficient implementation technique for lazy functional languages is graph reduction, which is described informally in this section. An in-depth treatment of functional language implementation by graph reduction can be found in the Ph. D. theses of Johnsson [Joh87] and Augustsson [Aug87].

A formal definition of the graph reduction mechanisms built into ATAF is presented in the following two sections, which give a fully detailed specification of a G-machine.

Note: One might ask, why a G-machine is used to implement the functional part of FATOM as there are fancier and more efficient solutions to this problem like the Spineless Tagless G-machine [PS89, Pey92] or the Three Instruction Machine [FW87, Arg89]. The reason is purely practical: a G-machine is less work to implement and sufficient for a prototype. It is certainly possible to replace the G-machine part by a more sophisticated machine like the STG-machine or the TIM if the necessity should arise.

3.1.1 Lazy evaluation

There are two ingredients for lazy evaluation of functional programs:

- normal order reduction and
- updates.

Normal order reduction is one possible evaluation strategy of the Lambda-Calculus (see e. g. [Bar84]). Its key property is the following: if there exists a normal form of a lambda-expression at all, normal order reduction leads to this normal form. This property also holds for an enriched set of lambda-expressions like FATOM.
From a language implementor's point of view, normal order reduction manifests itself in the fact that arguments are passed unevaluated to a function as defined in transition rules (*appSC*), (*let*), and (*case*). This semantics is also known as call-by-need because expressions are only evaluated if they needed in normal form.
The second ingredient comes into play in expressions like

$$\begin{array}{l}\textbf{let } x = e \\ \textbf{in } f\ x\ x\end{array}$$

with e being an arbitrary (possibly very complex) expression and f being some function. If ever in the program's evaluation the necessity arises to reduce f's arguments to normal form this work should certainly not be done twice. In a lazy evaluation scheme, this multiple work is avoided by replacing x with a reference to the normal form of x. This replacement is called *updating* and is an important optimisation for lazy functional languages. As functional languages lack any side-effects it is guaranteed that a re-evaluation of x leads to no other result than the first evaluation.
There is a qualitative difference between updating and normal order reduction as the latter can be easily expressed in the Lambda-Calculus in a state-less fashion, whereas updating requires some kind of mutable state in which a part of an expression can be replaced by a reference to another one.
Of course, every state transition system can be expressed in a purely state-less manner by passing around the state as additional argument to all state-transforming functions. But with regard to an implementation on a real machine, this is unnecessarily inefficient and the remainder of this chapter is kept in a slightly imperative fashion with mutable state.
During the examples of graph reduction in this section, the simple pure functional program of Listing 3.1 will be used as running example.

3.1.2 Data structures for graph reduction

Graph reduction requires two data structures: a heap and a stack.
The heap is used to store the expression evaluated as a graph. The stack stores pointers to certain interesting parts of this graph.
The graph contains different kinds of nodes:

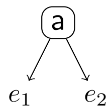
Listing 3.1 Simple functional example program to illustrate graph reduction.

$$
\begin{aligned}
&\textit{compose}\ f\ g\ x &&= f\ (g\ x)\\
&\textit{twice}\ f\ x &&= \textit{compose}\ f\ f\ x\\
&\textit{double}\ x &&= x + x
\end{aligned}
$$

- Supercombinators and built-in functions are represented like this:

  $\boxed{\textit{twice}}$ $\boxed{+}$

- Application of two expressions e_1 and e_2 is indicated by an application node:

  The directed edges from $\boxed{\mathsf{a}}$ to e_1 and e_2 should be interpreted as pointers to the graphs representing the expressions e_1 and e_2.

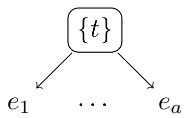
- Natural and floating point numbers are shown like this:

  $\boxed{\mathsf{23}}$ $\boxed{\texttt{1.6e-19}}$

- Indirection nodes are nodes that point to other graphs. They are needed for updating (see Section 3.1.4) and look like a rotated return key:

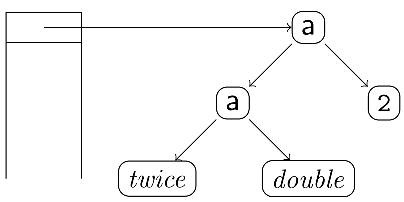
  $\boxed{\Rsh}$

- Constructors are represented by a pack-node:

  The number in curly braces is the constructor's tag, and $e_1, \dots, e_a$ are the graphs representing the sub-terms.

The state of a graph reduction is determined by the content of the stack and the heap. The contents of the stack and heap will be shown as in the next drawing for the simple expression *twice double* 2:



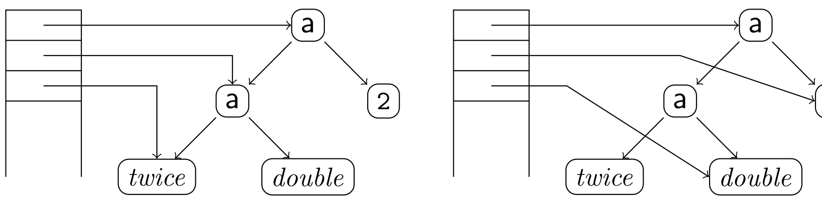
As indicated in this drawing, the stack grows to the bottom of the page.

3.1.3 Finding the next redex

Following normal order reduction, the left-most outer-most expression has to be reduced next in an evaluation. The next candidate for reduction is called *redex*, which is short for reducible expression. In the graph representation of expressions, this redex is always found by traversing the left edges of application nodes until a function node is found. The resulting path is called the *spine* of an expression, the process of finding the next redex is called *unwinding the spine*. During unwinding, a pointer to each new node that comes along is pushed onto the stack.

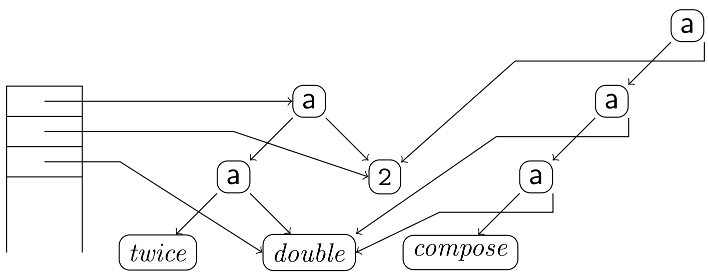
Once the next redex is found the stack contains pointers to all its application nodes. The arguments to a function are the right successors of these application nodes. For straight access to the arguments, these pointers are rearranged to point directly to the arguments but retaining a pointer to the n^{th} application node with n being the functions arity. The n^{th} application node is the root of the redex, and a pointer must be kept for updating.

The next drawing shows the unwound spine before and after the pointer rearrangement for the running example:

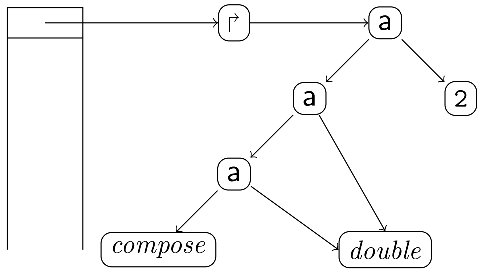


3.1.4 Instantiating a supercombinator

Evaluation continues by instantiating a copy of the supercombinator's body. Instantiation means substitution of parameters by pointers to the graphs of the arguments:



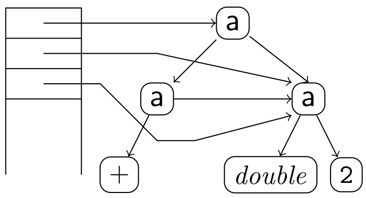
The next step is to discard all pointers on the stack to arguments of *twice* and updating the root of the redex. Updating is done by overwriting the root of the redex with an indirection node pointing to the newly instantiated graph. There are no pointers left to the spine of the old graph. It has become garbage and is silently ignored from now on:



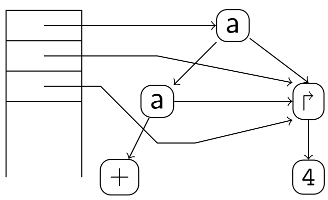
Evaluation continues with further unwinding the spine and instantiation of the *compose* supercombinator. If the evaluation finds a pointer to an indirection node on the stack this indirection is “short-circuited”, and a pointer to the indirection’s target is pushed on the stack. The indirection node remains in the heap because surrounding graphs may have pointers to it.

3.1.5 Evaluating built-in operators

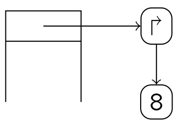
After instantiation of the *compose* and *double* supercombinators and continued unwinding the state of evaluation is as shown:



It is important that both argument pointers of + reference the same graph. If this was not the case *double* 2 had to be evaluated twice.
As + is built into the language there is no body to instantiate. Furthermore, only numbers can be added and it is therefore necessary to evaluate all arguments of + to normal form, which has to be a number in type correct programs, i. e. built-in functions impose a strict context.
The expression *double* 2 is evaluated on a fresh stack and its root is updated as usual:



Apart from the indirection node both arguments to + are now in normal form and can be added. The final state for this computation leaves a pointer to the expected result of `8` on the stack:



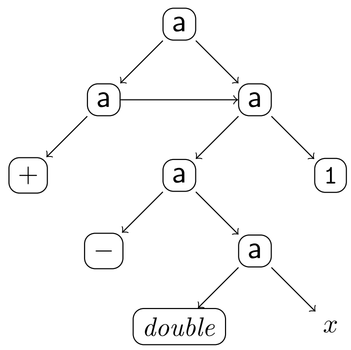
3.1.6 Graph reduction of let- and case-expressions

The language elements left are **let**- and **case**-expressions. **let**-expressions are interpreted as a textual description of a graph; **case**s impose a strict context on the evaluation of the compound expression, like built-in operators.

let-expressions Given the following supercombinator f:

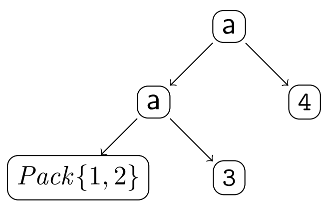
$$\begin{aligned}\textbf{def}\ f\ x = \textbf{let}\ & v = \mathit{double}\ x - 1 \\ \textbf{in}\ & v + v\end{aligned}$$

The result of instantiating f's body is a graph, where the local variable v has disappeared and is replaced by pointers:

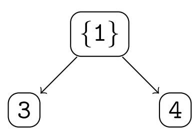


In this graph, x represents the graph passed as argument to f.

case-expressions For a case analysis, it is necessary to evaluate the expression in question to normal form, which results in a constructor on top of the stack. Therefore, it is necessary to have a look at the evaluation of a constructor application first.
A constructor application like $\mathit{Pack}\{1,2\}\ 3\ 4$ leads to the graph



Similar to evaluation of built-in functions there is no body to instantiate but instead a pack-node is created with `Pack`'s arguments as sub-terms and the corresponding tag:

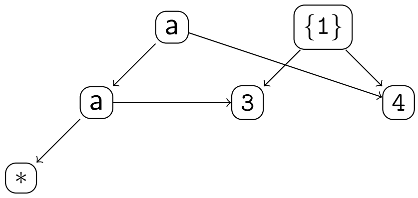


As for function application, the root of the redex is updated using an indirection node.

Having reduced the compound expression in a case analysis to a pack-node, the corresponding branch is instantiated with the local variables replaced by pointers to the constructor's sub-terms. Evaluating the expression

$$\begin{array}{l} \textbf{let } v = \textbf{let } v = \mathit{Pack}\{1,2\}\ 3\ 4 \\ \qquad\quad \textbf{in case } v \textbf{ of} \\ \qquad\qquad\quad \{1\}\ \mathit{fst}\ \mathit{snd} \to \mathit{fst} * \mathit{snd} \end{array}$$

results in the following graph:



It should be noticed that there is no such thing like a case-node because only the alternative selected by the constructor's tag is ever constructed and instantiated. This concludes the exemplified review of graph reduction. Some details have been left out (like additional strict contexts) but they are considered in Section 3.4 when code generation for the G-machine part of ATAF is examined.

3.2 Design of ATAF

This section is about the overall design of the abstract machine ATAF, its different components and its relationship to real single- and multiprocessor machines. Although most machine instructions are specified in Section 3.3.4, some of them are so closely related to the structures introduced in this section that it is more concise to present them earlier.

Instructions are defined in an imperative pseudo-code. The reason for this is twofold: firstly, the implementation described in Chapter 4 is programmed in a monadic style which closely resembles the pseudo-code, i. e. the gap between specification and implementation is small; secondly, presenting the instructions in a state-transition system is less compact as most state components remain unchanged, thus containing lots of redundancy. Nevertheless, Section 3.3, especially 3.3.4, relates the pseudo-code to the underlying state-transition system forming the operational semantics.

3.2.1 The territory of processes

A process is one of the basic concepts of FATOM's evaluation model as described in Section 2.4. This concept is directly represented in the machine's structure: for each process a certain space in memory is reserved that hosts this process' heap and stack. The purpose of these two areas is exactly as introduced in the last section: they are required for graph reduction.

Two more areas of memory are shared between all processes:

- a code segment which stores the machine instructions of the program loaded into the machine and

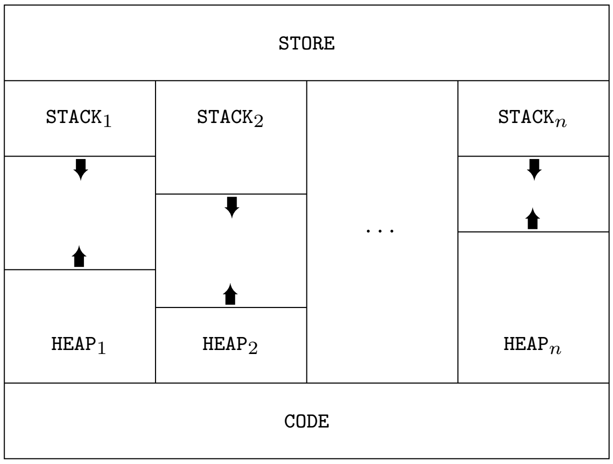


Figure 3.1 An overview of the memory architecture of ATAF for n processes.

- the constraint store which is used for cooperation and communication between processes.

Figure 3.1 shows the four different areas of memory of each process. Many details to be introduced in the subsequent parts of this section are still left out. As indicated by the arrows heap and stack grow towards each other.
Each time a new process is spawned a new heap and stack are created for this process. Similarly, when a process finishes execution its stack and heap are removed.

Note: The memory layout of ATAF differs from many common concurrent and parallel implementations like e. g. the JAVA virtual machine where all threads share a common heap. A shared heap is not necessary because inter-process communication is done by constraints and the constraint store.

Running in parallel

If the abstract machine is running simultaneously on several processors or machines (processing elements), which have a common interconnect like shared memory or a network, each processing element has a set of processes to run. In this case, there is a true parallel evaluation, but as there may be more processes than processing elements processes have to share the real machines, i. e. the processes are executed in an interleaving fashion. This situation is illustrated by Figure 3.2 for *noPE* processing elements.
The number of processing elements is constant during runtime of the machine and the decision about its size is made on startup of ATAF. The procedure of processor allocation depends on the particular host architecture the abstract machines run on.
Despite running on several, possibly very loosely coupled machines, all processes have access to the store independent of their actual host and to certain other structures on other processing elements. It is left as a problem of implementation how to establish these connections as it is highly architecture dependent. A description of a prototypical implementation using MPI is described in Chapter 4.

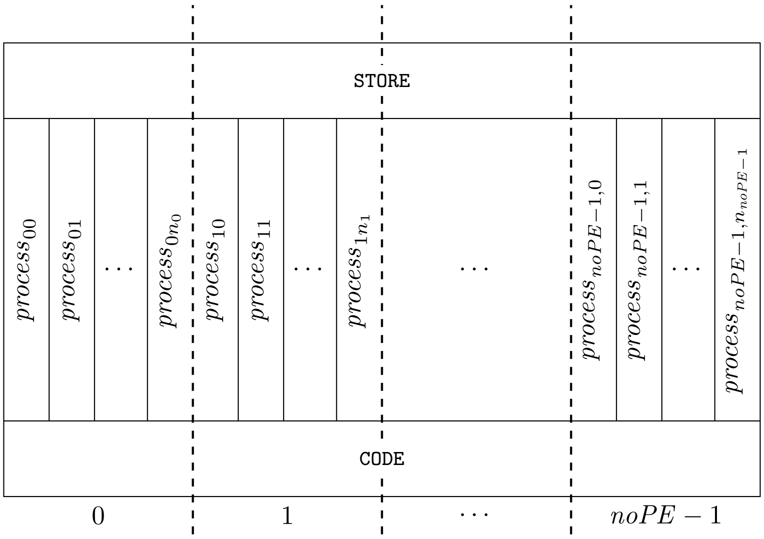


Figure 3.2 Running ATAF in parallel on *noPE* processing elements. The dashed lines indicate real machine boundaries.

3.2.2 The machine memory

A single process has access to four different memory areas:

- the code segment which is read-only and shared by all processes,
- a heap which is a private writable segment,
- a stack which is a private area managed in the usual last-in, first-out discipline, and
- the constraint store which is a shared segment with strong synchronisation restrictions.

The memory is divided into words of equal size and each word is accessible by an address. The structure of words and access to memory is stated precisely by the next definition:

Definition 3.1 *(Memory, word, address space)*
The memory *of* ATAF *is an array of words numbered from a to b*

$$MEM_a^b := ARRAY_a^b\ Word$$

The set $\{0,\ldots,N\}$ *is called Addr and contains all valid memory addresses. The entirety of all addresses a process has access to is called its* address space*.*
Each word *stores a tagged value. The set of words is defined as*

$$
\begin{array}{lcll}
\textit{Word} & \rightarrow & \langle \mathsf{PTR} \mid \textit{Addr} \rangle & \textit{(Pointer)} \\
& \mid & \langle \mathsf{PCK} \mid \textit{Nat Addr}^* \rangle & \textit{(Constructor)} \\
& \mid & \langle \mathsf{NAT} \mid \textit{Nat} \rangle & \textit{(Natural number)} \\
& \mid & \langle \mathsf{FLT} \mid \textit{Float} \rangle & \textit{(Floating point number)} \\
& \mid & \langle \mathsf{APP} \mid \textit{Addr Addr} \rangle & \textit{(Application node)} \\
& \mid & \langle \mathsf{FUN} \mid \textit{Nat Addr} \rangle & \textit{(Function pointer)} \\
& \mid & \langle \mathsf{OPC} \mid \textit{Instr} \rangle & \textit{(Machine instruction)} \\
& \mid & \langle \mathsf{NIL} \rangle & \textit{(Empty word).}
\end{array}
$$

Nat, *Nat*, and *Float* are the sets of numbers as defined in Section 2.2, *Instr*, the instructions of ATAF, is defined in Section 3.3.4.

3.2.3 The process infrastructure

For the evaluation and scheduling of processes, it is necessary that each process can be uniquely identified across all running machine instances.

Definition 3.2 *(Machine instance, process identifier, (un)interruptible process)* *Each invocation of* ATAF *is called a* machine instance *and all instances are numbered from* 0 *to* $noPE - 1$ *with* $noPE$ *being the number of processing elements. Each process running on a certain instance is identified by an instance unique number paired with the instance number to form the* process identifier $pid \in PID$*:*

$$PID := \{(\!|\,, \langle\!|\,\} \times \{0, \ldots, noPE - 1\} \times \mathbb{N} \cup (\!|\cdot|\!)$$

For a machine instance number i *and an instance unique identifier* p *a process identifier* $\langle (\!|, i, p \rangle$ *is written as* $(\!|\, i, p \,|\!)$ *and denotes an* interruptible *process. Likewise, a process identifier* $\langle \langle\!|, i, p \rangle$ *is written as* $\langle\!|\, i, p \,|\!\rangle$ *and denotes an* uninterruptible *process.*
The symbol $(\!|\cdot|\!)$ *represents the empty process.*

The previous definition introduces uninterruptible processes $\langle\!|\, i, p \,|\!\rangle$. These processes get processing time until they set themselves to interruptible again. This distinction is important to keep blocking times of the store short (Section 3.2.4) and has to be regarded during process scheduling (Section 3.3.2).
Not all processes can be evaluated simultaneously: some may be waiting at a guard, others wait for write access to the store, and most processes are ready for evaluation but wait for processing time. To manage the states of these processes queues are used which are lists with a special set of operations:

Definition 3.3 *(Queue)*
A queue *is a list of process identifiers:*

$$QUEUE := LIST\ PID$$

Figure 3.3 shows the operations defined for queues.

To handle empty queues properly it is important that `dequeue` returns the empty process $(\!|\cdot|\!)$ if there is nothing to dequeue.

dequeue $q \equiv$
  IF $q = a : q'$ THEN
    $q := q'$
    RETURN a
  ELSE
    RETURN $(\!|\cdot|\!)$
  END IF

enqueue a $q \equiv$
  $q := q +\!\!+ [a]$

queueJump a $q \equiv$
  $q := a : q$

isEmpty $q \equiv$
  IF $q = []$ THEN
    RETURN *false*
  ELSE
    RETURN *true*
  END IF

Figure 3.3 Operations for queues.

Each machine instance has a `Ready` queue keeping all processes that are ready to run. The process which is currently evaluated is stored in a register `Run`. After a certain time, the process at head of the `Ready` queue is dispatched for evaluation, and the previously running process is enqueued in the `Ready` queue. Chapter 2 explained that processes may suspend in a number of circumstances. The structures for suspended processes are introduced in the next section.

For scheduling purposes, a global instruction counter `IC` is incremented each time a process executes an instruction. If the instruction counter exceeds a certain value NI, the active process is reinserted in the `Ready` queue and another process is executed with a reset counter. Details about scheduling are found in Section 3.3.2.

Apart from the instance global `Ready` queue and `Run` register, each process has a set of local registers to manage its local memory, i. e. its store and its heap:

- The stack pointer `SP` points to the next free word of the stack, and the stack base `SB` points to the highest address of the stack, which is the first stack word (the stack grows towards smaller address numbers).
- The heap pointer `HP` points to the next free word of the heap, and the first heap word is indicated by the heap base register `HB`.
- The instruction pointer `IP` references the currently executed instruction in the code segment. The first instruction word is indicated by `IB`.

For manipulation of local memory, a set of operations is provided similar to the operations on queues:

- The usual stack operations push and pop to put one more word on the stack and remove the top word, respectively. Additionally, top returns the top word but, unlike pop, does not remove it from the stack. nth returns the n^{th} word from the stack without changing its contents.
- To reserve a word in the heap, allocate is provided, which returns the address of the new word.

These operations modify the local registers if necessary; their definition is shown in Figure 3.4. For convenience, decrement and increment operations dec and inc are also defined.

The structure of an entire machine instance is illustrated in Figure 3.5. To distinguish the local registers and memories of the processes, they are subscripted

$$\mathsf{dec}\ r \equiv r := r - 1 \qquad \mathsf{inc}\ r \equiv r := r + 1$$

$$\mathsf{nth}\ n \equiv \text{RETURN}\ \texttt{STACK}[\texttt{SP} + 1 + n] \qquad \mathsf{top} \equiv \text{RETURN}\ \mathsf{nth}\ 0$$

$\mathsf{pop} \equiv$	$\mathsf{push}\ v \equiv$	$\mathsf{allocate} \equiv$
$\mathsf{inc}\ \texttt{SP}$	$\texttt{STACK}[\texttt{SP}] := v$	$\mathsf{inc}\ \texttt{HP}$
$\text{RETURN}\ \texttt{STACK}[\texttt{SP}]$	$\mathsf{dec}\ \texttt{SP}$	$\text{RETURN}\ \texttt{HP} - 1$

Figure 3.4 Operations for registers, stacks, and heaps.

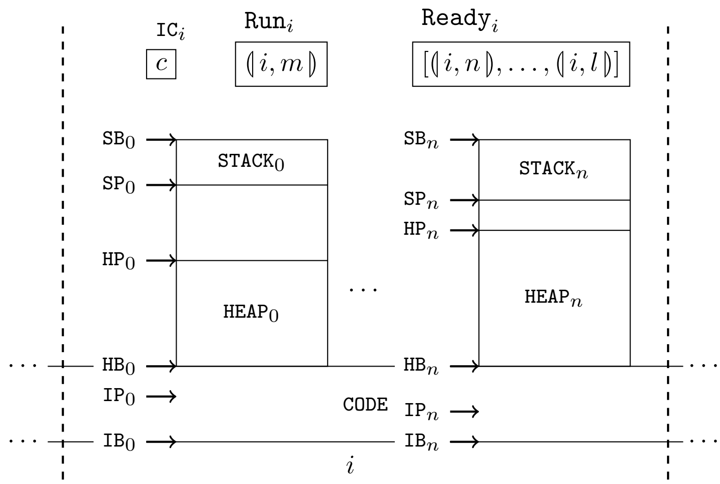


Figure 3.5 A complete machine instance i with $n+1$ processes without store.

with the second component of the process identifier. Similarly, the `Run` register and `Ready` queue have the instance number as indices.
In the snapshot of Figure 3.5, process $(\!| i, m |\!)$ is currently evaluated and process $(\!| i, n |\!)$ will presumably be the next one to get processing time.

3.2.4 The constraint store and its periphery

Operating the constraint store consists of three tasks:

- As the constraint store is shared between all processes of all instances, access has to be synchronised. This is done via mutual exclusion.
- Processes waiting for an event to happen have to be managed via `Suspend` queues.
- As the store also is a memory area, a minimal storage management has to be installed.

Mutual exclusion

To ensure consistency of the store access must be mutually exclusive.

The easiest way to achieve mutual exclusion is a global flag register `TF` that indicates if access to the store is possible. This register is set to *true* if no process is working on the store and *false* otherwise. To avoid race conditions two atomic instructions `LOCK` and `UNLOCK` are provided to test and set this register. Atomic means that at most one process is able to execute a `LOCK` or `UNLOCK` instruction.
If a process executes a `LOCK` instruction and finds the store busy another process should continue its execution. For this reason, the `LOCK` instruction enqueues the current process in a `Blocked` queue, which holds all processes waiting for access to the store. In turn, when an `UNLOCK` instruction is executed, the next waiting process is dequeued and inserted into its `Ready` queue. As there may be more processes waiting in the `Blocked` queue, it is crucial that the freshly released process frees the store as soon as possible. This is encouraged by the `UNLOCK` instruction by insertion at the front of the `Ready` queue. A process released from `Blocked` is allowed to do some queue-jumping. This is possibly an interesting point to investigate different queueing strategies and to compare their performance.
Figure 3.6 shows the definition of `LOCK` an `UNLOCK`. An important point is that a dequeued process is transferred to the `Ready` queue it came from. In ATAF, processes never change their instance. Apart from queue-jumping, uninterruptible execution is also responsible for short locking times of the store: a process successfully locking the store sets itself to uninterruptible and undoes this state with the corresponding `UNLOCK` instruction. The same is true for a process, which enqueues itself into `Blocked`. The last line of the `LOCK` and `UNLOCK` instruction increments the instruction pointer to execute the next instruction.

Note: Mutual exclusion for the store is a classical reader-writer problem. There are better solutions than the one presented in this paragraph which allow several readers at the same time in the critical section, thus shortening locking times. Furthermore, it is possible to either prefer readers or writers, see [Har98] for a detailed discussion. For ATAF, preference of writers would be desirable because changes in the store in general wake up suspended processes, which results in a tighter cooperation.
It is likely that a transaction based scheme is better suited to this synchronisation problem than mutual exclusion. A transaction based scheme approaches the critical section in an optimistic manner and all processes perform their action in this section and check if a conflict occurred after completion. If there was a conflict the whole transaction is rolled back and restarted. For situation in which conflicts are fairly rare, this synchronisation technique outperforms the classical mutual exclusion which, due to its pessimism, often serialises computation unnecessarily. Software transactional memory is one possible solution and e. g. discussed in [ST95].

Suspended processes

If a process inspects a guarded expression and finds no guard fulfilled it suspends until some variables change and the constraints may be fulfilled. The same is true for functional expressions containing unbound variables.
To keep track of processes that wait for certain variables each word of the store has a `Suspend` queue associated with it. All processes waiting for a change of a variable are enqueued in the queue associated with the word the variable is stored in.
As soon as another process changes a variable, all processes in the associated queue are moved to their `Ready` queues to re-evaluate the expressions they sus-

```
LOCK ≡
    ⦇i, p⦈ := Run
    IF TF THEN
        TF := false
        Run := ⟨| i, p |⟩
        inc IP
    ELSE
        enqueue ⟨| i, p |⟩ Blocked
        inc IP
        Run := dequeue Ready
    END IF

UNLOCK ≡
    unlock
    inc IP

unlock ≡
    ⟨| i, p |⟩ := Run
    Run := ⦇i, p⦈
    IF isEmpty Blocked THEN
        TF := true
    ELSE
        ⟨| i, p |⟩ := dequeue Blocked
        queueJump ⟨| i, p |⟩ Ready_i
    END IF
```

Figure 3.6 Definition of LOCK and UNLOCK instructions.

```
allocateVars n ≡
    TP := TP + n
    RETURN [TP − n, TP − n + 1, . . . , TP − 1]
```

Figure 3.7 allocateVars reserves a number of words in the store.

pended on. These processes may be waiting in additional Suspend queues from which they have to be removed to avoid that they appear several times the Ready queue.

The store is part of each process' address space, i. e. all words in the store are accessible by STORE$[k]$ with k being a valid address. The queue associated with STORE$[k]$ is therefore called Suspend$[k]$.

Storage management

The store is filled from lower to higher addresses similar to the heaps. A store base register TB points to the first word of the store and a store pointer TP indicates the next free word (they are called T· because S· are the stack registers and T is the next letter).

To reserve a number of words in the store an operation allocateVars is provided, which returns a list of addresses (see Figure 3.7). This operation has to be used mutually exclusive of course, i. e. between a LOCK, UNLOCK pair of instructions.

Figure 3.8 shows the store with its periphery. N is the last available address as in Definition 3.1.

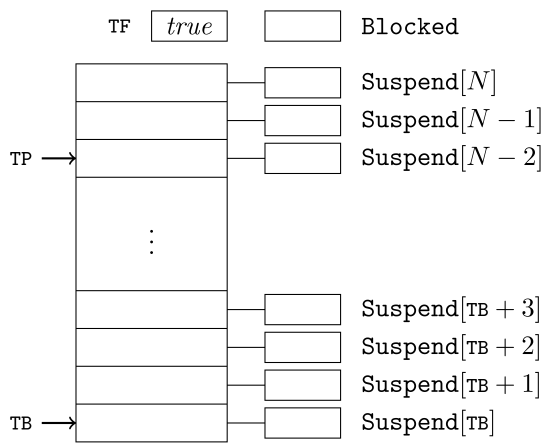


Figure 3.8 The constraint store with all queues and registers.

3.3 Instruction set and operational semantics

It is now time to unite the different bits and pieces of the last section into a single machine state and to give the rules how one state passes into the next one. Therefore, some operations to create, set up, and terminate processes are introduced before this section ends with the definition of the instruction set of ATAF.

3.3.1 Machine state

A machine state puts together the shared code and store segments and the instances:

Definition 3.4 *(Machine state)*
A machine state *is a tuple of a noPE machine instances $\mathcal{M}$, a code segment $\mathcal{I}$, and a constraint store $\mathcal{S}$:*

$$STATE := \mathcal{M}^{noPE} \times \mathcal{I} \times \mathcal{S}$$

The next definitions give the meanings of the components of a state.

Definition 3.5 *(Instance)*
An instance *consists of a process identifier, a queue, and a set of processes:*

$$\mathcal{M} := \mathbb{N} \times PID \times QUEUE \times \mathbb{P}\mathcal{P}$$

The first component is the instruction counter, the second component the `Run` *register, the third one the* `Ready` *queue, and the last component are all stacks, heaps, and registers of the processes.*

Definition 3.6 *(Process)*
A process consists of an identifier, a register set, and a memory area:

$$\mathcal{P} := PID \times \{ib\} \times Addr \times \{hb\} \times Addr \times \{sb\} \times Addr \times MEM_{hb}^{sb}$$

ib, hb, and sb are the code, heap, and stack base registers. Their right neighbours are the corresponding pointers IP, HP, *and* SP. *The last component is the process' stack and heap area.*

Definition 3.7 *(Code)*
The code segment is a memory area

$$\mathcal{I} := MEM_{ib}^{ib+l-1}$$

for which holds: $I[k] = \langle \mathsf{OPC} \mid \cdot \rangle$ for $k = ib, \ldots, ib + l - 1$ and $I \in \mathcal{I}$. l is the number of instructions loaded into the machine and depends on the size of a particular program.

Code segment and heap directly follow each other so that $hb = ib + l$ holds.

Definition 3.8 *(Store)*
The store consists of three registers, a queue, a memory area, and an array of queues:

$$\begin{aligned} STORE := \{tb\} \times Addr \times \{true, false\} \times QUEUE \times \\ MEM_{tb}^{N} \times ARRAY_{tb}^{N}\ QUEUE \end{aligned}$$

tb is the store base register TB, *the second component is the store pointer register* TP, *the third component the locking flag* TF, *and the fourth component the* `Blocked` *queue. The last two components are the memory cells and their associated suspend queues.*

Similar to the semantics of FATOM a transition relation between two states is defined which forms the operational semantics of ATAF.

Definition 3.9 *(Transition relation)*
The operational semantics of ATAF *is defined by the relation*

$$\Longmapsto \subseteq STATE \times STATE.$$

For two states $\varsigma, \varsigma' \in STATE$ the infix notation is used:

$$\varsigma \Longmapsto \varsigma' :\Leftrightarrow \langle \varsigma, \varsigma' \rangle \in \Longmapsto$$

Using Definitions 3.4 to 3.8, a machine state ς for $noPE = m + 1$ instances is a tuple like

$$\begin{aligned} &\langle \mathtt{IC}_0, \mathtt{Run}_0, \mathtt{Ready}_0, \{\langle pid_{00}, \mathtt{IB}_{00}, \mathtt{IP}_{00}, \mathtt{HB}_{00}, \mathtt{HP}_{00}, \mathtt{SB}_{00}, \mathtt{SP}_{00}, \mathtt{MEM}_{00}, \rangle, \ldots, \\ &\langle pid_{0n_0}, \mathtt{IB}_{0n_0}, \mathtt{IP}_{0n_0}, \mathtt{HB}_{0n_0}, \mathtt{HP}_{0n_0}, \mathtt{SB}_{0n_0}, \mathtt{SP}_{0n_0}, \mathtt{MEM}_{0n_0} \rangle\}, \ldots, \\ &\mathtt{IC}_m, \mathtt{Run}_m, \mathtt{Ready}_m, \{\langle pid_{m0}, \mathtt{IB}_{m0}, \mathtt{IP}_{m0}, \mathtt{HB}_{m0}, \mathtt{HP}_{m0}, \mathtt{SB}_{m0}, \mathtt{SP}_{m0}, \mathtt{MEM}_{m0}, \rangle, \ldots, \\ &\langle pid_{mn_m}, \mathtt{IB}_{mn_m}, \mathtt{IP}_{mn_m}, \mathtt{HB}_{mn_m}, \mathtt{HP}_{mn_m}, \mathtt{SB}_{mn_m}, \mathtt{SP}_{mn_m}, \mathtt{MEM}_{mn_m} \rangle\}, \\ &\mathtt{CODE}, \langle \mathtt{TB}, \mathtt{TP}, \mathtt{TF}, \mathtt{Blocked}, \mathtt{STORE}, \mathtt{Suspend} \rangle \rangle. \end{aligned}$$

Obviously, this description is not too handy to give the semantics in a state transition system. As already mentioned at the beginning of this section, instructions are therefore defined in an imperative pseudo-code, which is a shorthand for a certain transition. The relationship between pseudo-code and transition rules is explained Section 3.3.4.
Nevertheless, writing a machine state as an element of $STATE$ is useful to define operations for process management because this representation provides a global view on the machine state.

3.3.2 Process management and scheduling

For the creation, initialisation, and finalisation of processes, three operations are necessary:

- newProc creates a new process,
- pushProc sets up the environment of a process (stack and heap), and
- terminate removes a process.

These operations are used within machine instructions, and they may work across instance boundaries.
Their meaning is defined in the following:

newProc The operation newProc returns a process identifier for the new process:

$$\mathsf{newProc} : \langle\rangle \rightarrow PID$$

Given a machine state

$$\begin{array}{rll} \varsigma = & \langle M_0, \ldots, M_m, \texttt{CODE}, S\rangle, & m = noPE - 1, \\ \text{with } M_i = & \langle \texttt{IC}_i, \texttt{Run}_i, \texttt{Ready}_i, P_i\rangle, & i = 0, \ldots, m, \\ \text{and } P_i = & \{P_{i0}, \ldots, P_{in_i}\}, & \\ \text{and } P_{ij} = & \langle pid_{ij}, \texttt{IB}_{ij}, \texttt{IP}_{ij}, \texttt{HB}_{ij}, \texttt{HP}_{ij}, \texttt{SB}_{ij}, \texttt{SP}_{ij}, \texttt{MEM}_{ij}\rangle, & j = 0, \ldots, n_i, \end{array}$$

the operation newProc creates a new process

$$P_{kl} = \langle (\!| k, l |\!), ib, ib, hb, hb, sb, sb, [\langle \mathsf{NIL}\rangle]_{hb}^{sb}\rangle$$

such that $0 \leq k \leq m$, $l \notin \{0, \ldots, n_k\}$, and $\forall i : 0 \leq i \leq m \wedge |P_i| \geq |P_k|$ holds and returns $(\!| k, l |\!)$ as result.
That means that newProc inserts a new process P_{kl} in that machine instance that has fewest processes to run, thus implementing a simple load-balancing scheme. The process has an empty memory, and all pointer registers are set to their corresponding base registers.
Of course, P_{kl} has to be inserted into the process area of instance M_k such that the whole operation result in the transition

$$\langle M_0, \ldots, M_k, \ldots, M_m, \texttt{CODE}, S\rangle \Longmapsto \langle M_0, \ldots, M_k', \ldots, M_m, \texttt{CODE}, S\rangle$$

with $M_k' = \langle \texttt{IC}_k, \texttt{Run}_k, \texttt{Ready}_k, \{P_{k0}, \ldots, P_{kn_k}, P_{kl}\}\rangle$.
The new process identifier $(\!| k, l |\!)$ is not inserted into the `Ready` queue, and the new process is not yet executed. Preparation for real work is the task of the next operation pushProc.

Note: This choice of a load balancing rule may be suboptimal: just because an instance has few processes it does not necessarily have little work. If all processes are in the `Ready` queue and will not block in the near future this instance's load might well be much higher than on an instance with more processes that do almost nothing. Again, this rule is taken for simplicity and might be replaced by more sophisticated solutions.

pushProc The signature of pushProc is

$$\mathsf{pushProc} : \mathit{PID} \times \mathit{Addr} \times \mathit{LIST\ Addr} \rightarrow \langle\rangle.$$

Given two different processes

$$P_{i_1p_2} = \langle pid_1, \mathtt{IB}_{i_1p_1}, \mathtt{IP}_{i_1p_1}, \mathtt{HB}_{i_1p_1}, \mathtt{HP}_{i_1p_1}, \mathtt{SB}_{i_1p_1}, \mathtt{SP}_{i_1p_1}, \mathtt{MEM}_{i_1p_1}\rangle$$
$$P_{i_2p_2} = \langle pid_2, \mathtt{IB}_{i_2p_2}, \mathtt{IP}_{i_2p_2}, \mathtt{HB}_{i_2p_2}, \mathtt{HP}_{i_2p_2}, \mathtt{SB}_{i_2p_2}, \mathtt{SP}_{i_2p_2}, \mathtt{MEM}_{i_2p_2}\rangle$$

with process identifiers $pid_1 = (\!|\, i_1, p_1 |\!)$ and $pid_2 = (\!|\, i_2, p_2 |\!)$, $p_1 \neq p_2$, running on instances M_{i_1} and M_{i_2} of some state $\varsigma = \langle M_0, \ldots, M_{i_1}, \ldots, M_{i_2}, \ldots, M_m, \mathtt{CODE}, S\rangle$ the operation

$$\mathsf{pushProc}\ pid_2\ a\ as$$

executed by process $P_{i_1p_1}$ copies parts of the heap of $P_{i_1p_1}$ to the heap of $P_{i_2p_2}$. The parts to be copied are specified by the addresses $as = [a_1, \ldots, a_n]$ in the last argument of pushProc:

- If an address a_i is a store address, nothing is copied but a pointer to a_i is pushed on $P_{i_2p_2}$'s stack.
- If an address a_i is a heap address its heap word is copied to the next free heap word of $P_{i_2p_2}$, and a pointer to the newly allocated word is pushed on its stack.

The next step is to copy all words reachable by the already copied words and to adjust the addresses. This is very similar to two-space garbage collection [FY69], but it is not possible to set forward pointers because the from-space (the heap of $P_{i_1p_1}$) must remain intact. Instead of forward pointers, an environment recording already copied words is used. For the copying a helper function copy is used, see Figure 3.10.

The details of the copying algorithm are best presented by a pseudo-code description shown in Figure 3.9.

Additionally, the pushProc operation also enqueues $P_{i_2p_2}$ into the `Ready` queue and sets its instruction pointer which completes the process creation by setting the first instruction $P_{i_2p_2}$ is going to execute.

The motivation for the operation pushProc is the necessity to provide a new process with the required environment, i. e. those values that it needs from its parent process, the one which called newProc, to operate properly.

terminate The terminate operation is very simple: if a process P_{kl} performs this operation it schedules the next ready process by `Run` := dequeue `Ready` for execution. As `Ready` may be empty, `Run` may contain the empty process $(\!|\cdot|\!)$. Afterwards, P_{kl} is removed from the process set of instance k.

This operation differs from newProc and pushProc as the affected process is the running one, i. e. $\mathtt{Run}_k = (\!|\, k, l\, |\!)$.

So, the resulting transition is

$$\langle M_0, \ldots, M_k, \ldots, M_m, \mathtt{CODE}, S\rangle \Longmapsto \langle M_0, \ldots, M'_k, \ldots, M_m, \mathtt{CODE}, S\rangle$$

with

$$M_k = \langle \mathtt{IC}_k, (\!|\, k, l\, |\!), \mathtt{Ready}_k, \{P_{k0}, \ldots, P_{k,l-1}, P_{kl}, P_{k,l+1} \ldots, P_{kn_k}\}\rangle$$
$$M'_k = \langle 0, \mathtt{Run}'_k, \mathtt{Ready}'_k, \{P_{k0}, \ldots, P_{k,l-1}, P_{k_l+1}, \ldots, P_{kn_k}\}\rangle.$$

$\mathtt{Run}'_k$ and $\mathtt{Ready}'_k$ are the result of the assignments (:=) in dequeue.

$\mathsf{pushProc}\ (\!|\, i_2, p_2 \,|\!)\ ip\ as \equiv$

$\quad copied := \emptyset$

$\quad$ FOREACH a IN as DO

$\quad\quad$ IF $a < \mathtt{TB}$ THEN

$\quad\quad\quad copied := \mathsf{copy}\ [a]\ copied$

$\quad\quad\quad \mathtt{STACK}_{i_2 p_2}[\mathtt{SP}_{i_2 p_2}] := \langle \mathsf{PTR} \mid copied(a) \rangle$

$\quad\quad$ ELSE

$\quad\quad\quad copied := copied.[a \mapsto a]$

$\quad\quad\quad \mathtt{STACK}_{i_2 p_2}[\mathtt{SP}_{i_2 p_2}] := \langle \mathsf{PTR} \mid a \rangle$

$\quad\quad$ END IF

$\quad\quad \mathsf{inc}\ \mathtt{SP}_{i_2 p_2}$

$\quad$ END FOREACH

$\quad ptr := \mathtt{HB}_{i_2 p_2}$

$\quad$ WHILE $ptr < \mathtt{HP}_{i_2 p_2}$ DO

$\quad\quad$ IF $\mathtt{HEAP}_{i_2 p_2}[ptr] = \langle \mathsf{PTR} \mid a \rangle$ THEN

$\quad\quad\quad copied := \mathsf{copy}\ [a]\ copied$

$\quad\quad\quad \mathtt{HEAP}_{i_2 p_2}[ptr] := \langle \mathsf{PTR} \mid copied(a) \rangle$

$\quad\quad$ ELSE IF $\mathtt{HEAP}_{i_2 p_2}[ptr] = \langle \mathsf{APP} \mid a_1\ a_2 \rangle$ THEN

$\quad\quad\quad copied := \mathsf{copy}\ [a_1, a_2]\ copied$

$\quad\quad\quad \mathtt{HEAP}_{i_2 p_2}[ptr] := \langle \mathsf{APP} \mid copied(a_1)\ copied(a_2) \rangle$

$\quad\quad$ ELSE IF $\mathtt{HEAP}_{i_2 p_2}[ptr] = \langle \mathsf{PCK} \mid t\ [a_1, \ldots, a_n] \rangle$ THEN

$\quad\quad\quad copied := \mathsf{copy}\ [a_1, \ldots, a_n]\ copied$

$\quad\quad\quad \mathtt{HEAP}_{i_2 p_2}[ptr] := \langle \mathsf{PTR} \mid t\ [copied(a_1), \ldots, copied(a_n)] \rangle$

$\quad\quad$ END IF

$\quad\quad \mathsf{inc}\ ptr$

$\quad$ END WHILE

$\quad \mathsf{enqueue}\ (\!|\, i_2, p_2 \,|\!)\ \mathtt{Ready}_{i_2}$

$\quad \mathtt{IP}_{i_2 p_2} := ip$

Figure 3.9 Definition of the pushProc operation.

$\mathsf{copy}\ as\ copied \equiv$
FOREACH a IN as DO
IF $copied(a) = undefined$ THEN
$\mathtt{HEAP}_{i_2p_2}[\mathtt{HP}_{i_2p_2}] := \mathtt{HEAP}_{k_1l_1}[a]$
$copied := copied.[a \mapsto \mathtt{HP}_{i_2p_2}]$
$\mathsf{inc}\ \mathtt{HP}_{i_2p_2}$
END IF
END FOREACH
RETURN $copied$

Figure 3.10 Helper function copy for pushProc.

Scheduling

The scheduler of ATAF is a simple round robin scheduler. Round robin means that each process may execute a certain number NI of instructions until it is another process' turn. The scheduler is activated each time a process modifies its instruction pointer IP, which indicates completion of an instruction. If the instruction counter of the process' instance IC exceeds NI the process is enqueued in Ready, and the next process is set to run with a reset instruction counter. There are two exceptions:

- If a process is uninterruptible it may run any time until it sets itself to interruptible again.
- If the Ready queue is empty the currently running process may continue with a reset instruction counter.

To integrate the scheduler in the instructions the operation set IP is provided, which does the required checks and scheduling operations (Figure 3.11). To use the inc operation with IP, inc IP is declared as an abbreviation for set IP $(\mathtt{IP}+1)$. To handle the state where Run is the empty process but one or more processes have been enqueued in Ready in the meantime, e. g. processes returning from a Suspend or the Blocked queue, one more transition is required:

$$\langle M_0, \ldots, M_k, \ldots, M_m, \mathtt{CODE}, S\rangle \Longmapsto \langle M_0, \ldots, M_k', \ldots, M_m, \mathtt{CODE}, S\rangle$$

with $M_k = \langle \mathtt{IC}_k, (\!|\cdot|\!), pid : pids, P_k\rangle$ and $M_k' = \langle \mathtt{IC}_k, pid, pids, P_k\rangle$.
That means the empty process can be replaced by a real one to continue execution of the program.

Note: This scheduling scheme is very simple and may even be unfair in a sense that a process gets more processing time than another one despite executing the same number of instructions: mere instruction counting neglects the difference in execution time of instructions.

Process state transitions

To summarise the different states introduced in the previous sections Figure 3.12 shows a state transition diagram.

set IP $a \equiv$
    IP $:= a$
    IF Run $\neq \langle\!| i, p |\!\rangle \ \wedge$ IC $\geq NI$ THEN
        $next :=$ dequeue Ready
        IF $next \neq (\!|\cdot|\!)$ THEN
            enqueue Run Ready
            Run $:= next$
        END IF
        IC $:= 0$
    ELSE
        inc IC
    END IF

Figure 3.11 Operation set IP to modify the instruction pointer including scheduling.

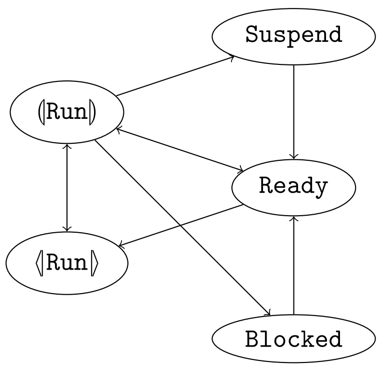


Figure 3.12 Process state transition diagram.

In this diagram a node

q

means that a process is waiting in queue q.
The Run nodes are special as they are not queues. (|Run|) means a process is running in an interruptible state, whereas ⟨|Run|⟩ means a process is running in an uninterruptible state.
The transitions shown are not all transitions possible in principle, but only those a correctly compiled FATOM program performs. Especially uninterruptible execution is only entered from the Ready queue or the Run register but not from any of the store queues.

3.3.3 Initial and final state

The initial state of ATAF is defined by certain constants chosen on startup of the machine:

- the sequence of instructions, i. e. the program, to execute and its length l,
- the number of processing elements $noPE$,
- the values for the base registers ib, hb, sb, and
- the last address of the store N.

The value for hb is determined by ib and l, sb sets the size for the stacks and heaps, N defines together with sb the store size.
The *initial state* ς_0 of ATAF is a certain number of instances, $noPE = m + 1$, which have empty process sets except for the first one, with the program loaded in to CODE segment:

$$\varsigma_0 = \langle M_0, \ldots, M_m, \texttt{CODE}, S\rangle$$

with

$$\begin{aligned} M_0 &= \langle 0, (\!|\, 0, 0 \,|\!), [], \{P_{00}\}\rangle, \\ M_i &= \langle 0, (\!|\cdot|\!), [], \emptyset\rangle, \quad i > 0, \\ S &= \langle sb + 1, sb + 1, \mathit{true}, [], [\langle \mathsf{NIL}\rangle]^N_{sb+1}, [[]]^N_{sb+1}\rangle, \\ P_{00} &= \langle (\!|\, 0, 0 \,|\!), ib, ib, l + 1, l + 1, sb, sb, [\langle \mathsf{NIL}\rangle]^{sb}_{l+1}\rangle. \end{aligned}$$

The only process that exists in the initial state is scheduled for execution: it occupies the `Run` register and starts execution with the instruction at memory address ib. All processes have $sb - l$ words of local memory.
The constants N and sb should be chosen large enough for the program to terminate without memory shortage.
The *final state* is a state without any processes, i. e. for all M_i, $i = 0, \ldots, m$, holds $M_i = \langle ic, (\!|\cdot|\!), [], \emptyset\rangle$ for any instruction counter ic.
There is a second final state, called a *failure state*, representing a failed computation. It is denoted by $\lightning$, and there is no further transition possible. This state is entered by the fail operation.

3.3.4 Instructions

To relate the pseudo-code with the transition relation, two things have to be clarified:

- the dispatching of instructions and
- the effect of assignments.

Each process has an instruction pointer IP, and the instruction stored at CODE[IP] of a running process determines the next state for its instance. All instances do a state transition simultaneously. Therefore, the general layout of a transition is

$$\langle M_0, \ldots, M_m, \texttt{CODE}, S\rangle \Longmapsto \langle M'_0, \ldots, M'_m, \texttt{CODE}, S'\rangle.$$

The M'_i evolve out of the M_i in the following way:

$$M_i = \langle ic, (\!|\, i, p_i \,|\!), \texttt{Ready}_i, P_i\rangle \;\Rightarrow\; M'_i = \langle ic', (\!|\, i, p'_i \,|\!), \texttt{Ready}'_i, P'_i\rangle$$

The M_i' are the result of execution of the instruction $\mathtt{CODE}[\mathtt{IP}_{ip_i}]$ by processes $(\!| i, p_i |\!)$, $i = 0, \ldots, m$. One process at a time may also modify the store, i. e. there may be one instruction $\mathtt{CODE}[\mathtt{IP}_{i^* p_i^*}]$ that modifies the store resulting in a change from S to S'.
To ensure that at most one process is working on the store, LOCK and UNLOCK instructions have to be used correctly (which is the responsibility of the compiler, cf. Section 3.4), but they also have to be atomic. The atomicity is forced in the transition relation: let $\mathtt{CODE}[\mathtt{IP}_{kl_k}]$ be UNLOCK or LOCK for $k \in I$, $I \subseteq \{0, \ldots, m\}$, and $\mathtt{Run}_k = (\!| k, l_k |\!)$. For a transition

$$\langle M_0, \ldots, M_m, \mathtt{CODE}, S\rangle \Longmapsto \langle M_0', \ldots, M_m', \mathtt{CODE}, S'\rangle,$$

it must hold that only one of the M_k', $k \in I$, is different from M_k.
The changes from M_i to M_i' or S to S' are caused by assignments written as := in the pseudo-code. An assignment *lhs* := *rhs* causes the component *lhs* of the previous state to be replaced by *rhs* in the next state.
To reduce clutter, process local components like registers or memory are used without process indices in the pseudo-code and have to be interpreted relatively to the current content of the `Run` register. Furthermore, HEAP is used synonymously for MEM in heap operations and analogously STACK in stack operations.

Overview of ATAF's instruction set

Figure 3.13 shows the set *Instr* of ATAF machine instructions with references to their description and definition. As certain subsets of instructions are very similar to each other, some of them are not printed in this section but only in Appendix B, which lists all instructions in alphabetical order.

Instructions for constraints

SPAWN The instruction SPAWN implements the (*conj*) transition rule of FATOM. SPAWN creates a new process and copies the data referenced by the last n pointers of the parent's stack to the new process environment. The new process is scheduled for execution starting with the IP-relative address a.

ISPACK, ISBOUND, ISUNBOUND The three instructions ISPACK, ISBOUND, and ISUNBOUND check primitive constraints; ISPACK checks a *pack*, ISBOUND a *bound*, and ISUNBOUND an *unbound* constraint. They work like conditional jumps: if the constraint is fulfilled execution proceeds with the next instructions, otherwise an IP-relative jump to the instruction at address a is performed. The first argument n indicates the stack-frame which points to the variable in question.
These instruction not only work on words in the store but also on local words. This is not strictly necessary but simplifies code generation (cf. Section 3.4.1) a lot.
Instruction ISPACK with its support operation is shown in Figure 3.15, ISBOUND and ISUNBOUND work analogously and are found on page 110 in Appendix B.

ALLOC This instruction reserves a number of words from the store, thus implementing the **with** construct from FATOM. Pointers to the variables are left on the stack.

Instr	$\rightarrow$	SPAWN *Num Addr*	*(Page 67, Figure 3.14)*
	|	ISPACK *Nat Nat Nat Addr*	*(Page 67, Figure 3.15)*
	|	ISBOUND *Nat Addr*	
	|	ISUNBOUND *Nat Addr*	
	|	ALLOC *Nat*	*(Page 67, Figure 3.16)*
	|	SUSPEND Nat^{+}	*(Page 68, Figure 3.17)*
	|	TELL *Addr*	*(Page 70, Figure 3.18)*
	|	LOCK | UNLOCK	*(Page 56, Figure 3.6)*
	|	PUSHFUN *Addr Nat* | PACK *Nat Nat*	*(Page 70, Figure 3.19)*
	|	PUSHVAR *Nat* | PUSHNAT *Nat*	
	|	PUSHFLOAT *Float*	
	|	MKAP | UPDATE *Nat*	*(Page 72, Figure 3.20)*
	|	POP *Nat* | SLIDE *Nat*	*(Page 72, Figure 3.21)*
	|	CASEJUMP $(Nat\ Addr)^{+}$	*(Page 72, Figure 3.23)*
	|	SPLIT *Nat* | JUMP *Addr*	
	|	UNWIND	*(Page 75, Figure 3.24)*
	|	EVAL	*(Page 75, Figure 3.25)*
	|	ADD | SUB | MUL | DIV | NEG	*(Page 75, Figure 3.26)*
	|	LT | LE | EQ | NE | GE | GT	
	|	AND | OR | NOT	
	|	TONAT | TOFLOAT	
	|	ERROR	*(Page 76, Figure 3.27)*
	|	NOPE	*(Page 77, Figure 3.28)*

Figure 3.13 ATAF's instruction set *Instr*.

SUSPEND If a process finds none of its guards fulfilled or an argument to a function unbound it sleeps until any of these variables changes their state. This is accomplished by the SUSPEND instruction, which enqueues the process in the SUSPEND queues of the given stack words $n_1, \ldots, n_m$.
SUSPEND includes unlocking the store with operation unlock (see Figure 3.6). The reason is that suspending on variables has to take place under mutual exclusion but neither of the sequences

- UNLOCK, SUSPEND $n_1 \ldots n_m$ and
- SUSPEND $n_1 \ldots n_m$, UNLOCK

is able to guarantee proper unlocking and suspension under all circumstances: in the first case, the process may be interrupted after UNLOCK and later suspend on variables which may have changed during the interruption and in consequence never wake up again. In the second case, the process will never unlock the store because it passes control to another process in SUSPEND.
In addition to these complications, it is also not possible to use inc IP because this does not necessarily schedule the next process. As shown in the full definition

SPAWN $n\ a \equiv$
    $pid :=$ newProc
    pushProc pid $(\texttt{IP} + a)$ $[\texttt{STACK}[\texttt{SP}+1], \ldots, \texttt{STACK}[\texttt{SP}+n]]$
    inc IP

Figure 3.14 Instruction SPAWN – create a new process.

ISPACK $n\ t\ ar\ a \equiv$
    IF lookup $n = \langle \text{PCK} \mid t\ as \rangle \wedge$
        $|as| = ar$ THEN
        inc IP
    ELSE
        set IP $(\texttt{IP} + a)$
    END IF

lookup $n \equiv$
    $\langle \text{PTR} \mid ptr \rangle :=$ nth n
    IF $ptr <$ TB THEN
        RETURN $\texttt{HEAP}[ptr]$
    ELSE
        RETURN $\texttt{STORE}[ptr]$
    END IF

Figure 3.15 Instruction ISPACK and its helper operation lookup – example for primitive constraints.

ALLOC $n \equiv$
    $as :=$ allocateVars n
    FOREACH a IN as DO
        push $\langle \text{PTR} \mid a \rangle$
    END FOREACH
    inc IP

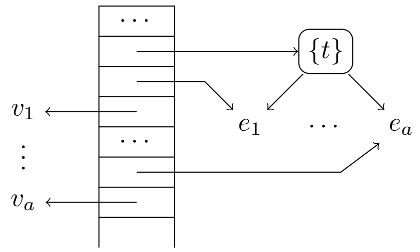
Figure 3.16 Instruction ALLOC – reserve words in the store.

SUSPEND $n_1 \ldots n_m \equiv$
    unlock
    FOREACH n IN $[n_1, \ldots, n_m]$ DO
        $\langle \text{PTR} \mid a \rangle :=$ nth n
        enqueue Run Suspend$[a]$
    END FOREACH
    $\texttt{IP} := \texttt{IP} + 1$
    Run := dequeue Ready

Figure 3.17 Instruction SUSPEND – wait for an event.

of SUSPEND in Figure 3.17, this scheduling is done explicitly.

TELL The transition rules (*tellV*) and (*tellP*) are implemented by this instruction. To distinguish between the two rules, the top value on the stack, which is the result of the last computation, is examined. If it is a number, (*tellV*) is applied, i. e. the number is written to the store.
If a constructor is on top of the stack, it is inserted into the store as required by (*tellP*) and variables for its sub-terms are allocated. These variables are paired with pointers to the sub-terms and pushed onto the stack. For a constructor $Pack\{t, a\}\ e_1 \ldots e_a$ the relevant parts of the stack and heap look as follows:



As the store is not shown in the drawing, $v_1, \ldots, v_a$ indicate the allocated words in the store. It is important that the top value of the stack, which is propagated to the store, is *not* popped off the stack. This way it is possible for subsequent instructions to decide if sub-terms have to be taken care of according to (*tellP*) (see Section 3.4.1 on code generation).
After writing the value to the store, all processes in the Suspend queue waiting for that variable have to be awoken. These processes might also wait for other variables and must be removed from those queues, too. This is done by wakeup, defined in Appendix B on page 121.
The machine fails if TELL is requested to bind a store variable which is already bound to a different value.

Instructions for graph reduction

The instructions of this section form the G-machine part of ATAF. They perform graph reduction as described in Section 3.1.
The idea behind the G-machine is to have a sequence of instructions that when executed builds and instantiates the graph of the body of a supercombinator. Apart from these graph building instructions there are instructions to change the flow of control.

Graph building: PUSHFUN, PACK, PUSHVAR, PUSHNAT, PUSHFLOAT The instructions PUSHFUN, PACK create constant nodes in the heap and leave a pointer to this node on the stack. PUSHFUN creates a $\langle$FUN $\mid a\ ar\rangle$ node that references to the code of a ar-ary function starting at address a in the CODE segment.
PACK creates a $\langle$PCK $\mid t\ as\rangle$ node in the heap. The addresses as of the sub-terms are the first $|as|$ terms of the stack, which are removed and replaced by a pointer to the constructor node.
PUSHVAR differs slightly as it does not push some kind of constant on the stack but a pointer to an argument of a supercombinator. As shown in Section 3.1.3,

```
TELL n ≡
    ⟨PTR | a⟩ := nth n
    ⟨PTR | ptr⟩ := top
    w := lookup ptr
    IF STORE[a] = ⟨NIL⟩ THEN
        IF w = ⟨NAT | ·⟩ ∨ w = ⟨FLT | ·⟩ THEN STORE[a] := w
        ELSE
            ⟨PCK | t as⟩ := w
            vs := allocateVars |as|
            STORE[a] := ⟨PCK | t vs⟩
            FOREACH (a′, v′) IN (as, vs) DO
                push ⟨PTR | a′⟩
                push ⟨PTR | v′⟩
            END FOREACH
        END IF
        wakeup a
    ELSE IF
        STORE[a] ≠ w THEN fail
    END IF
    inc IP
```

Figure 3.18 Instruction TELL – augment a binding to the store.

```
PUSHFUN a ar ≡
    a′ := allocate
    HEAP[a′] := ⟨FUN | a ar⟩
    push ⟨PTR | a′⟩
    inc IP

PUSHVAR n ≡
    push (nth n)
    inc IP

PACK t ar ≡
    a := allocate
    as := []
    FOR; i := 1 TO ar DO
        ⟨PTR | a′⟩ := pop
        as := as ++ [a′]
    END FOR
    HEAP[a] := ⟨PCK | t as⟩
    push ⟨PTR | a⟩
    inc IP
```

Figure 3.19 Instructions PUSHFUN, PACK, and PUSHVAR – graph building.

unwinding the spine results in pointers to a function's arguments on the stack. PUSHVAR creates a pointer to the argument in the n^{th} stack word.
The pseudo-code for PUSHFUN, PACK, and PUSHVAR is shown in Figure 3.19. Two more very similar instructions to push natural and floating-point numbers on the stack, PUSHFLOAT and PUSHNAT can be found in Appendix B on page 112 and page 113.
The instructions PUSHNAT and PUSHFLOAT allocate a $\langle$NAT$\rangle$ or a $\langle$FLT$\rangle$ node respectively and leave a pointer to this node on the stack.

Graph building: MKAP, UPDATE The operation MKAP creates an application node that points to the two graphs referenced by the two topmost stack words. UPDATE implements updateing of the root of a redex after an instantiation is finished (cf. Section 3.1.4).

Stack handling: POP, SLIDE The instruction POP throws away the the first n words of the stack. SLIDE is similar but retains the topmost element.

Control flow: JUMP, CASEJUMP, SPLIT A case analysis is implemented by the two instructions CASEJUMP and SPLIT. CASEJUMP examines the tag of the constructor on top of the stack and jumps to the address of this tag's branch.
The binding of a constructor's sub-terms to fresh local variables, i. e. pointers on the stack, is done by a SPLIT instruction.
The unconditional, IP-relative jump is performed by a JUMP instruction; see Figure 3.23 for all three instructions.
There is one complication: the value in examination by a CASEJUMP or a SPLIT may be an unbound value in the store. In that case, the process has to suspend on this variable. The check if one (or more) variables are bound is done by the operation suspend (Figure 3.22), which either enqueues the process in one or more Suspend queues or returns the values of the variables. For CASEJUMP and SPLIT only one variable is examined, but suspend is also used in arithmetic and logic operations that require more variables (see below). The instruction pointer of the running process is not modified before it is enqueued in any of the Suspend queues. This means that the same CASEJUMP or SPLIT instruction is completely retried after being woken up.

MKAP $\equiv$	UPDATE $n \equiv$
$a :=$ allocate	$\langle$PTR $\mid a\rangle :=$ nth n
$\langle$PTR $\mid a_1\rangle :=$ pop	HEAP$[a] :=$ pop
$\langle$PTR $\mid a_2\rangle :=$ pop	inc IP
HEAP$[a] := \langle$APP $\mid a_1\ a_2\rangle$	
push $\langle$PTR $\mid a\rangle$	
inc IP	

Figure 3.20 Instructions MKAP and UPDATE – graph building.

```
POP n ≡                  SLIDE n ≡
    SP := SP + n             STACK[SP + n + 1] := top
    inc IP                   SP := SP + n
                             inc IP
```

Figure 3.21 Instructions POP, SLIDE – stack handling.

```
suspend as ≡
    unbound := false
    ws := []
    FOR a IN as DO
        w := lookup a
        ws := ws ++ [w]
        IF w = ⟨NIL⟩ THEN
            unbound := true
            enqueue Run Suspend[a]
        END IF
    END FOR
    IF unbound THEN
        Run := dequeue Ready
    ELSE
        RETURN ws
    END IF
```

Figure 3.22 Operation suspend.

```
JUMP a ≡ set IP (IP + a)

CASEJUMP t1 a1 ... tn an ≡       SPLIT n ≡
    ⟨PTR | a⟩ := top                ⟨PTR | a⟩ := top
    [w] := suspend [a]              [w] := suspend [a]
    ⟨PCK | t ·⟩ := w                pop
    FOR i := 1 TO n DO              ⟨PCK | · [a1, ..., an]⟩ := w
        IF t = ti THEN              FOREACH a' IN [an, ..., a1] DO
            JUMP ai                     push ⟨PTR | a'⟩
        END IF                      END FOREACH
    END FOR                         inc IP
    fail
```

Figure 3.23 Instructions JUMP, CASEJUMP, SPLIT – control flow.

```
UNWIND ≡
    IF SP = SB THEN
        terminate
    END IF
    ⟨PTR | a⟩ := top
    IF a < TB THEN
        w := HEAP[a]
        IF w = ⟨FUN | a′ ar⟩ THEN
            rearrange ar
            JUMP (a − IP)
        ELSE IF ground w THEN
            undump
        ELSE IF w = ⟨APP | a1 ·⟩ THEN
            push ⟨PTR | a1⟩
            UNWIND
        ELSE IF w = ⟨PTR | ·⟩ THEN
            pop
            push w
            UNWIND
        END IF
    ELSE
        undump
    END IF

undump ≡
    v := pop
    ⟨PTR | ip⟩ := pop
    push v
    set IP (ip + 1)

rearrange ar ≡
    FOR i := 1 TO ar DO
        w := HEAP[SP + i + 1]
        ⟨APP | · a2⟩ := w
        STACK[SP + i] := a2
    END FOR

ground v ≡
    RETURN v = ⟨PCK | · ·⟩ ∨
        v = ⟨NAT | ·⟩ ∨
        v = ⟨FLT | ·⟩
```

Figure 3.24 Instruction UNWIND with its helper operations – unwinding the spine.

Control flow: UNWIND Unwinding the spine is done by UNWIND. It is the most complex control instruction. As long as the stack's top points to an application node the spine is further unwound. As soon as a supercombinator is reached, i. e. the next redex is found, control is passed to the supercombinator's code to build and instantiate the body. If the stack is empty the process terminates as it has finished its work.
If there is a number or constructor on the stack this is a result of a computation in a strict context, e. g. application of a built-in functions or a tell equality, and the instruction pointer of the previous computation has to be undumped from the stack.
UNWIND requires two helper operations: rearrange and ground, which are also shown in Figure 3.24. rearrange modifies the pointers to an unwound spine to point to the argument rather than the application nodes, and ground checks if a word is a constant (a constructor or a number).

Strict evaluation: EVAL The instruction EVAL forces the evaluation of an expression. It is e. g. necessary to evaluate arguments to built-in functions before the functions itself is evaluated (see Section 3.1.5). To resume the computation of the function after evaluation of an argument the current instruction pointer is dumped on the stack. A pointer to the graph to be reduced is pushed on the stack and its spine is unwound.
Undumping the instruction pointer takes place in the UNWIND instructions (see previous paragraph).

```
EVAL ≡
    ptr := pop
    push ⟨PTR | IP⟩
    push ptr
    UNWIND
```

Figure 3.25 Instruction EVAL – imposing strict context.

Instructions for built-in functions

As FATOM has a set of built-in functions for arithmetics, comparisons etc., ATAF provides instructions for these operations.
As built-in functions are strict in all arguments, suspend is used to make sure that variables are bound like in CASEJUMP and SPLIT.

Arithmetics: ADD, SUB, MUL, DIV, NEG The arithmetics instructions ADD, SUB, MUL, and DIV remove the first two words from the stack and replace them with the result of the arithmetic operations $+$, $-$, $*$, $/$ respectively.
NEG is a unary instruction and replaces the number on top of the stack by its complement.

Comparisons: LT, LE, EQ, NE, GE, GT The comparison instructions LT, LE, EQ, NE, GE, and GT compare two numbers on top of the stack and replace them by the result of the comparison with $<$, $\leq$, $=$, $\neq$, $\geq$, $>$ respectively. A constructor $\langle \mathsf{PCK} \mid 1\ [] \rangle$ means *true*, $\langle \mathsf{PCK} \mid 2\ [] \rangle$ represents *false*.

Boolean instructions: AND, OR, NOT The boolean instructions AND, OR, and NOT work on the constructor representation of *true* and *false*. The first two instructions replace the two topmost words on the stack by the result of the connective, NOT is unary and replaces only the top word.

Conversion instructions: TONAT, TOFLOAT The conversion instructions convert natural numbers to floats (TOFLOAT) and vice versa (TONAT).

As all these instructions do not differ much from each other, only ADD is shown in Figure 3.26, all other instructions are listed in Appendix B. Like in CASEJUMP or SPLIT, the machine has to make sure that the arguments on the stack are bound.

```
ADD ≡
    arith +
    inc IP
arith ⊕ ≡
    a := allocate
    a1 := nth 0
    a2 := nth 1
    [v1, v2] := suspend [a1, a2]
    pop
    pop
    IF v1 = ⟨NAT | n1⟩ ∧ v2 = ⟨NAT | n2⟩ THEN
        HEAP[a] := ⟨NAT | n1 ⊕ n2⟩
    ELSE
        ⟨· | x1⟩ := v1
        ⟨· | x2⟩ := v2
        HEAP[a] := ⟨FLT | x1 ⊕ x2⟩
    END IF
    push ⟨PTR | a⟩
```

Figure 3.26 Instruction ADD – example for a binary arithmetic instruction.

Stopping execution: ERROR The instruction ERROR stops the entire machine with an error.

$$\mathsf{ERROR} \equiv \mathsf{fail}$$

Figure 3.27 Instruction ERROR – halting the machine.

Number of processing elements: NOPE The number of processing elements is put on the stack by NOPE.

$$
\begin{array}{l}
\mathsf{NOPE} \equiv \\
\quad a := \mathsf{allocate} \\
\quad \mathtt{HEAP}[a] := \langle \mathsf{NAT} \mid noPE \rangle \\
\quad \mathsf{push}\ \langle \mathsf{PTR} \mid a \rangle \\
\quad \mathsf{inc}\ \mathtt{IP}
\end{array}
$$

Figure 3.28 Instruction NOPE – number of processing elements.

3.4 Compiling FATOM to ATAF

After the description of the instruction set of the ATAF abstract machine, a compiler to translate FATOM programs into a list of ATAF instructions is necessary. This section defines a series of compilation schemes for this purpose. A compilation scheme (or translation function) is a function that takes a FATOM program or a part of a FATOM program and maps it to a list of ATAF instructions. These functions are defined recursively along the syntax of FATOM. Unfortunately, compilation requires slightly more information than the plain abstract syntax is ready to provide:
For a supercombinator definition

$$\mathbf{def}\ v\, v_1 \ldots v_n = e,$$

the compiler, i. e. the translation function, needs to know if v is a functional or constraint abstraction.
Therefore, the compilation schemes are defined along an annotated abstract syntax. Annotations are appended to a node in the abstract syntax tree by a colon as in Section 2.3.
The annotations $\{\mathsf{C}, \mathsf{F}\}$ are appended to a supercombinator definition with C for constraint abstractions and F for functional abstractions, e. g.

$$[\![(\mathbf{def}\ \mathit{farm}\ f\ l\ r = e) : \mathsf{C}]\!].$$

This annotation tells the compilation function that *farm* is a constraint abstraction.
As far as this section is concerned, annotations are assumed to be available. Nevertheless, Appendix C shows an inference algorithm for the supercombinator types of a given program for use in a compiler implementation.
Related to the variable annotations are compilation environments:

Definition 3.10 *(Compilation environment)*
A compilation environment Γ *is an environment mapping variable names to numbers.*

$$\Gamma \in ENV^{\mathbb{N}}_{Var}$$

These environments are necessary to keep track of which variable is stored in which stack word. According to Definition 1.4, transformation of an environment's value domain is written Γ^f. This section uses a function $+n(m) = n+m$ to shift the values in a compilation environment.

Before the presentation of the translation functions one additional notational convention has to be introduced. The translation functions return a sequence of ATAF machine instructions. Some of these instructions contain relative addresses. These can only be calculated after the entire sequence is known, i. e. it is easier to calculate them after recursively traversing the abstract syntax. To indicate which address refers to which instruction the following notation is used:

- For a sequence of instructions $[i_1, \ldots, i_n]$, an index a pre- or appended to the sequence refers to the first or last instruction. For example, in ${}_a[i_1, \ldots, i_n]$, a refers to i_1, and in $[i_1, \ldots, i_n]_a$ it refers to i_n.

- If a sequence is itself the result of a function $f(x)$ then a in ${}_a\big|f(x)$ refers to the address of the first instruction and in $f(x)\big|_a$ to the address of the last one.

- These subscript are relative addresses. The following example illustrates their calculation. Given the sequence of instructions

$$_{a_2}[\mathsf{PUSHNAT}\ 2, \mathsf{JUMP}\ a_1] +\!\!+ [\mathsf{SPAWN}\ 2\ a_2] +\!\!+ {}_{a_1}\big|f(x)$$

  with $f(x) = [\mathsf{MKAP}, \mathsf{MKAP}]$ the calculation of the addresses a_1 and a_2 yields

$$[\mathsf{PUSHNAT}\ 2, \mathsf{JUMP}\ 2, \mathsf{SPAWN}\ 2\ \ -2, \mathsf{MKAP}, \mathsf{MKAP}].$$

The next two sections basically define two compilation schemes: $\mathcal{C}$ for constraint abstractions and $\mathcal{F}$ for functional abstractions. $\mathcal{F}$ consists of two parts: $\mathcal{F}^{\mathsf{L}}$ for compilation in a lazy context and $\mathcal{F}^{\mathsf{S}}$ for compilation in a strict context. This distinction is made because more efficient code can be generated for expressions in strict context.

Strict context requires an expression to be evaluated to normal form. To indicate that an expression e is in strict context $\mathsf{S}[e]$ is written. The set of expressions in strict context is inductively defined:

Definition 3.11 *(Expressions in strict context)*
The set of expressions in strict context is defined as:

$$\begin{aligned}
\textbf{def } v\, v_1 \ldots v_n = e &\Rightarrow \mathsf{S}[e] \\
\mathsf{S}\begin{bmatrix} \textbf{case } e \textbf{ of} \\ \quad \{t_1\}\, v_{11} \ldots v_{1n_1} \to e_1\,; \\ \quad \vdots \\ \quad \{t_m\}\, v_{m1} \ldots v_{mn_m} \to e_m \end{bmatrix} &\Rightarrow \mathsf{S}[e], \mathsf{S}[e_1], \ldots, \mathsf{S}[e_m] \\
\mathsf{S}\begin{bmatrix} \textbf{let } v_1 = e_1\,; \ldots ; v_n = e_n \\ \textbf{in } e \end{bmatrix} &\Rightarrow \mathsf{S}[e] \\
\mathsf{S}[e_1 \oplus e_1] &\Rightarrow \mathsf{S}[e_1], \mathsf{S}[e_2] \\
\mathsf{S}[b\, e] &\Rightarrow \mathsf{S}[e] \quad \textit{for } b \in \textit{Builtin} \\
\mathsf{S}[v =:= e] &\Rightarrow \mathsf{S}[e] \\
\mathsf{S}[e_1 \,\&\, \ldots \,\&\, e_n] &\Rightarrow \mathsf{S}[e_1], \ldots, \mathsf{S}[e_n]
\end{aligned}$$

All other expressions are in a lazy context. Especially application $e_1\, e_2$ of non-built-in functions and local definitions in **let**-expressions are lazy.

3.4.1 Compiling constraint abstractions

A constraint abstraction v is compiled by compiling its body in an environment containing all parameters. As shown in Section 3.1.4, the arguments to a supercombinator are on the stack when its body is instantiated. They have to be removed (POP) before unwinding continues.

$$\begin{aligned}
&\mathcal{C}[\![(\textbf{def } v\, v_1 \ldots v_n = e) : \mathbf{C}]\!] = \\
&\qquad \mathcal{C}[\![e]\!]\, [v_1 \mapsto 0, \ldots, v_n \mapsto n-1] +\!\!+ [\mathsf{POP}\ n+1, \mathsf{UNWIND}]
\end{aligned} \tag{3.1}$$

The body of a constraint abstraction can be a guarded expression, a **with**-expression, a conjunction of atoms, or a single atom.

Guarded expressions

The compilation of a guarded expression is a rather complicated:

$$\begin{aligned}
&\mathcal{C}\left[\!\!\left[\begin{matrix} c_{11}[v_{11}] \,\&\, \ldots \,\&\, c_{1n_1}[v_{1n_1}] \Rightarrow e_1 \\ | \cdots | \\ c_{m1}[v_{m1}] \,\&\, \ldots \,\&\, c_{mn_m}[v_{mn_m}] \Rightarrow e_m \end{matrix}\right]\!\!\right] \Gamma = \\
&\qquad [\mathsf{LOCK}]_b +\!\!+ \mathcal{C}_{a_1}[\![c_{11}[v_{11}] \,\&\, \ldots \,\&\, c_{1n_1}[v_{1n_1}]]\!]\,\Gamma +\!\!+ \\
&\qquad [\mathsf{UNLOCK}] +\!\!+ \mathcal{C}[\![e_1]\!]\,\Gamma +\!\!+ [\mathsf{JUMP}\ e+1]_{a_1} +\!\!+ \\
&\qquad \cdots +\!\!+ \\
&\qquad \mathcal{C}_{a_m}[\![c_{m1}[v_{m1}] \,\&\, \ldots \,\&\, c_{mn_m}[v_{mn_m}]]\!]\,\Gamma +\!\!+ \\
&\qquad [\mathsf{UNLOCK}] +\!\!+ \mathcal{C}[\![e_m]\!]\,\Gamma +\!\!+ [\mathsf{JUMP}\ e+1] +\!\!+ \\
&\qquad [\mathsf{SUSPEND}\ \Gamma(v_1), \ldots, \Gamma(v_l)]_{a_m} +\!\!+ [\mathsf{JUMP}\ b]_e \\
&\text{with } v_k \in \{v_{ij} \mid i = 1, \ldots, m, j = 1, \ldots, n_i\}, v_{k_1} \neq v_{k_2} \text{ for } k_1 \neq k_2
\end{aligned} \tag{3.2}$$

The notation $c_{ij}[v_{ij}]$ means a primitive constraint c_{ij} on variable v_{ij}, e. g. *bound buf*, which is a primitive constraint on the variable *buf*.

The guards have to be evaluated with exclusive access to the store, hence the enclosing LOCK, UNLOCK instructions. As the primitive constraints are compiled to conditional jumps (see below), the address for these jumps is passed to their compilation function. If any of the constraints in a guard fails the guarded expression e_i is skipped, and the next guard is checked. If none of the guards is true the process suspends on all constrained variables. When it is awoken it jumps back to the code of the first guard. If a guard is fulfilled and its body has been executed all other guarded expressions are skipped by a jump over all remaining instructions ($\mathsf{JUMP}\ e+1$).

A guard is compiled into a sequence of conditional jumps:

$$\mathcal{C}_a[\![c_1 \,\&\, \ldots \,\&\, c_n]\!]\,\Gamma = \mathcal{C}_a[\![c_1]\!]\,\Gamma +\!\!+ \cdots +\!\!+ \mathcal{C}_a[\![c_n]\!]\,\Gamma \tag{3.3}$$

$$\mathcal{C}_a[\![\mathtt{pack}\ v\ t\ ar]\!]\,\Gamma = [\mathsf{ISPACK}\ k\ t\ ar\ a] \tag{3.4}$$

$$\mathcal{C}_a[\![\mathtt{bound}\ v]\!]\,\Gamma = [\mathsf{ISBOUND}\ k\ a] \tag{3.5}$$

$$\mathcal{C}_a[\![\mathtt{unbound}\ v]\!]\,\Gamma = [\mathsf{ISUNBOUND}\ k\ a] \quad \text{with } k = \Gamma(v) \tag{3.6}$$

with-expressions

A **with**-expression introduces new constrained variables. These are allocated in the store and inserted into the environment in which the body of the expression is compiled. The variables are removed from the stack after execution of the body.

$$\begin{aligned}&\mathcal{C}[\![\mathbf{with}\ v_1 \ldots v_n\ \mathbf{in}\ e]\!]\,\Gamma = \\ &\qquad [\mathsf{ALLOC}\ n] +\!\!+ \mathcal{C}[\![e]\!]\,\Gamma^{+n}.[v_1 \mapsto 0, \ldots, v_n \mapsto n-1] +\!\!+ [\mathsf{POP}\ n]\end{aligned} \tag{3.7}$$

Conjunctions

A conjunction is compiled into a series of instruction lists, one for each atom in the conjunction. For each of piece of code, a new process is spawned, which takes the current environment with it. The parent process skips all instructions of the new processes.

$$\begin{aligned}&\mathcal{C}[\![e_1 \,\&\, \ldots \,\&\, e_n]\!]\,\Gamma = \\ &\qquad [\mathsf{SPAWN}\ |\Gamma|+1\ a_1, \ldots, \mathsf{SPAWN}\ |\Gamma|+1\ a_n, \mathsf{JUMP}\ a+1] +\!\!+ \\ &\qquad {}_{a_1}\big|\mathcal{C}[\![e_1]\!]\,\Gamma +\!\!+ [\mathsf{JUMP}\ a+1] +\!\!+ \\ &\qquad \cdots +\!\!+ \\ &\qquad {}_{a_{n-1}}\big|\mathcal{C}[\![e_{n-1}]\!]\,\Gamma +\!\!+ [\mathsf{JUMP}\ a+1] +\!\!+ {}_{a_n}\big|\mathcal{C}[\![e_n]\!]\,\Gamma\big|_a\end{aligned} \tag{3.8}$$

The number of processes spawned can be reduced if the code of one process is executed by the parent. This is a simple optimisation to lower the communication and administration overhead.

The alternate compilation scheme, which does not create a new process for the last expression e_n, is a slight modification of (3.8):

$$\begin{aligned}
&\mathcal{C}'[\![e_1 \;\&\; \dots \;\&\; e_n]\!]\,\Gamma = \\
&\qquad [\mathsf{SPAWN}\;|\Gamma|+1\;a_1, \dots, \mathsf{SPAWN}\;|\Gamma|+1\;a_{n-1}, \mathsf{JUMP}\;a] \;+\!\!+ \\
&\qquad {}_{a_1}\big|\mathcal{C}[\![e_1]\!]\,\Gamma \;+\!\!+\; [\mathsf{JUMP}\;e+1] \;+\!\!+ \\
&\qquad \cdots \;+\!\!+ \\
&\qquad {}_{a_{n-1}}\big|\mathcal{C}[\![e_{n-1}]\!]\,\Gamma \;+\!\!+\; [\mathsf{JUMP}\;e+1] \;+\!\!+\; {}_{a}\big|\mathcal{C}[\![e_n]\!]\,\Gamma\big|_e
\end{aligned} \tag{3.9}$$

This compilation sequence especially suppresses spawning of a new process for a conjunction of just a single atom.

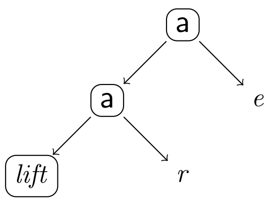
Atoms

Atoms may be tell equalities or functional expressions.

Tell-equalities A tell equality forces the evaluation of the right hand side, i. e. the expression can be compiled by the strict compilation scheme. Nevertheless, the value may be an unbound variable in the store, in which case the process has to wait until it is bound. This is done by the following instruction sequence:

$$\varphi(k) = [\mathsf{LOCK}, \mathsf{ISBOUND}\;0\;4, \mathsf{TELL}\;k, \mathsf{JUMP}\;3, \mathsf{SUSPEND}\;0, \mathsf{JUMP}\;\; -6]$$

$\varphi(k)$ checks if the value is bound, if so, it is propagated to the store, if not, the process suspends and returns to the check later.
After the execution of TELL there may be sub-terms on the stack that have to be evaluated by new processes according to the semantic rule (*tellP*). The next fragment $\psi(a)$ of code first checks if there are sub-terms; in that case, there is a fresh store variable on the stack (cf. drawing on page 70). As a second step, a process is spawned for each variable-expression-pair in a small loop. This is accomplished by applying the *lift* combinator to the variable-expression pair, i. e. unwinding the graph



in a new process. The *lift* combinator just wraps a tell-equality into a supercombinator:

$$\textbf{def}\;\; \textit{lift}\; r\; e = r =:= e$$

As the *lift* combinator is required to compile tell equalities,

$$\mathcal{C}[\![(\textbf{def}\;\; \textit{lift}\; r\; e = r =:= e) : \mathbf{C}]\!]$$

has to be included in all compilations.

So, the $\psi(a)$ sequence looks like this:

$$\begin{aligned}\psi(a) = [&\mathsf{ISBOUND}\ 0\ 3, \mathsf{POP}\ 1, \mathsf{JUMP}\ 8, \mathsf{SPAWN}\ 2\ 3, \mathsf{POP}\ 2,\\ &\mathsf{JUMP}\ -5, \mathsf{PUSHFUN}\ a\ 2, \mathsf{MKAP}, \mathsf{MKAP}, \mathsf{UNWIND}]\end{aligned}$$

a is the (absolute) address of the *lift* combinator. As one can see, a bound value returned by TELL is just popped off the stack and execution continues after the $\psi(a)$ sequence.
Glueing all pieces together, the compilation scheme for tell equalities is this:

$$\mathcal{C}[\![v =:= e]\!]\,\Gamma = \mathcal{F}^{\mathsf{S}}[\![e]\!]\,\Gamma + \varphi(k) + \psi(a) \text{ with } k = \Gamma(v) \tag{3.10}$$

At this point, a cheap optimisation comes in. With compilation scheme (3.10), a new process is spawned for each sub-term. This is of course what FATOM's operational semantics requires. But spawning processes is quite expensive and should be avoided if possible. All processes may be executed at the same time or in any sequential order that is not contradicting the constraints. So, instead of spawning plenty of processes, the subexpressions can also be evaluated by the same process, one after the other. A slight change in $\psi(a)$ implements this compilation scheme:

$$\begin{aligned}\psi'(a) = [&\mathsf{ISBOUND}\ 0\ 3, \mathsf{POP}\ 1, \mathsf{JUMP}\ 6,\\ &\mathsf{PUSHFUN}\ a\ 2, \mathsf{MKAP}, \mathsf{MKAP}, \mathsf{EVAL}, \mathsf{JUMP}\ -7]\end{aligned}$$

Instead of spawning a process, the *lift* combinator is just executed by the same process that evaluated the tell-equality.

Functional expressions Functional expressions may be applications, local definitions with **let**, or case analyses.
An application is compiled in a strict context as stated by Definition 3.11:

$$\mathcal{C}[\![e_1\ e_2]\!]\,\Gamma = \mathcal{F}^{\mathsf{S}}[\![e_1\ e_2]\!]\,\Gamma \tag{3.11}$$

Local definitions are compiled in a lazy context. Their instruction sequences are appended in reversed order such that the first local variable is on top of the stack.

$$\begin{aligned}\mathcal{C}[\![\mathbf{let}\ v_1 = e_1\,; \ldots ; v_n = e_n\ \mathbf{in}\ e]\!]\,\Gamma = \\ \mathcal{F}^{\mathsf{L}}[\![e_n]\!]\,\Gamma + \mathcal{F}^{\mathsf{L}}[\![e_{n-1}]\!]\,\Gamma^{+1} \cdots + \mathcal{F}^{\mathsf{L}}[\![e_1]\!]\,\Gamma^{+(n-1)} + \\ \mathcal{C}[\![e]\!]\,\Gamma^{+n}.[v_1 \mapsto 0, \ldots, v_n \mapsto n-1] + [\mathsf{POP}\ n]\end{aligned} \tag{3.12}$$

For a **case**-expression, the test expression is compiled in a strict context and a CASEJUMP instruction selects the branch. The first instruction of each branch splits the constructor and the branch's expression is compiled in an environment enriched by the local variables the sub-terms are bound to. An unconditional jump to the end of the sequence skips subsequent branches.

$$\begin{aligned}
&\mathcal{C}\left[\!\!\left[\begin{array}{l}\textbf{case } e \textbf{ of}\\ \quad \{t_1\}\, v_{11} \ldots v_{1n_1} \to e_1\,;\\ \quad \vdots \\ \quad \{t_m\}\, v_{m1} \ldots v_{mn_m} \to e_m\end{array}\right]\!\!\right] \Gamma = \\
&\qquad \mathcal{F}^{\mathsf{S}}[\![e]\!]\,\Gamma \mathbin{+\!\!+} [\mathsf{CASEJUMP}\ t_1\ a_1 \ldots t_m\ a_m] \mathbin{+\!\!+} \\
&\qquad [\mathsf{SPLIT}\ n_1]_{a_1} \mathbin{+\!\!+} \mathcal{C}[\![e_1]\!]\,\Gamma^{+n_1}.[v_{11} \mapsto 0, \ldots, v_{1n_1} \mapsto n_1 - 1] \mathbin{+\!\!+} \\
&\qquad [\mathsf{POP}\ n_1, \mathsf{JUMP}\ a+1] \mathbin{+\!\!+} \\
&\qquad \cdots \mathbin{+\!\!+} \\
&\qquad [\mathsf{SPLIT}\ n_m]_{a_m} \mathbin{+\!\!+} \mathcal{C}[\![e_m]\!]\,\Gamma^{+n_m}.[v_{n1} \mapsto 0, \ldots, v_{mn_m} \mapsto n_m - 1] \mathbin{+\!\!+} \\
&\qquad [\mathsf{POP}\ n_m]_a
\end{aligned} \tag{3.13}$$

3.4.2 Compiling functional abstractions

Functional expressions consist of applications, **let**- and **case**-expressions, constructor applications, and built-in functions.
As already explained, strict and lazy context is distinguished because for expressions in a strict context, it is possible to inline built-in functions to avoid the overhead of graph construction and unwinding the spine.
The functional compilation schemes look nearly the same as those for constraint abstractions with the difference that updates have to be performed. Constraint abstractions do not require updates as they do not return a result on the stack.
The compilation scheme for a functional supercombinator looks like this:

$$\begin{aligned}
&\mathcal{F}[\![(\textbf{def}\ \ v\ v_1 \ldots v_n = e) : \mathbf{F}]\!] = \\
&\qquad \mathcal{F}^{\mathsf{S}}[\![e]\!]\,[v_1 \mapsto 0, \ldots, v_n \mapsto n-1] \mathbin{+\!\!+} \\
&\qquad [\mathsf{UPDATE}\ n, \mathsf{POP}\ n, \mathsf{UNWIND}]
\end{aligned} \tag{3.14}$$

The following two sections show the translation functions for strict and lazy contexts.

Compiling functional expressions in strict context

Numbers Natural and floating point numbers are directly translated into the corresponding push instructions:

$$\mathcal{F}^{\mathsf{S}}[\![n]\!]\,\Gamma = [\mathsf{PUSHNAT}\ n] \qquad \text{with } n \in \mathit{Nat} \tag{3.15}$$

$$\mathcal{F}^{\mathsf{S}}[\![x]\!]\,\Gamma = [\mathsf{PUSHFLOAT}\ x] \qquad \text{with } x \in \mathit{Float} \tag{3.16}$$

Constructors For a constructor call, pointers to its ar arguments have to be on the stack, i. e. only saturated constructors can be compiled.

$$\begin{aligned}
&\mathcal{F}^{\mathsf{S}}[\![\mathit{Pack}\{t\ ar\}\, e_1 \ldots e_{ar}]\!]\,\Gamma = \\
&\qquad \mathcal{F}^{\mathsf{L}}[\![e_{ar}]\!]\,\Gamma \mathbin{+\!\!+} \mathcal{F}^{\mathsf{L}}[\![e_{ar-1}]\!]\,\Gamma^{+1} \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} \mathcal{F}^{\mathsf{L}}[\![e_1]\!]\,\Gamma^{+(n-1)} \mathbin{+\!\!+} \\
&\qquad [\mathsf{PACK}\ t\ ar]
\end{aligned} \tag{3.17}$$

The arguments to a constructor must be compiled by the lazy scheme as they might or might not be evaluated. In this sense, a constructor application is like function application.

Application of built-in functions For built-in functions, the corresponding machine instruction can be inlined (unlike in lazy context, see below).

- Binary operators:

$$\mathcal{F}^{\mathsf{S}}[\![e_1 \oplus e_2]\!]\,\Gamma = \mathcal{F}^{\mathsf{S}}[\![e_2]\!]\,\Gamma +\!\!+ \mathcal{F}^{\mathsf{S}}[\![e_1]\!]\,\Gamma^{+1} +\!\!+ [\chi_1(\oplus)]$$
$$\text{with } \chi_1 = \begin{bmatrix} + \mapsto \mathsf{ADD}, - \mapsto \mathsf{SUB}, * \mapsto \mathsf{MUL}, / \mapsto \mathsf{DIV}, \\ < \mapsto \mathsf{LT}, \leqslant \mapsto \mathsf{LE}, == \mapsto \mathsf{EQ}, \\ \neq \mapsto \mathsf{NE}, \geqslant \mapsto \mathsf{GE}, > \mapsto \mathsf{GT}, \\ \&\& \mapsto \mathsf{AND}, \| \mapsto \mathsf{OR} \end{bmatrix} \tag{3.18}$$

- Unary built-in functions:

$$\mathcal{F}^{\mathsf{S}}[\![b\;e]\!]\,\Gamma = \mathcal{F}^{\mathsf{S}}[\![e]\!]\,\Gamma +\!\!+ [\chi_2(b)] \quad \text{for } b \in \mathit{Builtin}$$
$$\text{with } \chi_2 = \begin{bmatrix} \mathit{neg} \mapsto \mathsf{NEG}, \mathit{not} \mapsto \mathsf{NOT}, \\ \mathit{natToFloat} \mapsto \mathsf{TOFLOAT}, \\ \mathit{floatToNat} \mapsto \mathsf{TONAT} \end{bmatrix} \tag{3.19}$$

- Builtin constant application forms:

$$\mathcal{F}^{\mathsf{S}}[\![b]\!]\,\Gamma = [\chi_3(b)] \quad \text{for } b \in \mathit{Builtin}$$
$$\text{with } \chi_3 = [\mathit{error} \mapsto \mathsf{ERROR}, \mathit{nope} \mapsto \mathsf{NOPE}] \tag{3.20}$$

let-expressions The compilation of local definitions looks nearly the same as for local definitions in constraint abstractions. The difference is that a SLIDE instead of a POP instruction removes the local variables from the stack as there is the result of e on top which must not be removed.

$$\begin{aligned} &\mathcal{F}^{\mathsf{S}}[\![\textbf{let } v_1 = e_1\,;\ldots;v_n = e_n \textbf{ in } e]\!]\,\Gamma = \\ &\qquad \mathcal{F}^{\mathsf{L}}[\![e_n]\!]\,\Gamma +\!\!+ \mathcal{F}^{\mathsf{L}}[\![e_{n-1}]\!]\,\Gamma^{+1} \cdots +\!\!+ \mathcal{F}^{\mathsf{L}}[\![e_1]\!]\,\Gamma^{+(n-1)} +\!\!+ \\ &\qquad\quad \mathcal{F}^{\mathsf{S}}[\![e]\!]\,\Gamma^{+n}.[v_1 \mapsto 0, \ldots, v_n \mapsto n-1] +\!\!+ [\mathsf{SLIDE}\ n] \end{aligned} \tag{3.21}$$

case-expressions Like compilation of **let**-expressions does not differ much between the $\mathcal{C}$ and $\mathcal{F}^{\mathsf{S}}$ scheme, translation of **case** is also nearly the same as in the previous section:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\textbf{case } e \textbf{ of}\\ \quad \{t_1\}\, v_{11} \ldots v_{1n_1} \to e_1\,;\\ \quad \vdots\\ \quad \{t_m\}\, v_{m1} \ldots v_{mn_m} \to e_m\end{array}\right]\!\!\right]\Gamma =$$

$$\begin{aligned}\mathcal{F}^{\mathsf{S}}[\![e]\!]\,\Gamma \mathbin{+\!\!+} [\mathsf{CASEJUMP}\ t_1\ a_1 \ldots t_m\ a_m] \mathbin{+\!\!+}\\ [\mathsf{SPLIT}\ n_1]_{a_1} \mathbin{+\!\!+} \mathcal{C}[\![e_1]\!]\,\Gamma^{+n_1}.[v_{11} \mapsto 0, \ldots, v_{1n_1} \mapsto n_1 - 1] \mathbin{+\!\!+}\\ [\mathsf{SLIDE}\ n_1, \mathsf{JUMP}\ a+1] \mathbin{+\!\!+}\\ \cdots \mathbin{+\!\!+}\\ [\mathsf{SPLIT}\ n_m]_{a_m} \mathbin{+\!\!+} \mathcal{C}[\![e_m]\!]\,\Gamma^{+n_m}.[v_{n1} \mapsto 0, \ldots, v_{mn_m} \mapsto n_m - 1] \mathbin{+\!\!+}\\ [\mathsf{SLIDE}\ n_m]_a\end{aligned} \tag{3.22}$$

Default case Any other expression is compiled with the lazy scheme and forced to be evaluated by the EVAL instruction:

$$\mathcal{F}^{\mathsf{S}}[\![e]\!]\,\Gamma = \mathcal{F}^{\mathsf{L}}[\![e]\!]\,\Gamma \mathbin{+\!\!+} [\mathsf{EVAL}] \tag{3.23}$$

Compiling functional expressions in lazy context

Numbers and constructors are compiled exactly the same way as in strict context:

$$\mathcal{F}^{\mathsf{L}}[\![n]\!]\,\Gamma = \mathcal{F}^{\mathsf{S}}[\![n]\!]\,\Gamma \qquad \text{with } n \in \mathit{Nat} \tag{3.24}$$

$$\mathcal{F}^{\mathsf{L}}[\![x]\!]\,\Gamma = \mathcal{F}^{\mathsf{S}}[\![x]\!]\,\Gamma \qquad \text{with } x \in \mathit{Float} \tag{3.25}$$

$$\mathcal{F}^{\mathsf{L}}[\![\mathit{Pack}\{t\ ar\}\, e_1 \ldots e_{ar}]\!]\,\Gamma = \mathcal{F}^{\mathsf{S}}[\![\mathit{Pack}\{t\ ar\}\, e_1 \ldots e_{ar}]\!]\,\Gamma \tag{3.26}$$

Variables On compilation of variables the corresponding pointer is pushed onto the stack.

$$\mathcal{F}^{\mathsf{L}}[\![v]\!]\,\Gamma = [\mathsf{PUSHVAR}\ k] \tag{3.27}$$

Compilation of supercombinator application If a variable v is a supercombinator name a pointer to the code of its body is pushed on the stack. This address a_v is not known until all supercombinators are compiled and the lengths of their instruction lists is established.

$$\mathcal{F}^{\mathsf{L}}[\![v]\!]\,\Gamma = [\mathsf{PUSHFUN}\ a_v\ n] \quad \text{with supercombinator } v\, v_1 \ldots v_n = e \tag{3.28}$$

Application To translate an application $e_1\ e_2$ both expressions are compiled and their graphs connected by an application node:

$$\mathcal{F}^{\mathsf{L}}[\![e_1\ e_2]\!]\,\Gamma = \mathcal{F}^{\mathsf{L}}[\![e_2]\!]\,\Gamma \mathbin{+\!\!+} \mathcal{F}^{\mathsf{L}}[\![e_1]\!]\,\Gamma^{+1} \mathbin{+\!\!+} [\mathsf{MKAP}] \tag{3.29}$$

Application of built-in functions Built-in functions are a bit special in lazy context: like for supercombinators a pointer is pushed and a certain instruction template is added to the instruction sequence of the program, one for each built-in function used, similar to the *lift* combinator. The address a_b is the address of the code template $\Theta(b)$ for the built-in function b.

$$\mathcal{F}^{\mathsf{L}}[\![b]\!]\,\Gamma = [\mathsf{PUSHFUN}\ a_b\ n] \quad \text{for } b \in \mathit{Builtin} \cup \mathit{Binop} \tag{3.30}$$

The code templates Θ are the following:

$$\begin{aligned}
\Theta(\oplus) &= [\mathsf{PUSHVAR}\ 1, \mathsf{EVAL}, \mathsf{PUSHVAR}\ 1, \mathsf{EVAL}, \chi(\oplus), \mathsf{UPDATE}\ 2, \mathsf{POP}\ 2] \\
&\quad\ \text{for } \oplus \in \mathit{Binop} \\
\Theta(b) &= [\mathsf{PUSHVAR}\ 0, \mathsf{EVAL}, \chi(b), \mathsf{UPDATE}\ 1, \mathsf{POP}\ 1] \\
&\quad\ \text{for } b \in \{\mathit{not}, \mathit{neg}, \mathit{natToFloat}, \mathit{floatToNat}\} \\
\Theta(b) &= [\chi(b), \mathsf{UPDATE}\ 0] \\
&\quad\ \text{for } b \in \{\mathit{noPE}, \mathit{error}\}
\end{aligned}$$

$\chi = \chi_1 \cup \chi_2 \cup \chi_3$ with the χ_i environments as given in the previous section.

let-expressions Compilation of a **let**-expression is the same as for $\mathcal{F}^{\mathsf{S}}$ except that the body is compiled with the lazy scheme.

$$\begin{aligned}
&\mathcal{F}^{\mathsf{S}}[\![\textbf{let}\ v_1 = e_1\,;\ldots;v_n = e_n\ \textbf{in}\ e]\!]\,\Gamma = \\
&\qquad\qquad \mathcal{F}^{\mathsf{L}}[\![e_n]\!]\,\Gamma \mathbin{+\!\!+} \mathcal{F}^{\mathsf{L}}[\![e_{n-1}]\!]\,\Gamma^{+1} \cdots \mathbin{+\!\!+} \mathcal{F}^{\mathsf{L}}[\![e_1]\!]\,\Gamma^{+(n-1)} \mathbin{+\!\!+} \\
&\qquad\qquad\quad \mathcal{F}^{\mathsf{L}}[\![e]\!]\,\Gamma^{+n}.[v_1 \mapsto 0, \ldots, v_n \mapsto n-1] \mathbin{+\!\!+} [\mathsf{SLIDE}\ n]
\end{aligned} \tag{3.31}$$

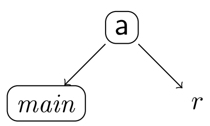
case-expressions Case analyses cannot be compiled in a lazy context. But it is possible to do a program transformation to compile them. Given an expression in lazy context containing a case analysis like $f \ldots (\textbf{case}\ e\ \textbf{of}\ b_1\,;\ldots;b_n) \ldots$, this expression can be transformed into $f \ldots (f'\ v_1\ \ldots\ v_m) \ldots$ with a fresh supercombinator

$$\textbf{def}\ f'\ v_1 \ldots v_m = \textbf{case}\ e\ \textbf{of}\ b_1\,;\ldots;b_n.$$

The $v_1, \ldots, v_m$ are all variables in scope in the original expression which are used in the case analysis.

3.4.3 Bootstrapping the machine

Following Section 3.3.3, the machine starts execution with the first loaded instruction and an empty heap and stack. To convince the machine to execute the *main* supercombinator, a little bootstrapping code has to be prepended to the instruction list. The *main* supercombinator has one argument r for the result of the computation, this word has to be allocated in the store. Then the graph



has to be unwound. This task is performed by the bootstrapping sequence

$$\beta(a) = [\mathsf{ALLOC}\ 1, \mathsf{PUSHFUN}\ a\ 1, \mathsf{MKAP}, \mathsf{UNWIND}]$$

with a being the address of the *main* supercombinator.

3.4.4 Example compilation

To conclude the section on code generation a tiny example is presented. The compilation of the FATOM program

$$\textbf{def}\ \ main\ r = r =:= 1 + 2$$

yields this machine code:

```
0   ALLOC 1         ; Beginning of bootstrapping.
1   PUSHFUN 4 1     ; Supercombinator main.
2   MKAP
3   UNWIND          ; End of bootstrapping.
4   PUSHNAT 2       ; Beginning of main supercombinator
5   PUSHNAT 1
6   ADD             ; Strict addition.
7   LOCK            ; Beginning of φ.
8   ISBOUND 0 4
9   TELL 1
10  UNLOCK
11  JUMP 3
12  SUSPEND 0
13  JUMP −6         ; End of φ.
14  ISBOUND 0 3     ; Beginning of ψ.
15  POP 1
16  JUMP 8
17  SPAWN 2 3
18  POP 2
19  JUMP −5
20  PUSHFUN 4 26
21  MKAP
22  MKAP
23  UNWIND          ; End of ψ.
24  POP 2
25  UNWIND          ; End of main.
26  PUSHVAR 1       ; Beginning of lift.
27  ...
```

The code of *lift* has been omitted.

Chapter 4

Implementation

This chapter briefly outlines the design and structure of a prototypical implementation of ATAF. To generate machine code a simple compiler has also been developed which is described in the second section. The third section concludes with a few performance measurements on a parallel machine.
Practical hints on how to build and use the implementation can be found in Appendix D.

4.1 `ataf` – the ATAF abstract machine interpreter

The implementation of the abstract machine, called `ataf`, is a machine code interpreter. This means it reads in a sequence of machine instructions and takes the appropriate action according to their semantics. This is in contrast to a native implementation that generates machine code for particular architecture. The latter is much faster but it is a very time-consuming task to implement. The former is slow but can be produced in a time-frame suitable for a diploma thesis and is sufficient to show that the concepts developed in Chapters 2 and 3 work in real life.

4.1.1 Overview of the implementation

The interpreter is written in HASKELL [PAB+03] but makes use of some language extensions by the GHC compiler.
Furthermore, it uses MPI [SOHL+96] as communication library. MPI is a standard message-passing suite available for C, C++, and FORTRAN that offers point-to-point and group communication. It is integrated into the program via HASKELL's foreign function interface [CFH+03]. An MPI program is started as one or more operating system processes, ideally each having a CPU for itself. These operating system processes correspond to ATAF instances.
Each instance has to perform two tasks:

1. execute the local set of processes and

2. handle requests from other instances.

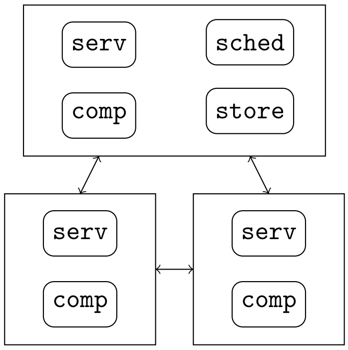


Figure 4.1 Instances and threads of `ataf`.

Besides these, the store has to be located somewhere. There are basically two possibilities: a centralised or a distributed store. The distributed implementation is possibly faster but complicated to realise, for this reason `ataf` uses a centralised store that is operated by one of the instances. It is also practical to have one more task, the scheduler, that keeps track of the workloads on each instance and assigns PIDs.

These four tasks must be handled concurrently, so the natural solution is multithreading using CONCURRENT HASKELL (see [PGF96]) with two or four threads per instance. One instance runs two threads for the store (`store`) and scheduler (`sched`) in addition to a computation (`comp`) and service (`serv`) thread, which also run on all other instances. Figure 4.1 shows the situation for three instances. The arrows indicate MPI messages transmitted between the instances.

The machine state, i. e. the store, the heaps, the stack, all register sets, and the `Ready` queue are encapsulated in a state monad thus utilising HASKELL's excellent support for monadic programming. This state monad is actually a state transformer (in the spirit of [Jon95]) implemented with the monad transformer library distributed with GHC. This state transformer is embedded in the IO monad such that input/output inside the machine monad is possible without the necessity to program everything as an IO action.

Furthermore, due to the monadic style the implementation of the machine instructions looks nearly like their specification in Chapter 3, e. g. the implementation of the PUSHFUN instruction looks like this:

```
iPushfun a ar = do d ← allocate
                   (HEAP, d) %= FUN a ar
                   push (PTR d)
                   inc IP
```

4.1.2 The threads in detail

The four threads composing a machine instance are now described in more detail.

The `store` thread

The `store` thread handles the requests for locking and unlocking the store, allocating words, writing and reading values, queueing in SUSPEND queues, and

waking up suspended processes.
The store segment itself is implemented as a functional array (`DiffArray`), which has constant access times in a single threaded computation. This single threadedness is guaranteed by the state monad. The `SUSPEND` queues are stored in an array of FIFO-buffers.

The `sched` thread

The `sched` thread records the number of processes currently running on each instance. Therefore, new PIDs, i. e. an assignment for a process to an instance, have to be requested from this thread. Of course, every time a process has terminated a message is sent to the scheduler to update the records.

The `serv` thread

The `serv` threads has a support function: every time a new process is spawned its environment has to be set up, i. e. a new stack and heap has to be reserved, and the process has to be enqueued in `Ready`. The `serv` thread receives the contents of the new heap and stack and sets up an appropriate environment.
It also queues processes in the `Ready` that were waken up by the store.

The `comp` thread

The `comp` thread performs the actual execution of processes. It runs a dispatch loop, looking up the current instruction of the current process and calling the corresponding function. Each machine instruction `INSTR` is implemented by a monadic operation `iInstr`, like the above example of `iPushfun`.
This thread does not handle any requests but it sends a lot of them to the store, the scheduler or other instances when spawning a new process.
The stack and heap segments of all processes are simply `DiffArray`s that are allocated whenever a new process is created. As the `serv` thread also accesses this process area when a new environment is set up, it is only modified inside the STM (software transactional memory, see [DHM+06]) monad. This avoids the necessity for lock variables or semaphores to achieve synchronisation.

4.1.3 MPI and threads

Unfortunately it is not as simple as described above: Most MPI implementations are not thread-safe, so it is not possible that several threads of the same program issue MPI calls. But all threads introduced so far have to use MPI functions to send or receive messages to or from other instances.
To overcome this situation a third (or fifth) thread `disp` is introduced which does all the MPI calls and dispatches the messages to their receivers. This technique is also known as *funnelling*. Figure 4.2 shows the situation for an instance without a store and a scheduler. The communication between the threads is done by channels, which are unbounded FIFO-buffers.

4.1.4 Garbage collection

One issue has been ignored up to now: the specification of the ATAF abstract machine describes how heap and store words are allocated but no precautions are

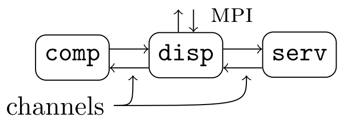


Figure 4.2 Two `ataf` threads with the communication dispatcher.

taken for the case that no free words are left. At this point, garbage collection comes into play.
The `ataf` implementation has two garbage collectors built in: one for the store and one for the heaps of the processes.

Heap garbage collector

The garbage collector for the heaps is a copying collector as developed in [FY69, Che70]. For copying collection to work, the available space is divided into two parts, one being the allocation area. If the allocation area is full, all words in the allocation area directly reachable by a pointer from the stack are evacuated into the second half of the heap, the to-space. The original word and the pointer on the stack are replaced by a forward pointer to the copied word.
In a second step all words in the to-space are scanned for pointers into the old allocation area. If such a pointer is found the word it points to is copied into the to-space and the pointer is updated to the new address. If a pointer from to-space points to one of the forwarding pointers from evacuation it is replaced by this address; no copying is necessary as the word is already copied.
After scanning all words in to-space, the two halves of the heap change roles, and the to-space is now the allocation area.
This algorithm has the advantage that no garbage has to be scanned but only words that are still in use. The disadvantages are that the effectively available memory is divided by two, which is not a severe problem on virtual memory systems, and that data locality is affected because of the breadth-first copying at each garbage collection.

Store garbage collector

The garbage collector of the store is a mark-and-scan collector [McC60]. The reason not to use copying collection is that this algorithm changes pointers in the stack and the heap. This would imply much more communication to change all the addresses in all processes after a garbage collection in the store. Using mark-and-scan collection requires a small change in memory management from the specification in Chapter 3: all free words are linked into a free-list, and the store pointer TP points to the head of the list. Whenever a word is allocated, TP is set to the pointer stored in the head of the free-list, thus advancing to the end of the list. When this end, indicated by a ⟨NIL⟩ word, is reached, garbage collection is initiated.
At this point a message is sent to all instances to return a list of pointers into the store. Now all store words still in use are marked (store words have an additional tag for this purpose) starting with the words referenced by the addresses returned from processes and recursively following all pointers in ⟨PCK⟩ words.

The second step is to linearly scan the entire store, insert all unmarked words into the free list, and untag all words still in use.
Mark-and-scan collection has the drawback that it always requires to scan the entire store, i. e. also the garbage has to be scanned.

4.1.5 Startup and termination

When `ataf` is started each instance reads the instruction file given on the command-line and loads the code into the code segment. Then one process is created which starts with the first instruction.
The implementation terminates when there are no processes left or some error occurred (like division by zero or call of the *error* function). As every termination of a process is registered with the scheduler, this thread notices when all processes have vanished and sends out a message to all instances to exit.

4.1.6 Some options concerning performance

The implementation provides two options that affect performance:

1. The flag `--store-island` reserves one node for the store and the scheduler, i. e. there is one fewer processing element for computation. This option was included experimentally but the impact on performance is unfortunately almost negligible (see Section 4.3).

2. The flag `--fast-prop` slightly changes the semantics of the `TELL` instruction: if the result of the computation is a constructor and one or more sub-terms of this constructor are numbers, e. g. $Pack\{2, 2\}$ 5 6, these numbers are directly propagated to the store. This reduces the amount of work to deal with sub-terms and is particular usable for lists of numbers.

4.2 `fc` – a compiler for FATOM

`fc` is a simple compiler, also written in HASKELL, that takes one or more FATOM source files and writes an ASCII file with one ATAF instruction per line.
The compiler consists of three modules:

1. the parser, which is a recursive descent parser implemented with the monadic parser combinator library PARSEC [LM01],

2. the context checker, which implements the inference algorithm of Appendix C, and checks for presence of a *main* function etc., and

3. the code generator, which is a simple syntax directed translator with address back-patching.

The code generator only translates supercombinators which are actually used in the program. For this purpose the set of used supercombinators is calculated with a fixed point iteration like for reachability in a graph.
The compiler offers two command-line options concerning code generation that correspond to the alternate compilation schemes presented in Section 3.4.1:

- `--no-spawn` corresponds to ψ' on page 82 and,
- `--opt-conj` corresponds to $\mathcal{C}'$ on page 81.

4.2.1 Testsuite

In addition to the compiler and the machine interpreter, there is a small testsuite implemented by a BASH script, which compiles a certain set of FATOM programs, runs them on the machine, and checks if the output is correct. Adding new tests is fairly simple, as one only has to add a new directory to the testcases containing four files:

- `input` contains a space separated list of FATOM source files,
- `fcflag` contains command-line flags for the call to `fc`,
- `atafflag` is the same for the call to `ataf`, and
- `expect` contains the expected output, which is literally compared to the output of the test run.

4.3 Measuring performance

Despite being a prototypical implementation, programs interpreted by `ataf` should show a decreasing execution time on several processors. This, of course, only holds if the program utilises several processes with a suitable granularity. Section 2.1 showed several example programs which should be suitable for parallel execution. The performance of two of them, Quicksort and *pfarm*,g will be examined very briefly in this section.
The measurements took place on the TUSCI cluster of the KBS group (Kommunikations- und Betriebssysteme) at Berlin Technical University (`http://kbs.cs.tu-berlin.de/projects/tusci.htm`).
The TUSCI cluster consists of 16 nodes each operating two 1.7 MHz Pentium IV Xeon CPUs and one Gigabyte of RAM. They are connected by the Scalable Coherent Interface [ANS92] by a two-dimensional torus topology.
Each node runs the GNU/LINUX operating system and job management is under control of CCS [KR98].
The ATAF machine implementation was linked against the MP-MPICH v1.3.0 [PSF+06] implementation of the MPI standard.
The time measurements were done by the `--profiling` option of `ataf` that reports the overall duration of execution.
The quantity parallel programmers usually are interested in is the *speed-up*.

Definition 4.1 *(Speed-up) The* speed-up S_n *is the ratio of execution time of a program on one processing element* T_1 *and execution time on* n *processing elements* T_n*:*

$$S_n = \frac{T_1}{T_n}$$

An ideal speed-up would be linear in n which is not possible due to AMDAHL's law [Amd67] stating that the maximum speed-up of a program is

$$S_n = \frac{1}{f + \frac{1-f}{n}} \tag{4.1}$$

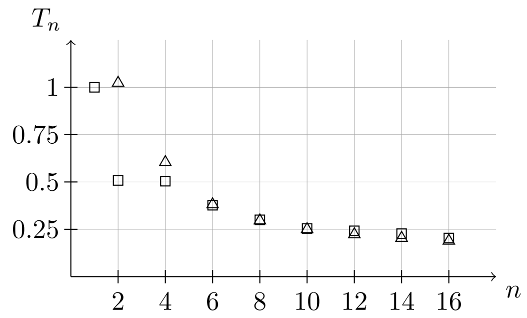


Figure 4.3 Speed-up of *pfarm*, with `--store-island` (△) and without (□).

with f being the program's fraction of code that cannot be parallelised. As f cannot be arbitrarily small, S_n approaches $\frac{1}{f}$ in the limit $n \to \infty$. That is, the speed-up saturates for greater n.
If T_1 is defined to be 1 the execution time is the reciprocal of the speed-up, which is the quantity measured in the following two sections. Defining T_1 to 1 and normalising the execution times of parallel runs accordingly also eases comparisons between different measurements. As an ideal speed-up would result in a straight line plot, execution times are supposed to look hyperbolic as the measurements will show.

4.3.1 Speed-up of *pfarm*

The first subject is the calculation of the square roots of the numbers 1 to 500. The *sqrt* function performs a Newton iteration to approximate a square root and is shown in Figure 4.1. As each square root can be calculated independently of all others, the *pfarm* abstraction can be used to iterate over the numbers 1 to 500. The *main* function for the measurement was simply:

def *main r* = *pfarm sqrt* (*enumFromTo* 1 500) *r*

It was compiled with the `--no-spawn` option to reduce the overhead and running with the `--fast-prop` option to reduce the load on the store.
Measurement were taken for 1, 2, 4, 6, 8, 10, 12, 14, and 16 processing elements, i. e. `ataf` was running on up to eight nodes. One series was with a dedicated store node (`--store-island` option), the second one without.
The results are shown in Diagram 4.3. There is no significant difference between a dedicated store node and without, apart from the fact that the former performs worse on fewer nodes. This is reasonable as a single computation node has higher load in these cases.
Concerning the dramatic speed-up of roughly factor two changing from one to two nodes it has to be kept in mind that these two instances run on the same node and thus communicate by shared memory and not via the SCI interconnect. For more than eight nodes there is still a speed up but it decreases rapidly as communication cost rise.
It is a common phenomenon that the speed-up can be improved by calculating bigger problems. The next Diagram 4.4 shows the speed-up for 500 square roots in comparison to 1000 square roots.

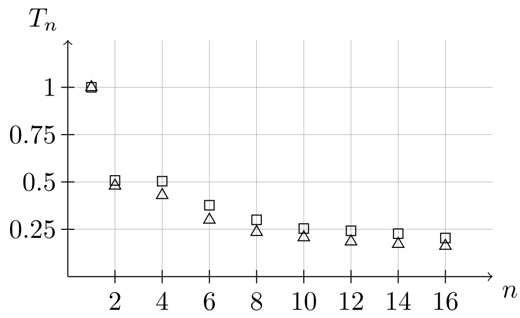


Figure 4.4 Speed-up of *pfarm* for 500 (□) and 1000 (△) square roots.

There is an improvement which is (unfortunately) not very significant.

Listing 4.1 The *sqrt* function to calculate a square root.

def *sqrt x* = *newton* (*sqrtFun x*) *sqrtDeriv* 0.0000001 (1.0 ∗ *x*)

def *sqrtFun x w* = *w* ∗ *w* − *x*

def *sqrtDeriv w* = 2.0 ∗ *w*

def *newton fun deriv eps x0* = **let** *x1* = *x0* − *fun x0* / *deriv x0*
in *if* (*abs* (*x1* − *x0*) > *eps*)
(*newton fun deriv eps x1*)
x1

4.3.2 Quicksort with granularity control

Like the previous example the Quicksort profiling was done with a `--no-spawn` compilation and using the fast propagation of `ataf`.
The *main* function is

def *main r* = *quicksortGc numbers r*

and *numbers* is a list of 100 or 500 randomly generated natural numbers between 0 and 10000, respectively. Figure 4.5 shows the result of the measurements.
Unfortunately, they are not very promising. There is a speed-up up to six or eight processing elements, afterwards, executions times look random. This behaviour is roughly the same for sorting 500 numbers as the △ in Figure 4.5 show.
One reason for this poorer performance in comparison to the *pfarm* example is that the functional computation in a Quicksort boils down to comparing numbers and constructor calls. This is much faster than calculating a square root thus shifting the ratio between computation and communication to the worse.

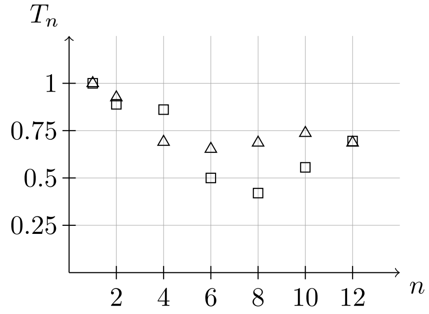


Figure 4.5 Speed-up of *quicksortGc* for 100 (□) and 500 (△) random numbers.

The constructor calls nearly all appear in a tell context and immediately lead to communication (TELL instructions). To improve the Quicksort, the communication behaviour between the store and the computations has to be optimised, e. g. by en-bloc communication.

Chapter 5

Conclusions

The preceding three chapters have shown that concurrent constraint functional programming is possible with a fairly small amount of language constructs, and that this system can be used to write parallel programs that run on real machines. The abstract machine presented is a suitable intermediate step between the abstract source form and a native implementation. However, there are certain issues that remain unanswered or need improvement, which shall be discussed in the following.

5.1 Expressiveness and usability

FATOM is intended as an intermediate representation for a compiler. The question how a high-level and usable concurrent constraint functional language can be translated into FATOM arises almost naturally. Issues like the constraint system of this new source language have to be discussed and whether they require more primitives in the underlying layers. A high-level language would certainly be a type-safe and probably statically typed language, based on the Hindley-Milner [Hin69, Mil78] type system. The integration of constraints into the type system has to be solved.
Another idea unrelated to the previous one is to prepare the system to run on a number of processing elements that varies during runtime. Being more flexible in this respect would avoid idle but allocated processors but also enable asking for more processors if a certain load limit is exceeded. On the one hand side, this requires an environment in which dynamic processor allocation is possible, which is not always the case. On the other hand side, *noPE* would no longer be constant, so another referentially transparent way of accessing and controlling the machine size has to be found.

5.2 Performance

Despite the measurable speed-up discussed in Section 4.3, work concerning the performance of the implementation remains to be done.
A machine interpreter can only serve as a prototype and to test ideas, for real world problems, it is much too slow. So, one task is the implementation of a native compiler for FATOM, i. e. a compiler which generates machine code for a

real architecture. This could be done by exchanging the ATAF instructions with machine code templates and linking against a suitable runtime system. This also includes the implementation of a complete new memory management integrated into the runtime system: in a memory and time efficient implementation, it is impossible to use tagged words of equal size in the store and the heaps. Furthermore, it is not reasonable to allocate one suspend queue for each store word but only for those that have waiting processes, e. g. using optimised data structures like hash tables.

Along with these implementation changes goes an optimising compiler. The compiler of Section 3.4 only distinguished between strict and lazy context. More contexts can be established, like e. g. number-strict context, that lead to more efficient code. Other classical optimisation techniques like inlining, tail-recursion optimisation, etc. should be implemented.

All these techniques improve the overall performance, independent of parallel or serial execution. For the concurrent part, the implementation of the tell equality is crucial, as the discussion about different compilation schemes (Section 3.4.1) and the `--fast-prop` option of `ataf` have shown. To improve performance in this part, messages to the store should be buffered and flushed once to reduce the communication and synchronisation overhead. It is not clear if new machine instructions are required for this, or if it is enough to change the existing instructions.

The next bottleneck is certainly the implementation of the store. It should be examined if a more localised implementation is possible. This would reduce the bottleneck character and also distribute the load on the different nodes more evenly.

At last, it has be investigated how a typed FATOM would help to improve code generation. A typed FATOM means a language in which all expressions are annotated with their type. This could be realised by adding type annotations during the translation from a typed high-level language to FATOM.

Appendix A

FATOM prelude

The function types are given in HASKELL syntax in a comment for documentation purposes.

A.1 PreludeFun.fatom

```
PreludeFun.fatom

Standard functional prelude of Fatom.

This file is part of the diploma thesis
"An Abstract Machine for a Concurrent (and Parallel)
Constraint Functional Programming Language"
by Florian Lorenzen <florenzATcs.tu-berlin.de>,
written in the Compiler Construction and Programming Languages group
of Prof. Dr. Peter Pepper at Berlin Technical University,
advised by Dr. Petra Hofstedt and Martin Grabmueller.
```

A.1.1 Function combination

Identity function.

$-- \; id :: \alpha \rightarrow \alpha$
def $id\ x = x$

Composing functions.

$-- \; com :: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$
def $com\ f\ g\ x = f\ (g\ x)$

A.1.2 Boolean functions

Constructors. True and false.

$-- \; True, False :: Bool$
def $True\ = Pack\{1, 0\}$
def $False = Pack\{2, 0\}$

If-"statement".

$$
\begin{array}{l}
-- \; if :: Bool \to \alpha \to \alpha \\
\textbf{def } if \; c \; t \; e = \textbf{case } c \textbf{ of } \{1\} \to t; \{2\} \to e
\end{array}
$$

Boolean *and* and *or* as supercombinators (to apply them partially).

$$
\begin{array}{l}
-- \; and, or :: Bool \to Bool \to Bool \\
\textbf{def } and \; x \; y = x \;\&\&\, y \\
\textbf{def } or \; x \; y \;\; = x \parallel y
\end{array}
$$

A.1.3 Functions on numbers

Successor and predecessor.

$$
\begin{array}{l}
-- \; succ, pred :: Nat \to Nat \\
\textbf{def } succ \; n = n + 1 \\
\textbf{def } pred \; n = n - 1
\end{array}
$$

Arithmetics.

$$
\begin{array}{l}
-- \; add, sub, mul, div :: Num \to Num \to Num \\
\textbf{def } add \; x \; y = x + y \\
\textbf{def } sub \; x \; y \; = x - y \\
\textbf{def } mul \; x \; y = x * y \\
\textbf{def } div \; x \; y \; = x \; / \; y
\end{array}
$$

Comparisons.

$$
\begin{array}{l}
-- \; lt, le, eq, ne, ge, gt :: Num \to Num \to Bool \\
\textbf{def } lt \; x \; y \; = x < y \\
\textbf{def } le \; x \; y \; = x \leqslant y \\
\textbf{def } eq \; x \; y = x == y \\
\textbf{def } ne \; x \; y = x \neq y \\
\textbf{def } ge \; x \; y = x \geqslant y \\
\textbf{def } gt \; x \; y \; = x > y
\end{array}
$$

Modulo function.

$$
\begin{array}{l}
-- \; mod :: Nat \to Nat \to Nat \\
\textbf{def } mod \; a \; b = if \; (a < b) \\
\qquad\qquad\qquad\quad a \\
\qquad\qquad\qquad\quad (mod \; (a - b) \; b)
\end{array}
$$

Test for odd- and evenness.

$$
\begin{array}{l}
-- \; even, odd :: Nat \to Bool \\
\textbf{def } even \; n = mod \; n \; 2 == 0 \\
\textbf{def } odd \; n \;\; = mod \; n \; 2 == 1
\end{array}
$$

Absolute value of a number.

$$
\begin{array}{l}
-- \; abs :: Num \to Num \\
\textbf{def } abs \; x = if \; (x < 0) \\
\qquad\qquad\quad\;\; (neg \; x) \\
\qquad\qquad\quad\;\; x
\end{array}
$$

A.1.4 Pairs

Constructor.

$-- \; P :: \alpha \to \beta \to (\alpha, \beta)$
$\textbf{def } P \; a \; b = \mathit{Pack}\,\{1,2\}\; a \; b$

Selectors.

$-- \; \mathit{fst} :: (\alpha, \beta) \to \alpha$
$\textbf{def } \mathit{fst} \; p = \textbf{case } p \textbf{ of } \; \{1\}\; a \; b \to a$

$-- \; \mathit{snd} :: (\alpha, \beta) \to \beta$
$\textbf{def } \mathit{snd} \; p = \textbf{case } p \textbf{ of } \; \{1\}\; a \; b \to b$

A.1.5 List functions

Constructors for lists.

$-- \; \mathit{Nil} :: [\alpha]$
$\textbf{def } \mathit{Nil} = \mathit{Pack}\,\{1,0\}$

$-- \; \mathit{Cons} :: \alpha \to [\alpha] \to [\alpha]$
$\textbf{def } \mathit{Cons} \; x \; xs = \mathit{Pack}\,\{2,2\}\; x \; xs$

Higher-order-functions.

The well-known *map*-, *filter*-, and *fold*-functions.

$$
\begin{array}{l}
-- \; \mathit{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\textbf{def } \mathit{map} \; f \; l = \textbf{case } l \textbf{ of} \\
\qquad\qquad\qquad\qquad \{1\} \qquad\;\; \to \mathit{Nil}; \\
\qquad\qquad\qquad\qquad \{2\}\; x \; xs \to \mathit{Cons}\; (f \; x)\; (\mathit{map}\; f \; xs)
\end{array}
$$

$$
\begin{array}{l}
-- \; \mathit{filter} :: (\alpha \to \mathit{Bool}) \to [\alpha] \to [\alpha] \\
\textbf{def } \mathit{filter} \; p \; l = \textbf{case } l \textbf{ of} \\
\qquad\qquad\qquad\qquad \{1\} \qquad\;\; \to \mathit{Nil}; \\
\qquad\qquad\qquad\qquad \{2\}\; x \; xs \to if\; (p \; x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{Cons}\; x \; (\mathit{filter}\; p \; xs)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{filter}\; p \; xs)
\end{array}
$$

$$
\begin{array}{l}
-- \; \mathit{foldlr} :: (\alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
\textbf{def } \mathit{foldl} \; f \; e \; l = \textbf{case } l \textbf{ of} \\
\qquad\qquad\qquad\qquad \{1\} \qquad\;\; \to e; \\
\qquad\qquad\qquad\qquad \{2\}\; x \; xs \to \mathit{foldl}\; f \; (f \; e \; x)\; xs
\end{array}
$$

$$
\begin{array}{l}
-- \; \mathit{foldl} :: (\alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
\textbf{def } \mathit{foldr} \; f \; e \; l = \textbf{case } l \textbf{ of}
\end{array}
$$

$$
\begin{aligned}
&\{1\} &&\rightarrow e;\\
&\{2\}\ x\ xs &&\rightarrow f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

zip is like map for two lists (it is unlike HASKELL's standard library *zip*, which pairs the element of two lists).

$$
\begin{aligned}
&-\!-\ zip :: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow [\gamma]\\
&\textbf{def}\ zip\ f\ l1\ l2 = \textbf{case}\ l1\ \textbf{of}\\
&\qquad \{1\} \qquad \rightarrow Nil;\\
&\qquad \{2\}\ x\ xs \rightarrow \textbf{case}\ l2\ \textbf{of}\\
&\qquad\qquad \{1\} \qquad \rightarrow Nil;\\
&\qquad\qquad \{2\}\ y\ ys \rightarrow Cons\ (f\ x\ y)\ (zip\ f\ xs\ ys)
\end{aligned}
$$

Lists of lists

Functions to manipulate lists of lists.
This one is often called *flatten*; it concatenates all lists of a list of lists.

$$
\begin{aligned}
&-\!-\ concat :: [[\alpha]] \rightarrow [\alpha]\\
&\textbf{def}\ concat\ l = foldl\ append\ Nil\ l
\end{aligned}
$$

partition splits a list l into a list of n list such that $concat\ (partition\ n\ l) = l$ for all $n \leqslant length\ l$.

$$
\begin{aligned}
&-\!-\ ASSERT : n \geqslant 1\\
&-\!-\ partition :: Nat \rightarrow [\alpha] \rightarrow [[\alpha]]\\
&\textbf{def}\ partition\ n\ l = \textbf{case}\ l\ \textbf{of}\\
&\qquad \{1\} \qquad \rightarrow Nil;\\
&\qquad \{2\}\ x\ xs \rightarrow \textbf{let}\ nxs = length\ l\ /\ n\\
&\qquad\qquad \textbf{in}\ parts\ n\ nxs\ l
\end{aligned}
$$

$$
\begin{aligned}
&-\!-\ parts :: Nat \rightarrow Nat \rightarrow [\alpha] \rightarrow [[\alpha]]\\
&\textbf{def}\ parts\ n\ nxs\ l = if\ (n == 1)\\
&\qquad (Cons\ l\ Nil)\\
&\qquad (\textbf{let}\ part = take\ nxs\ l;\\
&\qquad\qquad rest\ = drop\ nxs\ l\\
&\qquad\quad \textbf{in}\ Cons\ part\ (parts\ (n-1)\ nxs\ rest))
\end{aligned}
$$

Take elements apart

Dropping and selecting elements of list.

$$
\begin{aligned}
&-\!-\ ASSERT : n \leqslant length\ l\\
&-\!-\ take :: Nat \rightarrow [\alpha] \rightarrow [\alpha]\\
&\textbf{def}\ take\ n\ l = \textbf{case}\ l\ \textbf{of}\\
&\qquad \{1\} \qquad \rightarrow Nil;\\
&\qquad \{2\}\ x\ xs \rightarrow if\ (n == 0)\\
&\qquad\qquad Nil\\
&\qquad\qquad (Cons\ x\ (take\ (n-1)\ xs))
\end{aligned}
$$

$$
\begin{array}{l}
--\ \mathit{ASSERT} : n \leqslant \mathit{length}\ l \\
--\ \mathit{drop} :: \mathit{Nat} \rightarrow [\alpha] \rightarrow [\alpha] \\
\textbf{def}\ \mathit{drop}\ n\ l = \textbf{case}\ l\ \textbf{of} \\
\qquad\qquad\qquad \{1\} \qquad\quad \rightarrow \mathit{Nil}; \\
\qquad\qquad\qquad \{2\}\ x\ xs \rightarrow \mathit{if}\ (n == 0) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Cons}\ x\ xs) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{drop}\ (n-1)\ xs)
\end{array}
$$

Return all but the last element of the list.

$$
\begin{array}{l}
--\ \mathit{ASSERT} : \mathit{length}\ l \geqslant 1 \\
--\ \mathit{butlast} :: [\alpha] \rightarrow [\alpha] \\
\textbf{def}\ \mathit{butlast}\ l = \mathit{take}\ (\mathit{length}\ l - 1)\ l
\end{array}
$$

Return the last element of the list.

$$
\begin{array}{l}
--\ \mathit{ASSERT} : \mathit{length}\ l \geqslant 1 \\
--\ \mathit{last} :: [\alpha] \rightarrow \alpha \\
\textbf{def}\ \mathit{last}\ l = \mathit{hd}\ (\mathit{drop}\ (\mathit{length}\ l - 1)\ l)
\end{array}
$$

Return the list's head (partial function).

$$
\begin{array}{l}
--\ \mathit{hd} :: [\alpha] \rightarrow \alpha \\
\textbf{def}\ \mathit{hd}\ l = \textbf{case}\ l\ \textbf{of}\ \ \{2\}\ x\ xs \rightarrow x
\end{array}
$$

Return the list's tail (partial function).

$$
\begin{array}{l}
--\ \mathit{tl} :: [\alpha] \rightarrow [\alpha] \\
\textbf{def}\ \mathit{tl}\ l = \textbf{case}\ l\ \textbf{of}\ \ \{2\}\ x\ xs \rightarrow xs
\end{array}
$$

Access the *nth* element of a list. The first element has index 0.

$$
\begin{array}{l}
--\ \mathit{nth} :: \mathit{Nat} \rightarrow [\alpha] \rightarrow \alpha \\
\textbf{def}\ \mathit{nth}\ n\ l = \mathit{if}\ (n == 0) \\
\qquad\qquad\qquad\quad (\mathit{hd}\ l) \\
\qquad\qquad\qquad\quad (\mathit{nth}\ (n-1)\ (\mathit{tl}\ l))
\end{array}
$$

Combining lists

Append two lists.

$$
\begin{array}{l}
--\ \mathit{append} :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\
\textbf{def}\ \mathit{append}\ l1\ l2 = \textbf{case}\ l1\ \textbf{of} \\
\qquad\qquad\qquad\qquad \{1\} \qquad\quad \rightarrow l2; \\
\qquad\qquad\qquad\qquad \{2\}\ x\ xs \rightarrow \mathit{Cons}\ x\ (\mathit{append}\ xs\ l2)
\end{array}
$$

Deterministically merge lists (alternating takes).

$$
\begin{array}{l}
--\ \mathit{amerge} :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\
\textbf{def}\ \mathit{amerge}\ l1\ l2 = \textbf{case}\ l1\ \textbf{of} \\
\qquad\qquad\qquad\qquad \{1\} \qquad\quad \rightarrow l2; \\
\qquad\qquad\qquad\qquad \{2\}\ x\ xs \rightarrow \mathit{Cons}\ x\ (\mathit{amerge}\ l2\ xs)
\end{array}
$$

Merge sorted lists of numbers order preservingly.

$$
\begin{array}{l}
\text{-- } \mathit{smerge} :: [\mathit{Num}] \to [\mathit{Num}] \to [\mathit{Num}] \\
\textbf{def } \mathit{smerge}\ l1\ l2 = \textbf{case } l1 \textbf{ of} \\
\qquad \{1\} \qquad \to l2; \\
\qquad \{2\}\ x\ xs \to \textbf{case } l2 \textbf{ of} \\
\qquad\qquad \{1\} \qquad \to l1; \\
\qquad\qquad \{2\}\ y\ ys \to if\ (x \leqslant y) \\
\qquad\qquad\qquad (\mathit{Cons}\ x\ (\mathit{smerge}\ xs\ l2)) \\
\qquad\qquad\qquad (\mathit{Cons}\ y\ (\mathit{smerge}\ l1\ ys))
\end{array}
$$

Generating lists

Returns the list $[a, a+1, \ldots, b-1, b]$

$$
\begin{array}{l}
\text{-- } \mathit{enumFromTo} :: \mathit{Nat} \to \mathit{Nat} \to [\mathit{Nat}] \\
\textbf{def } \mathit{enumFromTo}\ a\ b = if\ (a > b) \\
\qquad \mathit{Nil} \\
\qquad (\mathit{Cons}\ a\ (\mathit{enumFromTo}\ (a+1)\ b))
\end{array}
$$

Returns the list $[a, a+1, \ldots]$

$$
\begin{array}{l}
\text{-- } \mathit{enumFrom} :: \mathit{Nat} \to [\mathit{Nat}] \\
\textbf{def } \mathit{enumFrom}\ a = \mathit{Cons}\ a\ (\mathit{enumFrom}\ (a+1))
\end{array}
$$

Sorting lists of numbers

Quicksort.

$$
\begin{array}{l}
\text{-- } \mathit{qsort} :: [\mathit{Num}] \to [\mathit{Num}] \\
\textbf{def } \mathit{qsort}\ l = \textbf{case } l \textbf{ of} \\
\qquad \{1\} \qquad \to \mathit{Nil}; \\
\qquad \{2\}\ x\ xs \to \textbf{let } ul = \mathit{filter}\ (gt\ x)\ l; \\
\qquad\qquad\qquad e \ = \mathit{filter}\ (eq\ x)\ l; \\
\qquad\qquad\qquad ug = \mathit{filter}\ (lt\ x)\ l \\
\qquad\qquad \textbf{in let } sl = \mathit{qsort}\ ul; \\
\qquad\qquad\qquad sg = \mathit{qsort}\ ug \\
\qquad\qquad \textbf{in } \mathit{concat}\ (\mathit{Cons}\ sl\ (\mathit{Cons}\ e\ (\mathit{Cons}\ sg\ \mathit{Nil})))
\end{array}
$$

Miscellaneous

Reverse a list.

$$
\begin{array}{l}
\text{-- } \mathit{reverse} :: [\alpha] \to [\alpha] \\
\textbf{def } \mathit{reverse}\ l = \textbf{case } l \textbf{ of} \\
\qquad \{1\} \qquad \to \mathit{Nil}; \\
\qquad \{2\}\ x\ xs \to \mathit{append}\ (\mathit{reverse}\ xs)\ (\mathit{Cons}\ x\ \mathit{Nil})
\end{array}
$$

Length of a list.

$-- \; length :: [\alpha] \to Nat$

$\mathbf{def}\ length\ l = \mathbf{case}\ l\ \mathbf{of}$

$\quad \{1\} \qquad\quad \to 0;$

$\quad \{2\}\ x\ xs \to 1 + length\ xs$

A.2 PreludeCoord.fatom

```
PreludeCoord.fatom

Fatom Prelude of coordination abstractions.

This file is part of the diploma thesis
"An Abstract Machine for a Concurrent (and Parallel)
Constraint Functional Programming Language"
by Florian Lorenzen <florenz@cs.tu-berlin.de>,
written in the Compiler Construction and Programming Languages group
of Prof. Dr. Peter Pepper at Berlin Technical University,
advised by Dr. Petra Hofstedt and Martin Grabmueller.
```

Lift a constant to a process.

$$
\begin{array}{l}
--\ \mathit{lift} :: \alpha \to \alpha \to C \\
\textbf{def}\ \mathit{lift}\ r\ e = r =:= e
\end{array}
$$

Non-deterministic merging of two lists.

$$
\begin{array}{l}
--\ \mathit{merge} :: [\alpha] \to [\alpha] \to [\alpha] \to C \\
\textbf{def}\ \mathit{merge}\ \mathit{l1}\ \mathit{l2}\ r = \begin{array}[t]{l}
\ \ \mathit{pack}\ \mathit{l1}\ 1\ 0\ \&\\
\ \ \mathit{pack}\ \mathit{l2}\ 1\ 0 \Rightarrow r =:= \mathit{Nil}\\
\mid \mathit{pack}\ \mathit{l1}\ 2\ 2 \Rightarrow \textbf{with}\ \mathit{rs}\ \mathit{l1}'\\
\qquad\qquad\qquad \textbf{in}\ r =:= \mathit{Cons}\ (\mathit{hd}\ \mathit{l1})\ \mathit{rs}\ \&\\
\qquad\qquad\qquad\quad \mathit{l1}' =:= \mathit{tl}\ \mathit{l1}\ \&\\
\qquad\qquad\qquad\quad \mathit{merge}\ \mathit{l1}'\ \mathit{l2}\ \mathit{rs}\\
\mid \mathit{pack}\ \mathit{l2}\ 2\ 2 \Rightarrow \textbf{with}\ \mathit{rs}\ \mathit{l2}'\\
\qquad\qquad\qquad \textbf{in}\ r =:= \mathit{Cons}\ (\mathit{hd}\ \mathit{l2})\ \mathit{rs}\ \&\\
\qquad\qquad\qquad\quad \mathit{l2}' =:= \mathit{tl}\ \mathit{l2}\ \&\\
\qquad\qquad\qquad\quad \mathit{merge}\ \mathit{l1}\ \mathit{l2}'\ \mathit{rs}
\end{array}
\end{array}
$$

Concurrent map: for each element x of l a process is spawned to compute $f\ x$.

$$
\begin{array}{l}
--\ \mathit{farm} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \to C \\
\textbf{def}\ \mathit{farm}\ f\ l\ r = \textbf{case}\ l\ \textbf{of}\\
\qquad\qquad\qquad \{1\}\qquad \to r =:= \mathit{Nil};\\
\qquad\qquad\qquad \{2\}\ x\ \mathit{xs} \to \textbf{with}\ \mathit{rs}\ \textbf{in}\ r =:= \mathit{Cons}\ (f\ x)\ \mathit{rs}\ \&\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{farm}\ f\ \mathit{xs}\ \mathit{rs}
\end{array}
$$

This version of *farm* spawns n processes each computing a part of $\mathit{map}\ f\ l$.

$$
\begin{array}{l}
--\ \mathit{pfarm} :: \mathit{Nat} \to (\alpha \to \beta) \to [\alpha] \to [\beta] \to C \\
\textbf{def}\ \mathit{nfarm}\ n\ f\ l\ r = \textbf{with}\ \mathit{rs}\\
\qquad\qquad \textbf{in let}\ \mathit{parts} = \mathit{partition}\ n\ l;\\
\qquad\qquad\qquad\quad \mathit{pf} \quad = \mathit{map}\ f\\
\qquad\qquad\quad \textbf{in}\ \mathit{farm}\ \mathit{pf}\ \mathit{parts}\ \mathit{rs}\ \&\ r =:= \mathit{concat}\ \mathit{rs}
\end{array}
$$

pfarm spawns one process for each processing element.

$$
\begin{array}{l}
--\ \mathit{pfarm} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \to C \\
\textbf{def}\ \mathit{pfarm}\ f\ l\ r = \mathit{nfarm}\ \mathit{noPE}\ f\ l\ r
\end{array}
$$

Appendix B

ATAF instruction set in alphabetical order

B.1 Machine instructions

ALLOC $n \equiv$

  $as :=$ allocateVars n
  FOREACH a IN as DO
    push $\langle$PTR $\mid a\rangle$
  END FOREACH
  inc IP

ADD $\equiv$

  arith $+$
  inc IP

AND $\equiv$

  bool $\wedge$
  inc IP

CASEJUMP $t_1\ a_1 \ldots t_n\ a_n \equiv$

  $\langle$PTR $\mid a\rangle :=$ top
  $[w] :=$ suspend $[a]$
  $\langle$PCK $\mid t\ \cdot\rangle := w$
  FOR $i := 1$ TO n DO
    IF $t = t_i$ THEN
      JUMP a_i
    END IF
  END FOR

DIV $\equiv$

  arith $/$
  inc IP

EQ $\equiv$

compare $=$

inc IP

EVAL $\equiv$

$ptr :=$ pop

push $\langle$PTR $\mid$ IP$\rangle$

push ptr

UNWIND

GE $\equiv$

compare $\geq$

inc IP

GT $\equiv$

compare $>$

inc IP

ISBOUND $n\ a \equiv$

IF lookup $n \neq \langle$NIL$\rangle$ THEN

inc IP

ELSE

set IP (IP $+ a$)

END IF

ISPACK $n\ t\ ar\ a \equiv$

IF lookup $n = \langle$PCK $\mid t\ as\rangle \wedge$

$|as| = ar$ THEN

inc IP

ELSE

set IP (IP $+ a$)

END IF

ISUNBOUND $ptr\ a \equiv$

IF lookup $n = \langle$NIL$\rangle$ THEN

inc IP

ELSE

set IP (IP $+ a$)

END IF

LE $\equiv$

compare $\leq$

inc IP

LOCK $\equiv$

$\quad (\!| i, p |\!) :=$ `Run`

$\quad$ IF `TF` THEN

$\quad\quad$ `TF` $:= \mathit{false}$

$\quad\quad$ `Run` $:= \langle\!| i, p |\!\rangle$

$\quad\quad$ inc `IP`

$\quad$ ELSE

$\quad\quad$ enqueue $\langle\!| i, p |\!\rangle$ `Blocked`

$\quad\quad$ inc `IP`

$\quad\quad$ `Run` := dequeue `Ready`

$\quad$ END IF

LT $\equiv$

$\quad$ compare $<$

$\quad$ inc `IP`

MKAP $\equiv$

$\quad a :=$ allocate

$\quad \langle \mathsf{PTR} \mid a_1 \rangle :=$ pop

$\quad \langle \mathsf{PTR} \mid a_2 \rangle :=$ pop

$\quad$ `HEAP`$[a] := \langle \mathsf{APP} \mid a_1 \ a_2 \rangle$

$\quad$ push $\langle \mathsf{PTR} \mid a \rangle$

$\quad$ inc `IP`

MUL $\equiv$

$\quad$ arith $*$

$\quad$ inc `IP`

NE $\equiv$

$\quad$ compare $\neq$

$\quad$ inc `IP`

NEG $\equiv$

$\quad a :=$ allocate

$\quad a_1 :=$ top

$\quad [v] :=$ suspend $[a_1]$

$\quad$ pop

$\quad$ IF $v = \langle \mathsf{NAT} \mid x \rangle$ THEN

$\quad\quad$ `HEAP`$[a] := \langle \mathsf{NAT} \mid 0 \rangle$

$\quad$ ELSE

$\quad\quad$ `HEAP`$[a] := \langle \mathsf{FLT} \mid -x \rangle$

$\quad$ END IF

$\quad$ push $\langle \mathsf{PTR} \mid a \rangle$

$\quad$ inc `IP`

NOPE $\equiv$

$a :=$ allocate

HEAP$[a] := \langle$NAT $\mid noPE\rangle$

push $\langle$PTR $\mid a\rangle$

inc IP

NOT $\equiv$

$a :=$ allocate

$a_1 :=$ top

$[b] :=$ suspend $[a_1]$

pop

IF $b = \langle$PCK $\mid 1\ []\rangle$ THEN

HEAP$[a] := \langle$PCK $\mid 2\ []\rangle$

ELSE

HEAP$[a] := \langle$PCK $\mid 1\ []\rangle$

END IF

push $\langle$PTR $\mid a\rangle$

inc IP

OR $\equiv$

bool $\vee$

inc IP

PACK $t\ ar \equiv$

$a :=$ allocate

$as := []$

FOR; $i := 1$ TO ar DO

$\langle$PTR $\mid a'\rangle :=$ pop

$as := as +\!\!+ [a']$

END FOR

HEAP$[a] := \langle$PCK $\mid t\ as\rangle$

push $\langle$PTR $\mid a\rangle$

inc IP

POP $n \equiv$

SP $:=$ SP $+ n$

inc IP

PUSHFLOAT $x \equiv$

$a :=$ allocate

HEAP$[a] := \langle$FLT $\mid x\rangle$

push $\langle$PTR $\mid a\rangle$

inc IP

PUSHFUN $a\ ar \equiv$

  $a' :=$ allocate

  $\texttt{HEAP}[a'] := \langle$FUN $\mid a\ ar\rangle$

  push $\langle$PTR $\mid a'\rangle$

  inc $\texttt{IP}$

PUSHNAT $n \equiv$

  $a :=$ allocate

  $\texttt{HEAP}[a] := \langle$NAT $\mid n\rangle$

  push $\langle$PTR $\mid a\rangle$

  inc $\texttt{IP}$

PUSHVAR $n \equiv$

  push (nth n)

  inc $\texttt{IP}$

SLIDE $n \equiv$

  $\texttt{STACK}[\texttt{SP} + n + 1] :=$ top

  $\texttt{SP} := \texttt{SP} + n$

  inc $\texttt{IP}$

SPAWN $n\ a \equiv$

  $pid :=$ newProc

  pushProc pid $(\texttt{IP} + a)$ $[\texttt{STACK}[\texttt{SP} + 1], \ldots, \texttt{STACK}[\texttt{SP} + n]]$

  inc $\texttt{IP}$

SPLIT $n \equiv$

  $\langle$PTR $\mid a\rangle :=$ top

  $[w] :=$ suspend $[a]$

  pop

  $\langle$PCK $\mid \cdot\ [a_1, \ldots, a_n]\rangle := w$

  FOREACH a' IN $[a_n, \ldots, a_1]$ DO

    push $\langle$PTR $\mid a'\rangle$

  END FOREACH

  inc $\texttt{IP}$

SUB $\equiv$

  arith $-$

  inc $\texttt{IP}$

SUSPEND $n_1 \ldots n_m \equiv$

  unlock

  FOREACH n IN $[n_1, \ldots, n_m]$ DO

    $\langle$PTR $\mid a\rangle :=$ nth n

    enqueue Run Suspend$[a]$

  END FOREACH

  $\texttt{IP} := \texttt{IP} + 1$

  Run := dequeue Ready

$\mathsf{TELL}\ n \equiv$
    $\langle \mathsf{PTR} \mid a \rangle := \mathsf{nth}\ n$
    $\langle \mathsf{PTR} \mid ptr \rangle := \mathsf{top}$
    $w := \mathsf{lookup}\ ptr$
    IF $\mathtt{STORE}[a] = \langle \mathsf{NIL} \rangle$ THEN
        IF $w = \langle \mathsf{NAT} \mid \cdot \rangle \ \vee\ w = \langle \mathsf{FLT} \mid \cdot \rangle$ THEN $\mathtt{STORE}[a] := w$
        ELSE
            $\langle \mathsf{PCK} \mid t\ as \rangle := w$
            $vs := \mathsf{allocateVars}\ |as|$
            $\mathtt{STORE}[a] := \langle \mathsf{PCK} \mid t\ vs \rangle$
            FOREACH (a', v') IN (as, vs) DO
                push $\langle \mathsf{PTR} \mid a' \rangle$
                push $\langle \mathsf{PTR} \mid v' \rangle$
            END FOREACH
        END IF
        wakeup a
    ELSE IF
        $\mathtt{STORE}[a] \neq w$ THEN fail
    END IF
    inc IP

$\mathsf{TOFLOAT} \equiv$
    $a := \mathsf{allocate}$
    $a_1 := \mathsf{top}$
    $[w] := \mathsf{suspend}\ [a_1]$
    $\langle \cdot \mid x \rangle := w$
    $\mathtt{HEAP}[a] := \langle \mathsf{FLT} \mid x \rangle$
    push $\langle \mathsf{PTR} \mid a \rangle$
    inc IP

$\mathsf{TONAT} \equiv$
    $a := \mathsf{allocate}$
    $a_1 := \mathsf{top}$
    $[w] := \mathsf{suspend}\ [a_1]$
    $\langle \cdot \mid x \rangle := w$
    $\mathtt{HEAP}[a] := \langle \mathsf{NAT} \mid round(x) \rangle$
    push $\langle \mathsf{PTR} \mid a \rangle$
    inc IP

$\mathsf{UNLOCK} \equiv$
    unlock
    inc IP

```
UNWIND ≡
    IF SP = SB THEN
        terminate
    END IF
    ⟨PTR | a⟩ := top
    IF a < TB THEN
        w := HEAP[a]
        IF w = ⟨FUN | a' ar⟩ THEN
            rearrange ar
            JUMP (a − IP)
        ELSE IF ground w THEN
            undump
        ELSE IF w = ⟨APP | a1 ·⟩ THEN
            push ⟨PTR | a1⟩
            UNWIND
        ELSE IF w = ⟨PTR | ·⟩ THEN
            pop
            push w
            UNWIND
        END IF
    ELSE
        undump
    END IF
```

```
UPDATE n ≡
    ⟨PTR | a⟩ := nth n
    HEAP[a] := pop
    inc IP
```

B.2 Internal operations

```
allocate ≡
    inc HP
    RETURN HP − 1
```

```
allocateVars n ≡
    TP := TP + n
    RETURN [TP − n, TP − n + 1, . . . , TP − 1]
```

arith $\oplus \equiv$

$a :=$ allocate
$a_1 :=$ nth 0
$a_2 :=$ nth 1
$[v_1, v_2] :=$ suspend $[a_1, a_2]$
pop
pop
IF $v_1 = \langle \mathsf{NAT} \mid n_1 \rangle \ \wedge \ v_2 = \langle \mathsf{NAT} \mid n_2 \rangle$ THEN
  $\mathtt{HEAP}[a] := \langle \mathsf{NAT} \mid n_1 \oplus n_2 \rangle$
ELSE
  $\langle \cdot \mid x_1 \rangle := v_1$
  $\langle \cdot \mid x_2 \rangle := v_2$
  $\mathtt{HEAP}[a] := \langle \mathsf{FLT} \mid x_1 \oplus x_2 \rangle$
END IF
push $\langle \mathsf{PTR} \mid a \rangle$

---

bool $\oplus \equiv$

$a :=$ allocate
$a_1 :=$ top
$a_2 :=$ top
$[b_1, b_2] :=$ suspend $[a_1, a_2]$
pop
pop
IF $b_1 = \langle \mathsf{PCK} \mid 1\ [] \rangle \ \wedge \ b_2 = \langle \mathsf{PCK} \mid 1\ [] \rangle$ THEN
  $r := \mathit{true} \oplus \mathit{true}$
ELSE IF $b_1 = \langle \mathsf{PCK} \mid 1\ [] \rangle \ \wedge \ b_2 = \langle \mathsf{PCK} \mid 2\ [] \rangle$ THEN
  $r := \mathit{true} \oplus \mathit{false}$
ELSE IF $b_1 = \langle \mathsf{PCK} \mid 2\ [] \rangle \ \wedge \ b_2 = \langle \mathsf{PCK} \mid 1\ [] \rangle$ THEN
  $r := \mathit{false} \oplus \mathit{true}$
ELSE IF $b_1 = \langle \mathsf{PCK} \mid 2\ [] \rangle \ \wedge \ b_2 = \langle \mathsf{PCK} \mid 2\ [] \rangle$ THEN
  $r := \mathit{false} \oplus \mathit{false}$
END IF
IF r THEN
  $\mathtt{HEAP}[a] := \langle \mathsf{PCK} \mid 1\ [] \rangle$
ELSE
  $\mathtt{HEAP}[a] := \langle \mathsf{PCK} \mid 2\ [] \rangle$
END IF
push $\langle \mathsf{PTR} \mid a \rangle$

---

compare $\oplus \equiv$
$a :=$ allocate
$a_1 :=$ top
$a_2 :=$ top
$[w_1, w_2] :=$ suspend $[a_1, a_2]$
pop
pop
$\langle \cdot \mid x_1 \rangle := w_1$
$\langle \cdot \mid x_2 \rangle := w_2$
IF $x_1 \oplus x_2$ THEN
$\mathtt{HEAP}[a] := \langle \mathsf{PCK} \mid 1\ [] \rangle$
ELSE
$\mathtt{HEAP}[a] := \langle \mathsf{PCK} \mid 2\ [] \rangle$
END IF
push $\langle \mathsf{PTR} \mid a \rangle$

---

copy *as copied* $\equiv$
FOREACH a IN *as* DO
IF $copied(a) = undefined$ THEN
$\mathtt{HEAP}_{i_2 p_2}[\mathtt{HP}_{i_2 p_2}] := \mathtt{HEAP}_{k_1 l_1}[a]$
$copied := copied.[a \mapsto \mathtt{HP}_{i_2 p_2}]$
inc $\mathtt{HP}_{i_2 p_2}$
END IF
END FOREACH
RETURN *copied*

---

dequeue $q \equiv$
IF $q = a : q'$ THEN
$q := q'$
RETURN a
ELSE
RETURN $(\!|\cdot|\!)$
END IF

---

enqueue $a\ q \equiv$
$q := q +\!\!+ [a]$

---

ground $v \equiv$
RETURN $v = \langle \mathsf{PCK} \mid \cdot\ \cdot \rangle \vee$
$v = \langle \mathsf{NAT} \mid \cdot \rangle \vee$
$v = \langle \mathsf{FLT} \mid \cdot \rangle$

---

isEmpty $q \equiv$
IF $q = []$ THEN
RETURN false
ELSE
RETURN true
END IF

---

lookup $n \equiv$
$\langle \mathsf{PTR} \mid ptr \rangle :=$ nth n
$w :=$ IF $ptr <$ TB THEN
RETURN HEAP$[ptr]$
ELSE
RETURN STORE$[ptr]$
END IF

---

pop $\equiv$
inc SP
RETURN STACK[SP]

---

push $v \equiv$
STACK[SP] $:= v$
dec SP

---

pushProc $(\!| i_2, p_2 |\!)$ a $as \equiv$
$\mathit{copied} := \emptyset$
FOREACH a IN as DO
IF $a <$ TB THEN
$\mathit{copied} :=$ copy $[a]$ copied
$\mathtt{STACK}_{i_2 p_2}[\mathtt{SP}_{i_2 p_2}] := \langle \mathsf{PTR} \mid \mathit{copied}(a) \rangle$
ELSE
$\mathit{copied} := \mathit{copied}.[a \mapsto a]$
$\mathtt{STACK}_{i_2 p_2}[\mathtt{SP}_{i_2 p_2}] := \langle \mathsf{PTR} \mid a \rangle$
END IF
inc $\mathtt{SP}_{i_2 p_2}$
END FOREACH

$ptr := \texttt{HB}_{i_2 p_2}$
WHILE $ptr < \texttt{HP}_{i_2 p_2}$ DO
  IF $\texttt{HEAP}_{i_2 p_2}[ptr] = \langle \textsf{PTR} \mid a \rangle$ THEN
    $copied := \textsf{copy}\ [a]\ copied$
    $\texttt{HEAP}_{i_2 p_2}[ptr] := \langle \textsf{PTR} \mid copied(a) \rangle$
  ELSE IF $\texttt{HEAP}_{i_2 p_2}[ptr] = \langle \textsf{APP} \mid a_1\ a_2 \rangle$ THEN
    $copied := \textsf{copy}\ [a_1, a_2]\ copied$
    $\texttt{HEAP}_{i_2 p_2}[ptr] := \langle \textsf{APP} \mid copied(a_1)\ copied(a_2) \rangle$
  ELSE IF $\texttt{HEAP}_{i_2 p_2}[ptr] = \langle \textsf{PCK} \mid t\ [a_1, \ldots, a_n] \rangle$ THEN
    $copied := \textsf{copy}\ [a_1, \ldots, a_n]\ copied$
    $\texttt{HEAP}_{i_2 p_2}[ptr] := \langle \textsf{PTR} \mid t\ [copied(a_1), \ldots, copied(a_n)] \rangle$
  END IF
  $\textsf{inc}\ ptr$
END WHILE
$\textsf{enqueue}\ (\!|\, i_2, p_2 \,|\!)\ \texttt{Ready}_{i_2}$
$\texttt{IP}_{i_2 p_2} := a$

$\textsf{queueJump}\ a\ q \equiv$
  $q := a : q$

$\textsf{rearrange}\ ar \equiv$
  FOR $i := 1$ TO ar DO
    $w := \texttt{HEAP}[\texttt{SP} + i + 1]$
    $\langle \textsf{APP} \mid \cdot\ a_2 \rangle := w$
    $\texttt{STACK}[\texttt{SP} + i] := a_2$
  END FOR

$\textsf{set}\ \texttt{IP}\ a \equiv$
    $\texttt{IP} := a$
  IF $\texttt{Run} \neq (\!|\, i, p \,|\!) \wedge \texttt{IC} \geq NI$ THEN
    $next := \textsf{dequeue}\ \texttt{Ready}$
    IF $next \neq (\!|\cdot|\!)$ THEN
      $\textsf{enqueue}\ \texttt{Run}\ \texttt{Ready}$
      $\texttt{Run} := next$
    END IF
    $\texttt{IC} := 0$
  ELSE
    $\textsf{inc}\ \texttt{IC}$
  END IF

suspend $as \equiv$

$\quad unbound := false$
$\quad ws := []$
$\quad$ FOR a IN as DO
$\quad\quad w :=$ lookup a
$\quad\quad ws := ws +\!\!+ [w]$
$\quad\quad$ IF $w = \langle \mathsf{NIL} \rangle$ THEN
$\quad\quad\quad unbound := true$
$\quad\quad\quad$ enqueue `Run` `Suspend`$[a]$
$\quad\quad$ END IF
$\quad$ END FOR
$\quad$ IF $unbound$ THEN
$\quad\quad$ `Run` := dequeue `Ready`
$\quad$ ELSE
$\quad\quad$ RETURN ws
$\quad$ END IF

---

undump $\equiv$

$\quad v :=$ pop
$\quad \langle \mathsf{PTR} \mid ip \rangle :=$ pop
$\quad$ push v
$\quad$ set IP $(ip + 1)$

---

unlock $\equiv$

$\quad \langle\!| i, p |\!\rangle :=$ `Run`
$\quad$ `Run` $:= (\!| i, p |\!)$
$\quad$ IF isEmpty `Blocked` THEN
$\quad\quad$ TF $:= true$
$\quad$ ELSE
$\quad\quad \langle\!| i, p |\!\rangle :=$ dequeue `Blocked`
$\quad\quad$ queueJump $\langle\!| i, p |\!\rangle$ `Ready`$_i$
$\quad$ END IF

```
wakeup a ≡
    FOR k := TB, ..., a − 1, a + 1, ..., TP − 1 DO
        FOREACH ⦇i, p⦈ IN Suspend[k] DO
            q := []
            dequeue := false
            FOREACH ⦇i′, p′⦈ IN Suspend[a] DO
                IF ⦇i, p⦈ = ⦇i′, p′⦈ THEN
                    dequeue := true
                END IF
                IF ¬dequeue THEN
                    q := q ++ [⦇i, p⦈]
                END IF
            END FOREACH
            Suspend[k] := q
        END FOREACH
    END FOR
    WHILE ¬(isEmpty Suspend[a]) DO
        ⦇i, p⦈ := dequeue Suspend[a]
        enqueue ⦇i, p⦈ Ready_i
    END WHILE
```

Appendix C

Inference algorithm for supercombinator types

Chapter 3.4 describes how to compile supercombinators under the assumption that their type is known, i. e. whether they are a constraint or a functional abstraction. This appendix briefly outlines an inference algorithm for the **C**, **F** annotations. The correctness of the algorithm is not proved but it is fairly straightforward and during the implementation of `fc` no (correct) FATOM program appeared on which the algorithm failed. There is one restriction, though. The types of arguments to supercombinators and all locally bound variables of **let**- oder **case**-expressions are always assumed to be of type **F**. As a consequence, functional abstractions may be passed around but not constraint abstractions. However, this restriction could be overcome by a proper type system and type annotations in the source code (cf. Section 2.3.2).
The input for the inference function *infer* is a program P, i. e. a sequence of supercombinator definitions. The output is a mapping from supercombinator names to their type.
The algorithm is a fixed point iteration, which is driven by the function

$$\mathit{fix}(f,x) = \begin{cases} x & \text{if } f(x) = x \\ \mathit{fix}(f, f(x)) & \text{otherwise.} \end{cases}$$

The inference function is just the fixed point of another function: *scTypes*, with $v_1, \ldots, v_n, \mathit{main}$ being the names of the supercombinators of the program P.

$$\mathit{infer}\ P = \mathit{fix}(\mathit{scTypes}(P), [v_1 \mapsto \Box, \ldots, v_n \mapsto \Box, \mathit{main} \mapsto \mathbf{C}])$$

The function *scTypes* iterates over all supercombinators and tries to infer their type. The second argument to *scTypes* is the current approximation to the complete type map. At the beginning, nothing is known (all entries are $\Box$) except that *main* is a constraint abstraction.

$$\begin{aligned} \mathit{scTypes}(P)(\Gamma) &= \Gamma' \quad \text{with } \langle \Gamma', \cdot \rangle = \mathit{mapAccumL}(\Gamma, \mathit{scType}, P) \\ \mathit{scType}(\Gamma, sc) &= \langle \Gamma.[v \mapsto \mathit{exprType}(\Gamma, e)], sc \rangle \\ &\qquad \text{with } sc = (\mathbf{def}\ v\ v_1 \ldots v_n = e) \end{aligned}$$

The function *exprType* now does the real work. It is defined along the syntax of FATOM.

$$
\begin{aligned}
exprType(\cdot, GExpr) &= \mathsf{C}\\
exprType(\cdot, \textbf{with}\ vs\ \textbf{in}\ e) &= \mathsf{C}\\
exprType(\cdot, e_1\ \&\ \ldots\ \&\ e_n) &= \mathsf{C}\\
exprType(\cdot, v =:= e) &= \mathsf{C}\\
exprType(\Gamma, e_1\ e_2) &= exprType(\Gamma, e_1)\\
exprType(\cdot, Const) &= \mathsf{F}\\
exprType(\cdot, Pack\{t,a\}) &= \mathsf{F}\\
exprType(\Gamma, v) &= \Gamma(v)\\
exprType\left(\Gamma, \begin{array}{l} \textbf{let}\ v_1 = e_1\,;\\ \quad\vdots\\ \quad v_n = e_n\\ \textbf{in}\ e\end{array}\right) &= exprType\left(\Gamma.\begin{bmatrix} v_1 \mapsto \mathsf{F},\\ \ldots,\\ v_n \mapsto \mathsf{F}\end{bmatrix}, e\right)\\
exprType\left(\Gamma, \begin{array}{l} \textbf{case}\ e\ \textbf{of}\\ \quad \{t\}\ v_1 \ldots v_n \to b\,;\\ \quad\vdots \end{array}\right) &= exprType\left(\Gamma.\begin{bmatrix} v_1 \mapsto \mathsf{F},\\ \ldots,\\ v_n \mapsto \mathsf{F}\end{bmatrix}, b\right)
\end{aligned}
$$

Concerning **case**-expressions only one branch is considered. As the types of the branches have to be equal, this is sufficient. But this may lead to problems if the program is incorrect and one branch has a C type and another one F.
The iteration function *mapAccumL* has the following definition:

$$
\begin{aligned}
mapAccumL(\cdot,\cdot,[]) &= []\\
mapAccumL(a,f,x:xs) &= \langle a'', y:ys\rangle\\
&\qquad \text{with}\quad \langle a', y\rangle = f(a,x)\\
&\qquad\qquad\ \ \langle a'', ys\rangle = mapAccumL(f, a', xs)
\end{aligned}
$$

Appendix D

Manual of `fc` and `ataf`

D.1 Installation

This section briefly outlines how to get and install the FATOM compiler `fc` and the ATAF abstract machine interpreter `ataf`.

D.1.1 Availabilty

The current version of the `atafc` bundle is v0.1. It can be downloaded from `http://user.cs.tu-berlin.de/~florenz/atafc/atafc-0.1.tar.gz`.

D.1.2 Requirements

To build `atafc`, the Glasgow Haskell Compiler v6.4.2 or newer is required. GHC is available from `http://www.haskell.org/ghc/`.
In addition to GHC, an MPI implementation is required. The package has been tested with MPICH v1.2.7 and MP-MPICH v1.3.0. Any other implementation should also do, as long as the `mpicc` program supports the `-compile_info` flag.
To compile the example programs (and run the testsuite) lhs2TEX v1.11 or later is required, which is available from `http://www.informatik.uni-bonn.de/~loeh/lhs2tex/`.

D.1.3 Building and installing

Download the file `atafc-0.1.tar.gz` and extract all files:

```
$ gunzip -c atafc-0.1.tar.gz | tar xf -
```

Change to the `atafc-0.1` directory and configure the package:

```
$ cd atafc-0.1
$ ./setup configure
```

For `./setup configure` to work, `runghc` has to be in the search `$PATH`. Otherwise, it is also possible to start the `Setup.hs` scripts directly:

```
$ /path/to/runghc fc/Setup.hs configure
$ /path/to/runghc ataf/Setup.hs configure
```

The `setup` scripts takes some options similar to those of AUTOCONF generated `configure` scripts. The `--help` options prints a list of them. The most important one is probably `--prefix` which sets the installation prefix for the binaries. To install the binaries in e. g. `/opt/atafc/bin`, the following command-line can be used:

```
$ ./setup configure --prefix=/opt/atafc
```

The `bin` is appended automatically. But before it possible to install the binaries they have to be built:

```
$ ./setup build
```

To install the binaries, type

```
$ ./setup install
```

It is not necessary to install the binaries on the system. They can be called directly from the source directory where they are created in `ataf/dist/build/ataf` and `fc/dist/build/fc`.

D.1.4 Running the testsuite

`atafc` has two testsuites. One is very small and tests only some internal functions. It can be started by

```
$ ./setup test
```

in the source directory. Its output should look like this:

```
Cases: 9  Tried: 9  Errors: 0  Failures: 0
Cases: 27  Tried: 27  Errors: 0  Failures: 0
```

The big testsuite compiles and runs FATOM programs and checks the computed results. This testuite needs lhs2TEX to preprocess the FATOM testprograms. This testsuite can be run using the newly created binaries with the following command:

```
$ test/test.sh --nobuild
```

The `--nobuild` option tells the script to use the binaries from the `dist` directory. The last line of output should indicate 0 failures:

```
TESTRESULT: 0 out of 30 tests failed
```

D.2 Compiling FATOM programs

FATOM programs are compiled into ATAF instructions by `fc`. The general layout of an `fc` command-like is this:

`fc` *options input-files...*

The compiler takes one or more input files, which are simply appended and then compiled as if the source code was saved in a single file. This is very handy to include the FATOM Prelude in other programs:

`$ fc` *options* `PreludeFun.fatom` *some-program*

D.2.1 Command-line options

The compiler understands the following options:

Short	Long option	Description
`-h`	`--help`	Show a brief help.
`-o`	`--output`	Name of the machine code file.
`-d`	`--debug`	Show debugging output.
`-p`	`--preprocessor`	Command to start as a preprocessor.
`-V`	`--version`	Show version.
`-n`	`--no-main`	Compile without *main* combinator.
	`--use-copy`	Use `COPY` instructions.
	`--no-spawn`	Do not spawn processes for *Pack*s in tells.
	`--spawn`	Spawn processes for *Pack*s in tells.
	`--opt-conj`	Use optimised compilation for conjunctions.

The meaning of the options is described in the following.

`-h`, `--help`
: `fc` just prints an option summary like the table above and exits.

`-o` *file*, `--output=`*file*
: The `ataf` machine code is written to *file*. If this option is not given, output is written to the file `a.out`.

`-d`, `--debug`
: This option is primarily for debugging the compiler itself. If it is given on the command line `fc` prints the following items on standard output:

- internal options table,
- the parsed abstract syntax in an s-expression like format,
- the result of supercombinator type inference, and
- the generated code.

`-p` *[command]*, `--preprocessor=`*[command]*
: This option switches on preprocessing of input files. All source files are piped through *command*, which should expect data in standard input and return the pre-processed stream via standard output. If *command* is omitted the default `lhs2TeX --code` is used, which is suitable for literate FATOM programs. It is customary to name literate FATOM programs with a `.fatom` suffix and plain FATOM source files with `.ftm`. *command* is executed via a `system(3)` call, i. e. it is possible to give command-line options, e. g.: `-p'lhs2TeX --code --path=:../share'`.

`-V`, `--version`
: Print the version number and exit.

`-n`, `--no-main`

Compile a program without a *main* function. Of course, this program cannot be executed, so this option is to check if a given source code is compilable. This is useful when writing programs that span over several files to check files separately.

`--use-copy`
: Use the `COPY` instruction to compile tell-equalities. This instruction is undocumented. Its purpose was to speed up the execution of tell equalities, but it did not work out immediately and thus was not developed further. *Do not use it.*

`--no-spawn`
: Use the optimised ψ' compilation scheme for tell-equalities. This scheme does not spawn processes for sub-terms of constructors but evaluates them in the same process.

`--spawn`
: The opposite of `--no-spawn`. This is the default.

`--opt-conj`
: Use the optimised $\mathcal{C}'$ scheme to compile conjunctions. This scheme spawns one process less than the default compilation.

D.3 Running FATOM programs

Compiled FATOM programs, i. e. ATAF instruction files, are interpreted by `ataf`.

D.3.1 Starting `ataf`

Depending in the MPI implementation and the machine running on, different startup commands have to be used. In the simplest case, which is e. g. possible with MPICH on a single processor machine, `ataf` can be called like this to execute *program*:

`ataf` *options program*

Running on several processors is often achieved with the `mpirun` command:

`mpirun -n` *noPE* `ataf` *options program*

In many cases, a queueing or cluster management system has to be used to start parallel jobs. For example, in CCS, `ataf` is started with this command-line:

`ccsalloc -s asap -n` *noPE* `mpich -- ataf` *options program*

D.3.2 Command-line options

The machine interpreter takes the following options:

Short	**Long option**	**Description**
`-d`	`--debug`	Debug mode.
`-h`	`--help`	Print short option table.
`-V`	`--version`	Print version number.
`-p`	`--profiling`	Switch on profiling.
`-f`	`--fast-prop`	Fast *Pack* propagation.
`-s`	`--store-island`	The store has a processor for itself.
`-N`	`--no-stats`	No execution statistics.
`-t`	`--time-slice`	Length of time slice.
`-H`	`--heap-size`	Size of heap in words.
`-S`	`--stack-size`	Size of stack in words.
`-T`	`--store-size`	Size of store in words.
	`--no-store-gc`	Switch off garbage collection in store.

`-d`, `--debug`
: This flag switches on debug mode. In this mode, lots of information is dumped to a file called `DEBUG.`*nnn* where *nnn* is the MPI rank number, i. e. every instance writes one file. The files are written in the current working directory of `ataf`. Note that some startup systems change the current working directory.
Several messages passed between the different threads are logged, as well as context switches. But the largest parts of the `DEBUG`-files is the machine state, which is dumped whenever an instruction is completed. So be warned, these files grow very large pretty soon. Be also aware of the fact that writing debug information slows down the machine by several orders of magnitudes. So it should only be used for short programs.

`-V`, `--version`
: Print version number and exit.

`-p`, `--profiling`
: Switch on profiling. At the moment, profiling only measures the total execution time. See also the next Section D.3.3.

`-f`, `--fast-prop`
: Use fast propagation for primitive sub-terms of constructors in tell equalities.

`-s`, `--store-island`
: Reserve one processing element to handle the store exclusively. This option has no effect when running on a single node and should only be used when running on more than four or six nodes.

`-N`, `--no-stats`
: Switch of statistics recording and printing. This suppresses statistical output at the end of the computation.

`-t` *number*, `--time-slice=`*number*
: Length of the time slice NI in number of instructions each process may execute before being preempted.

`-H` *size*, `--heap-size=`*size*

Size of heap in words. As a copying garbage collector is used a process can only use half of this amount at once. The default size is 16384 words.

`-S` *size*, `--stack-size=`*size*

Size of the stack in words. The default is 4096 words.

`-T` *size*, `--store-size=`*size*

Size of the store in words. The default is 8192 words.

`--no-store-gc`

This option disables the mark-and-scan garbage collector. When the store is full, execution is aborted, and the content of the store is printed. Sometimes, this is useful for debugging.

D.3.3 Output and result reporting

After a program has been interpreted `ataf` prints out the result, which is the first store word and all reachable words. They are printed in an s-expression like syntax, e. g.:

```
** RESULT (collected store)
**
** ({2}
**  1
**  ({2}
**   2
**   ({2}
**    5
**    ({2}
**     1
**     ({2}
**      13
**      ({1}))))))
```

The $\{i\}$ denote constructor tags, their sub-terms are parenthesised and indented.

If statistics are not switched off, the number of processes per instance, the number of garbage collections in the store, and the (total) number of store words used is also printed, e. g.:

```
** Scheduler statistics:
     # total processes: 19
    # processes on 000: 15
    # processes on 001: 4

** Store statistics:
    # GCs: 0, # store words: 18
```

If profiling is switched on (`-p`) option, another block containing the machine options and the total execution time is also printed, e. g.:

```
** Execution time measurement:
    Running on 1 processing elements
    Options =
      input:        fatom/Append.ataf
      profiling:    True
      fast prop:    False
      store island: False
      heap size:    16384
      stack size:   4096
      store size:   8192
      code base:    0
      time slice:   1
      statistics:   True
      debug:        False

    Execution time: 3 secs
```

D.3.4 Memory usage

`ataf` uses quite a lot of memory, roughly 400 MB with the default settings. This is to reduce the garbage collection time in the GHC runtime system and speeds up the computation several times. However, 400 MB of memory is too much for smaller machines and causes thrashing. The amount of memory used can be tweaked with the `-A` (set allocation area size) flag of the GHC runtime system. To reduce the amount of memory to 32 MB, say, the allocation area has to be half of it, i. e. 16 MB:

`ataf` *arguments* `+RTS -A16m -RTS`

Enclose the `-A` flag in `+RTS`, `-RTS`, to separate it from `ataf` options (and options which may be added by the MPI startup routine).

D.4 Documentation

Apart from this short manual there is no proper documentation. However, in `ataf/doc` and `fc/doc` a beautified version of the source code can be found that contains quite some comments.

Appendix E

The names FATOM and ATAF

The names FATOM and ATAF appear in the novel "Die $13^1/_2$ Leben des Käpt'n Blaubär" by Walter Moers and they are explained in the "Lexikon der erklärungsbedürftigen Wunder, Daseinsformen und Phänomene Zamoniens und Umgebung" by Prof. Dr. Abdul Nachtigaller as follows:

> **Fatom, das** Transluzide Daseinsform aus der Familie der Ruhelosen Geistwesen ohne Todesursache. Ist nur in halbstabilen Fata Morganas zu finden, besteht zum größten Teil aus reflektiertem Licht, gefrorenem Zuckerdampf und gasförmig verdünnter Seelenessenz. Wie im Absatz über halbstabile Fata Morganas schon erwähnt, schmilzt bei Temperaturen über 160 Grad Celsius der Zuckerstaub der Süßen Wüste, beginnt zu kochen und läßt dabei feinen Zuckerdampf aufsteigen. Wenn die Lufttemperatur in diesem Augenblick stark abfällt (etwa durch plötzliche Fallwinde), erhärtet sich der Zucker mitten in der Luft, und falls zusätzlich das Bild einer real existierenden Oasenstadt auf die kristallisierenden Zuckermoleküle fällt, kann sich das Stadtbild fest darauf einbrennen. Dasselbe kann mit den Lebewesen geschehen, die sich in so einer Stadt befinden. So entstehen die sogenannten Fatome. Im Gegensatz zu herkömmlichen Gespenstern sind die Fatome nicht die Geister von Toten, sondern von Existenzformen, die durchaus noch am Leben sein können.
> Fatome dürfen zu den bedauernswertesten der zamonischen Geistformen gezählt werden. Sie verfolgen keinen direkten Zweck wie etwa das Verängstigen oder Begruseln von lebendigen Daseinsformen. Sie gewinnen auch kein Vergnügen aus ihrer Existenz wie Polter- oder Klabautergeister. Sie sind lediglich dazu verdammt, die Tätigkeit, die sie im Entstehen der halbstabilen Fata Morgana verrichtet haben, auf alle Zeiten zu wiederholen.
> From: `http://www.zamonien.de/lexikon/default.asp?key=13&F=S`
>
> **Anagrom Ataf** Anagrom Ataf ist eine halbstabile Fata Morgana bzw. eine oasenstadtförmige teilkonkrete Luftspiegelung in der Süße Wüste genannten Landschaftsform im Kontinent Zamonien. Bei Temperaturen über 160 Grad Celsius schmilzt der Zuckerstaub der Süßen Wüste, beginnt zu kochen und läßt dabei feinen Zuckerdampf aufsteigen. Wenn die Lufttemperatur in diesem Augenblick

stark abfällt (etwa durch plötzliche Fallwinde), erhärtet sich der Zucker mitten in der Luft, und falls zusätzlich das Bild einer real existierenden Oasenstadt auf die kristallisierenden Zuckermoleküle fällt, kann sich das Stadtbild fest darauf einbrennen. So entsteht eine halbfeste spiegelverkehrte Scheinstadt, die vom Wind einmal hierhin, einmal dorthin geweht werden kann, so daß der Eindruck entsteht, daß diese Stadt sich aus eigenem Willen bewegt.

From: `http://www.zamonien.de/lexikon/default.asp?key=1&F=S`

Bibliography

[Amd67] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.

[ANS92] ANSI/IEEE. IEEE Standard for Scalable Coherent Interface (SCI), 1992. Document No. 1596-1992.

[Arg89] G. Argo. Improving the three instruction machine. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 100–115. ACM Press, 1989.

[Aug87] L. Augustsson. *Compiling lazy functional languages, part II.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, 1987.

[Bar84] H. Barendregt. *The lambda calculus, its syntax and semantics.* North Holland, 1984.

[BKL+97] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. DREAM: The DistRibuted Eden Abstract Machine. In *Implementation of Functional Languages*, pages 250–269. Springer Verlag, 1997. Available from World Wide Web: `http://www.mathematik.uni-marburg.de/~eden/paper/dreamIFL.ps` [cited October 13, 2006].

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. High-level Parallel and Concurrent Programming in Eden. In *APPIA-GULP-PRODE*, pages 213–224, 1997.

[CFH+03] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. Peyton Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 Foreign Function Interface 1.0 – An Addendum to the Haskell 98 Report, 2003. Available from World Wide Web: `http://www.cse.unsw.edu.au/~chak/haskell/ffi/` [cited October 21, 2006].

[CG86] K. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):1–49, January 1986.

[CGK98] M. M. T. Chakravarty, Y. Guo, and M. Kohler. Distributed Haskell: GOFFIN on the Internet. In *3rd Fuji International Symposium* *on Functional and Logic Programming*, pages 80–97, 1998. Available from World Wide Web: `http://citeseer.ist.psu.edu/chakravarty98distributed.html` [cited October 13, 2006].

[CGKL98] M. M. T. Chakravarty, Y. Guo, M. Kohler, and H. C. R. Lock. GOFFIN: Higher-Order Functions Meet Concurrent Constraints. *Science of Computer Programming*, 30(1-2):157–199, 1998. Available from World Wide Web: `http://citeseer.ist.psu.edu/chakravarty97goffin.html` [cited October 13, 2006].

[Cha97] M. M. T. Chakravarty. *On the Massively Parallel Execution of Declarative Programs.* PhD thesis, Technische Universität Berlin, 1997. Available from World Wide Web: `http://www.cse.unsw.edu.au/~chak/papers/diss.html` [cited October 13, 2006].

[Che70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–687, November 1970.

[DFG+94] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In Jürg 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 Verlag, 1994. Available from World Wide Web: `http://uebb.cs.tu-berlin.de/papers/published/DesignImplOpal.ps.gz` [cited November 4, 2006].

[DHM+06] A. Discolo, T. Harris, S. Marlow, S. Peyton Jones, and S. Singh. Lock Free Data Structures using STM in Haskell. In *FLOPS '06: Proceedings of the Eighth International Symposium on Functional and Logic Programming*, April 2006. Available from World Wide Web: `http://research.microsoft.com/~tharris/papers/2006-flops.pdf` [cited October 21, 2006].

[FW87] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 34–45. Springer Verlag, 1987.

[FY69] R. R. Fenichel and J. C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Systems. *Communications of the ACM*, 12(11):611–612, 1969.

[Har98] S. J. Hartley. *Concurrent Programming.* Oxford University Press, 1998.

[Hin69] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[HW06] P. Hofstedt and A. Wolf. *Einfürung in die Constraint-Programmierung.* Springer Verlag, 2006. To appear.

[Joh87] T. Johnsson. *Compiling lazy functional languages*. PhD thesis, Chalmers Tekniska Högskola, Göteborg, 1987.

[Jon95] M. P. Jones. *Advanced Functional Programming*, chapter Functional Programming with Overloading and Higher-Order Polymorphism, pages 97–136. Springer Verlag, 1995. Available from World Wide Web: `http://web.cecs.pdx.edu/~mpj/pubs/springschool95.pdf` [cited October 20, 2006].

[KR98] A. Keller and A. Reinefeld. CCS Resource Management in Networked HPC Systems. In *Proc. Heterogenous Computing Workshop HCW'98 at IPPS, Orlando, 1998*. IEEE Comp. Society Press, 1998. Available from World Wide Web: `http://wwwcs.uni-paderborn.de/pc2/uploads/media/CCS_Resource_Management_in_Networked_HPC_Systems.pdf` [cited October 13, 2006].

[LM01] D. Leijen and E. Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001. Available from World Wide Web: `http://www.cs.uu.nl/~daan/download/papers/parsec-paper.pdf` [cited 21 October, 2006].

[LOP05] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marb́i. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[LP92] D. R. Lester and S. L. Peyton Jones. *Implementing Functional Languages: a tutorial*. Prentice Hall, 1992. Available from World Wide Web: `http://research.microsoft.com/~simonpj/Papers/pj-lester-book/` [cited October 13, 2006].

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Nit05] T. Nitsche. *Data Distribution and Communication Management for Parallel Systems*. PhD thesis, Technische Universität Berlin, 2005.

[NN92] H. R. Nielson and F. Nielson. *Semantics With Applications*. Wiley, 1992. Available from World Wide Web: `http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html` [cited October 13, 2006].

[PAB+03] S. L. Peyton Jones, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, J. Hughes, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, April 2003. Available from World Wide Web: `http://www.haskell.org/haskellwiki/Language_and_library_specification` [cited October 13, 2006].

[Pep97] P. Pepper. Programmiersprachen und -systeme. Lecture notes, 1997.

[Pey92] S. L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992. Available from World Wide Web: `http://citeseer.ist.psu.edu/peytonjones92implementing.html` [cited October 13, 2006].

[PGF96] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL '96: Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, 1996. Available from World Wide Web: `http://citeseer.ist.psu.edu/143219.html` [cited October 13, 2006].

[PH06] P. Pepper and P. Hofstedt. *Funktionale Programmierung – Sprachdesign und Programmiertechnik.* Springer Verlag, 2006.

[PS89] S. L. Peyton Jones and J. Salkild. The spineless tagless G-machine. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201. ACM Press, 1989.

[PSF+06] M. Pöppe, S. Schuch, R. Finocchiaro, C. Clauss, B. Bierbaum, and J. Worringen. *MP-MPICH – User Documentation & Technical Notes*, 2006. Available from World Wide Web: `http://www.lfbs.rwth-aachen.de/users/global/mp-mpich/mp-mpich_manual.pdf` [cited October 13, 2006].

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[Sha89] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys (CSUR)*, 21(3):413–510, September 1989.

[SOHL+96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

[SR89] V. A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM Press, 1989.

[ST95] N. Shavit and D. Touitou. Software Transactional Memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[Sta05] W. Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, 2005.

[THLP98] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998. Available from World Wide Web: `http://www.macs.hw.ac.uk/~dsg/gph/papers/abstracts/strategies.html` [cited November 4, 2006].

[UK05] K. Ueda and N. Kato. LMNtal: a Language Model with Links and Membranes. In *Proc. Fifth Int. Workshop on Membrane Computing*, pages 110–125. Springer Verlag, 2005. Available from World Wide Web: `http://citeseer.ist.psu.edu/638362.html` [cited October 13, 2006].

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den

Florian Lorenzen