



Universität zu Lübeck

Tiark Rompf

Design and Implementation of a Programming Language for Concurrent Interactive Systems

Diplomarbeit

am Institut für Softwaretechnik und Programmiersprachen

Direktor: Prof. Dr. Walter Dosch

Technisch-Naturwissenschaftliche Fakultät

Universität zu Lübeck

Betreuerin:

Dr. Annette Stümpel

2007-06-20

1. Gutachter: Prof. Dr. Walter Dosch
Institut für Softwaretechnik und Programmiersprachen
2. Gutachter: PD Dr. Andreas Jakoby
Institut für Theoretische Informatik

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Lübeck, 20. Juni 2007

Abstract

This thesis describes the design and implementation of a new experimental programming language with a focus on creating concurrent, interactive systems. As such, the language makes concurrent programming the default case, providing fine-grained concurrency primitives at the language level and a runtime system that is geared towards execution on multi-processor hardware. The language semantics are first described operationally, in terms of generalized Functional Petri Nets, and then refined to a model of asynchronously interacting components. Code examples are shown that describe how advanced object-oriented and functional programming concepts such as open generic functions/multimethods are a natural specialization of the core language abstractions. Other programming examples draw from areas such as dataflow-oriented graphical user interfaces, animation scripting and server-side web applications. A reference compiler/interpreter for the new language is also supplied, targeting the Java platform and showing good speedups on multi-processor hardware.

Acknowledgements

First and foremost, I would like to thank my advisors Prof. Dr. Walter Dosch and Dr. Annette Stümpel. In particular, Annette's continuous supervision and our numerous discussions have been extremely helpful and a great source of inspiration. Thanks, Annette!

Moreover, I would like to thank PD Dr. Andreas Jakoby for taking the job of a coadvisor and for providing additional valuable suggestions. A number of other people also deserve acknowledgement for proof-reading parts of this work.

Finally, I say "thank you" to all my friends and family for support and distraction, to V&S AB, Sweden, for their Absolutely inspiring products, to Steve Harris et. al. for making great music, and to Mary W. for obvious reasons.

Contents

1	Introduction	1
1.1	Outline	3
1.2	Contributions	3
1.3	Motivation	4
1.3.1	Multithreading	5
1.3.2	Event-Driven Programming	5
1.3.3	Graphical User Interfaces	6
1.3.4	Game Engines	7
1.3.5	Web Applications	8
1.3.6	Industrial Real-Time Systems	8
1.4	Goals	10
2	Design Decisions	11
2.1	Philosophers' Dinner Revisited	13
2.1.1	Function Symbols and Function Bodies	15
2.1.2	Function Bodies may Link Multiple Function Symbols	15
2.1.3	Function Symbols are First-Class Values	16
2.2	Advanced Concepts	18
2.2.1	Object-Oriented Programming	18
2.2.2	Functional Programming	19
2.2.3	Mutable State	22
2.2.4	Generators	23

2.3	Type System	27
2.4	Related Work	29
2.4.1	Join-Calculus Inspired Languages	30
2.4.2	Actor Languages	30
2.4.3	Other Influences	31
3	Core Language and Execution Model	33
3.1	Background — Formal Models of Concurrency	34
3.1.1	Petri Nets	34
3.1.2	Colored Petri Nets	35
3.1.3	Process Calculi	35
3.1.4	The Join Calculus	36
3.1.5	Functional Nets	37
3.2	Core Semantics I — Operational Execution Model	38
3.2.1	Informal Description	38
3.2.1.1	From Colored to Dynamic Nets	38
3.2.1.2	Patterns and Predicates	39
3.2.2	Definitions	42
3.2.2.1	Notational Conventions	42
3.2.2.2	Basic Abstract Data Types	43
3.2.2.3	Programs, Rules and Nets	44
3.2.2.4	Enabledness, Conflicts and Activation	45
3.2.2.5	Program Syntax and Execution	47
3.3	Core Semantics II — Component-Based Execution Model	50
3.3.1	Informal Description	50
3.3.1.1	Separating Concerns	50
3.3.1.2	Taming Non-Determinism	51
3.3.1.3	Differences from the Petri-Net Model	53
3.3.2	Definitions	55

3.3.2.1	The Scheduler Component	55
3.3.2.2	The Executor Component	56
3.3.2.3	The Activator Component	57
3.3.2.4	A Taxonomy of Patterns	59
3.4	Advanced Refinements	63
3.4.1	Distributed Model	63
3.4.2	Concurrent Model	64
4	High-Level Language	67
4.1	Instantiation of the Execution Model	68
4.1.1	Values	68
4.1.2	Patterns	70
4.1.3	Matching	71
4.2	High-level Syntax Encoding	73
4.2.1	Sequential and Parallel Composition	73
4.2.2	Value Expressions	76
4.2.3	Pattern Expressions in Assignments	77
4.2.4	Function Definitions	77
4.2.5	Type Definitions	79
4.2.6	Compilation Units	79
4.2.7	Integration with Platform Code	80
4.2.8	Syntactic Sugar	81
4.3	Reference Implementation	85
4.3.1	Availability	86
4.3.2	Performance Evaluation	86
5	Code Examples	89
5.1	Declarative Dataflow GUI Programming	90
5.1.1	Accessing Legacy Java Classes	90
5.1.2	Introducing Data Dependencies	91

5.2	Walking Robot (“glutmech”) Animation	94
5.2.1	Keyframe Animation	94
5.2.2	Identifying the Building Blocks	96
5.2.3	Re-assembling the Parts	97
5.2.4	Parallel Composition	98
5.2.5	The Power of Generators	99
5.3	Web-Application Server	101
5.3.1	Main Server Loop	101
5.3.2	Selectors and Callbacks	102
5.3.3	Parallelizing Callbacks	103
5.3.4	Accepting Connections	104
5.3.5	Retrieving Data	104
5.3.6	Application-Level Control Flow	106
5.3.7	Infrastructure	108
6	Discussion	111
6.1	Future Work	113
A	Syntax Specification	115
A.1	Compilation Units	115
A.2	Statements	116
A.3	Value Expressions	118
A.4	Pattern Expressions	121
A.4.1	Patterns in Rule Definitions	121
A.4.2	Patterns in Assignments	123

Chapter 1

Introduction

There is an increasing demand for concurrent and interactive software systems, but most established programming languages do not provide adequate support for concurrent programming. Mainstream languages that do support concurrency in most cases do so by means of threads, which do not prove satisfactory for fine-grained application-level concurrency.

In this thesis, we describe the design and implementation of a new experimental programming language that embodies parallelism as a fundamental construct and not as an add-on.

The new language has been inspired by the language Funnel developed by Odersky et al. [1999] and builds on the same theoretical underpinnings, an amalgamation of concepts from Petri Nets [Petri, 1962] and the Join Calculus [Fournet and Gonthier, 1996]. Like in Funnel, there is an $m:n$ -relation between function symbols and function bodies, with function symbols being first-class values. The fundamental rule remains, that a function body can only be executed when all its associated function symbols have been called.

In the new language, however, departing from the Join Calculus heritage, function bodies need not be defined together with their related function symbols.

Furthermore, the appropriate function body for a given list of actual parameters is selected dynamically by a best-match strategy respecting all the arguments. Combined with subtyping, this leads to a multimethod approach to object-orientation like in Dylan [Shalit, 1996] or Nice [Bonniot, 2003a], where methods and classes can be defined separately, making it easy to extend existing classes with new behavior.

Distributed systems can be realized by sending proxies of function symbols to a distant site. Calling a proxy symbol will send the arguments over the network and execute the function body at its home site, while setting up a function body on proxy symbols (whose originals reside on the same remote site—otherwise it is an error) will transfer the code there.

This thesis presents the semantics of a small core language on top of which higher-level language features are expressed. A reference implementation is supplied in form of a high-level to core language compiler and a core language interpreter running on the Java platform.

On multi-processor hardware, good speedups are achieved by employing optimistic software transactions and a work-stealing scheduler, a combination that allows for an essentially lock-free implementation.

1.1 Outline

In this introductory chapter, Section 1.3 describes the mainstream concurrent programming methodologies and explains their shortcomings. Section 1.4 lays down the goals of this thesis and the requirements for the new language.

Chapter 2 presents the proposed main language concepts, with Section 2.1 conveying a high-level impression of the language syntax by means of a code example, showing a solution to the Dining Philosophers problem. Section 2.2 explains how concepts from object-oriented and functional programming are provided, while Section 2.3 describes the type system used and Section 2.4 reviews related concurrent language research.

Chapter 3 defines the execution model and a small core language. Section 3.1 reviews the historic background of Petri Nets and the Join Calculus, which are the underlying theoretical abstractions. Section 3.2 describes an operational semantics in terms of a generalization of Petri Nets, which is recast in Section 3.3 in a component-oriented model that is easier to implement. Refinements of this model are discussed in Section 3.4.

Chapter 4 presents the high-level language. Section 4.1 describes how the execution model is instantiated, and Section 4.2 shows how high-level syntax translates to the core language. Section 4.3 describes the reference implementation and investigates its runtime performance.

Chapter 5 describes several non-trivial programming examples. Section 5.1 illustrates a method of data-flow oriented user-interface programming, Section 5.2 shows an example of modular keyframe-animation scripting and Section 5.3 describes the implementation of a web-application server.

Chapter 6 concludes and proposes some directions for future research.

1.2 Contributions

To the author's knowledge, the following aspects of this work are novel:

- combining Join Calculus concurrency with open generic functions (multimethods)
- using the model of asynchronously interacting components to describe programming language semantics

1.3 Motivation

In programming, concurrency is becoming an ever more important topic. One of the reasons is that software use-cases and requirements increasingly demand distributed, mobile, interactive and concurrent systems. In fact, it appears safe to say that most significant software systems being developed today are inherently concurrent and interactive, in the sense that their operation depends on a history of external stimuli.

Another reason is that future advances in hardware development are expected to occur more likely in the domain of parallelism, increasing the number of processor cores per chip or the number of chips per machine, than in an unceasing exponential growth in clock speed. Sutter speaks of “a fundamental turn toward concurrency in software” and concludes that “the free lunch is over”, i.e. applications will not continue to profit from radical CPU throughput gains without adapting to concurrent hardware [Sutter, 2005].

Programming concurrent systems, on the other hand, has always been a major challenge. Some regard concurrency as “the next revolution in how we write software”, comparable to Object-Oriented Programming “both in the (expected) scale of the revolution and in the complexity and learning curve of the technology” [Sutter, 2005].

One reason for this difficulty is the fact that concurrent programming is accounted for only inadequately by most programming languages. This lack of language support is widely recognized as an obstacle towards better software systems. Meyer for example, writing about concurrent programming as “the next programming frontier”, indulges in an especially colorful analysis of the situation [Meyer, 2005]:

“Like an entire province forgotten by history, one area of programming is still living in the nineteen sixties, to the point that you can almost see the flowers in the hairdo and hear the Beatles on the radio. [...]

What makes this situation ever less bearable is that programming is becoming increasingly concurrent. Multithreading, in particular, is everywhere, handled through incredibly low-level techniques, usually through libraries relying on semaphores and similar techniques and affixed to object-oriented languages like a wooden wing to a race car.

We all suffer from the results, because so many applications today need multithreading or some other form of concurrency.”

In Section 1.3.1 below, we discuss multithreading as today’s predominant concurrent programming model. In Section 1.3.2, we contrast multithreading with the event-driven approach to interactive systems.

1.3.1 Multithreading

The prevailing concurrent programming methodology is based on the notion of multiple threads of execution. Threads are sequential in nature, but can execute concurrently, possibly signaling, halting, delaying or otherwise starting/stopping each other.

Threads, as provided by most operating systems (OSs), are the method of choice to utilize the parallelism of multi-core hardware, but they are not well suited for fine-grained application-level concurrency.

Threads model independent sequential computations and do not by themselves support synchronization or communications. Instead, threads rely on mutex-locks, semaphores [Dijkstra, 1968] or monitors [Hoare, 1974] to provide synchronization and to protect shared resources. These are very basic tools that require careful use by the programmer. Lee shows that it can be difficult to make even simple programs “thread-safe” i.e. behave in a predictable, deterministic way and concludes that “non-trivial multi-threaded programs are incomprehensible to humans” [Lee, 2006].

Furthermore, creating and destroying OS threads is expensive, and there is a high memory cost per thread because each thread needs to maintain an individual calling-stack. This poses a scalability problem for highly interactive systems with many, short-lived, concurrent tasks. Therefore, application programmers employ techniques like thread pools to keep the number of OS threads low, which further complicates the programming.

All in all, many consider threads too low-level to qualify as a suitable programming abstraction. Again quoting Lee [2006]:

“If we expect concurrent programming to be mainstream, and if we demand reliability and predictability from programs, then we must discard threads as a programming model.”

1.3.2 Event-Driven Programming

In order to circumvent the scalability and synchronization issues of multithreading, programmers often resort to variants of explicitly coded state machines, or event-driven programming [Dabek et al., 2002], where an *event*, in this sense, is the occurrence of a state

change. Event-driven programming is especially common in designing interactive systems, where much of the programming logic is concerned with responding to external input.

In event-driven models, there are no blocking operations such as those commonly used to read input data from I/O devices. Instead, the programmer defines event handlers, which specify the actions to take when a particular event occurs (e.g. the arrival of input data, or a mouse-click).

Detecting the occurrence of events and calling the appropriate event handlers is the responsibility of a runtime environment, which may or may not be part of the program itself (as an explicitly coded event-loop) or provided by some kind of library or framework. Because of this call-back nature, event-driven programming is said to incur an “inversion of control”, as it is not the program itself that proactively decides what to do when, but the event-dispatching mechanism.

Event-driven programming, while being far less prone to synchronization-related problems, has other drawbacks [Haller and Odersky, 2006]. Most notably, conceptually related programming logic is fragmented across multiple event handlers, violating the principles of encapsulation and modularity. Another drawback is that control flow between event handlers is defined only implicitly. As a result, event-driven code has a reputation of being tedious to write and hard to maintain.

Furthermore, event-driven systems must be carefully crafted to take advantage of multi-processor hardware. In those cases where event-driven systems are built in mainstream languages, there is often only one global event-handling loop (or a small number thereof), which obstructs an efficient parallelization of the system.

In the remainder of this section, we take a closer look on some important programming use-cases where multithreading is frequently avoided and some form of more or less elaborate event-driven style is relied upon.

1.3.3 Graphical User Interfaces

Graphical user interfaces (GUIs) are traditionally built around a single event loop. An application retrieves one event at a time from a central event queue, and dispatches that event to the appropriate UI component. After the event is fully taken care of, the application waits for the next event to occur. Modern UI toolkits allow application components to register callback handlers, so the event loop is not visible anymore, but in most cases the basic sequential structure remains.

Some ten years ago, Fuchs has shown how user interfaces could benefit from a more natural programming style [Fuchs, 1996]. The suggested approach was to associate callback handlers with explicit continuations, thereby implementing a form of application-level threading. More recent research [Quan et al., 2003, Graunke and Krishnamurthi, 2002] also suggests to make use of continuation-passing paradigms in order to make modal dialog boxes less disruptive to the user’s workflow and to “bookmark” user-interface state.

When thinking of user interfaces, what comes to most people’s minds are word processing, spreadsheet—or in general: desktop—applications. But sophisticated and unobtrusive user interfaces are at least as important in embedded systems like e.g. point of sale platforms or medical equipment such as anaesthesia devices in operating rooms.

A recent trend is applications switching to web technologies on the UI side. Examples of this are Google Docs & Spreadsheets [Google, 2006] or the hospital information system Siemens Soarian [Haux et al., 2003]. It has been only recently that developing rich web-based UIs has become feasible with the advent of AJAX (Asynchronous JavaScript And XML). Another approach is the one taken by Macromedia Flex [Adobe Systems, 2004] and OpenLaszlo [Laszlo Systems, 2005]. With these toolkits, developers can create sophisticated multi-window applications by declaratively specifying GUI elements along with an associated data model. The provided features include sequential and parallel animation effects, and the runtime system ensures that declared properties are kept satisfied by a form of automatic constraint solving.

It is a speculation, but declarative UIs may turn out to be a unifying foundation sufficiently general for all of the above use cases (desktop, embedded and web clients). As a basis for implementing such versatile frameworks, languages which provide concurrent programming in a modular way might be a valuable tool.

1.3.4 Game Engines

Computer games typically involve numerous actors—entities that behave on their own and interact with each other. Most of the time, games are set up as one single loop that calculates the current time, updates the actors’ positions, processes collisions and other events, and finally draws the environment and all the actors to the screen. Sophisticated game engines provide scripting languages that allow developers to express actor behavior without touching engine code, but most languages are limited to a reactive programming style. That is, the programmer may only define the direct reaction to a single event. To

implement sophisticated actor behavior that spans multiple animation frames, a lot of state has to be tracked.

Only recent developments improve on this design. Using microthreads provided by the Stackless variant of the language Python, the game engine Sylphis3D is one such example. The author of this toolkit also gives an in-depth account of the methodology used and how a straightforward concurrent programming model can lead to a cleaner game design [Kalogirou, 2005].

While not intended for entertainment purposes, a number of industrial simulation tools also use game development methodologies. An example is the simulation toolkit Geist3D [Stier et al., 2006] which combines Petri Net based simulations with Python scripting, thus following an approach that is not unsimilar to the one taken in this thesis.

1.3.5 Web Applications

Web applications need to track user sessions across individual HTTP requests, even more so with the advent of AJAX. As a very simple example, web applications often require users to authenticate on a separate login page, before they can access the application. When accessing an internal page of the application via a bookmark, the user is first redirected to the login page. After logging in, the application needs to present the originally requested page to the user.

From a high-level point of view, presenting a login page before the requested one can be seen as a form of subroutine call. Recently, some web development packages (e.g. Apache Cocoon [The Apache Software Foundation, 2006] or the SmallTalk-based Seaside [Ducasse et al., 2004]) have been progressing in this direction by allowing cross-request scripting, again by the notion of continuations. Of course, the situation is complicated by the user possibly hitting the browser's "back"-button or opening a link in a new browser window [Graunke et al., 2003].

Similar use cases appear in service oriented architectures (SOAs), where scripting across Web Service invocations becomes necessary, and in a variety of other communication protocols.

1.3.6 Industrial Real-Time Systems

Software systems in the industry are frequently concerned with the supervision and control of machinery or other tasks that impose explicit timing constraints on the application.

Violation of these constraints may be hazardous, so special care must be taken in developing such systems.

While there exist a number of multi-threaded real-time operating systems like QNX [Sastry and Demirci, 1995] or VxWorks [Beneden, 2002], real-time systems are frequently implemented on bare hardware. Systems developed in this manner usually poll for input data at fixed time intervals, utilizing more or less advanced priority schemes. As a simple example, high-priority sensor A might be checked at every iteration, while low-priority sensors B and C are only checked at even and odd iterations, respectively. The fixed time intervals guarantee a maximum latency. But in order to keep up this property, computing the system's reaction to some sensor input must also be known to finish in time.

Even though the final implementation of a real-time system is often very close to the hardware, developers use sophisticated tools like those based on Colored Petri Nets [Ratzer et al., 2003] in the specification and prototyping phases to study a system's behavior before it is actually built.

Having a suitable concurrent programming language at hand might be of use to real-time developers so they can combine the features of Petri Net or state-machine based simulation tools with the expressiveness of a programming language and build even more complete simulations. The ultimate but distant goal would be to design systems using a high-level language and to be able to compile this prototype implementation directly to a low-level executable with statically checked real-time behavior.

1.4 Goals

From the remarks in Section 1.3, a number of conclusions can be drawn:

- Concurrency is everywhere.
- Programmers use threads, which is bad because few do it right.
- Programmers don't use threads, which is also bad because code gets cluttered and concurrent hardware is not put to use.

We argue that the problems concurrency poses are largely due to the fact that mainstream languages do not offer adequate concurrency abstractions and that expressive languages with strong application-level concurrency might help alleviate these problems.

The goal of this thesis is to develop such a language. Rather than adding concurrency to an existing language as an afterthought, we feel that the natural way should be to make concurrent programming the general case and customary object-oriented and functional programming paradigms a specialization. Figure 1.1 contains a summary of the requirements we set forth for the new concurrent language to be developed.

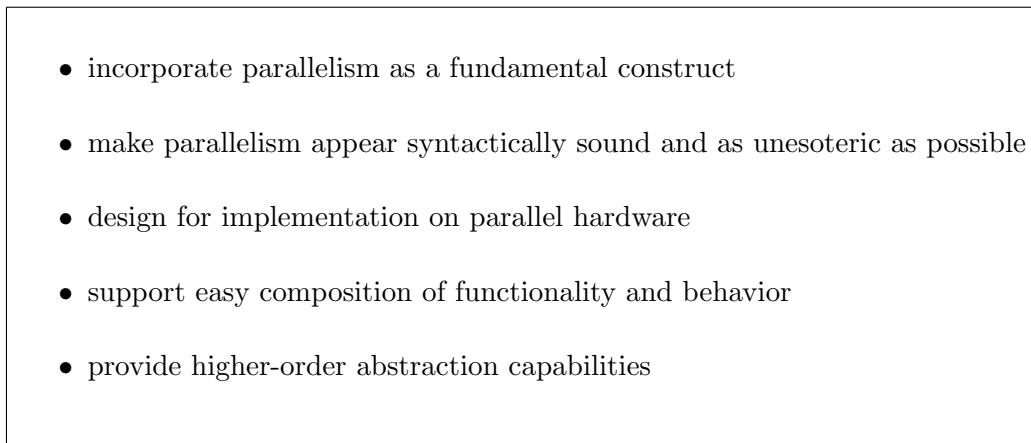
- 
- incorporate parallelism as a fundamental construct
 - make parallelism appear syntactically sound and as unesoteric as possible
 - design for implementation on parallel hardware
 - support easy composition of functionality and behavior
 - provide higher-order abstraction capabilities

Figure 1.1: Requirements for the new programming language.

Chapter 2

Design Decisions

In this chapter, we present the main concepts of the language design we propose in pursuit of the goals described in Section 1.4. The description is intentionally high-level, as a formal treatment follows in Chapters 3 and 4.

- Function Symbols and Function Bodies are defined separately.
- Function Symbols are First-Class Values.
 - ⇒ *higher-order composability*
- Function Bodies may link multiple Function Symbols.
 - ⇒ *body executed only when all associated symbols called*
 - ⇒ *executing body consumes pending calls*
- Function Symbols may have multiple associated Function Bodies.
 - ⇒ *calling function symbol will choose best match*
 - ⇒ *symmetric multiple dispatch*

Figure 2.1: An overview of the main language concepts.

The basic rules are summarized in Figure 2.1. Striving for concurrency as a *fundamental* property of the language, we investigate the concurrency-related items first, by means of a programming example (see Section 2.1). Section 2.2 takes up upon the basics and describes how object-oriented and other programming characteristics fit into the model.

The concurrency model of our choice is to a great extent inspired by the language Funnel [Odersky et al., 1999] and the Join Calculus [Fournet and Gonthier, 1996]¹. Section 2.4 reviews the concurrency approaches of other programming languages and their relations to our model, including a brief account of the differences between our language and Funnel.

¹The Join Calculus as a theoretical foundation is reviewed in Section 3.1.4.

2.1 Philosophers' Dinner Revisited

In order to get acquainted to the language, we examine an implementation of the Dining Philosophers problem [Dijkstra, 1971] in some detail, explaining the main language concepts as we go along. The setup of this problem, which is among the classics of resource allocation problems, is as follows:

A certain number of philosophers sit around a dining table, each pair of neighbors sharing a single fork. The philosophers spend their lives alternating between thinking and eating but in order to begin eating, a philosopher needs to pick up both forks next to him. When starting to think, the philosopher puts down the forks again.

In Figure 2.2 on page 14, the unabridged code is shown while the remainder of this section consists in a bottom-up description of the individual pieces. Regarding the code in Figure 2.2, we can make several observations:

- the philosopher code is succinct and high-level
- several forks can be grabbed at once without risk of deadlock
- philosophers are highly encapsulated: they do not depend on aspects of the outer world other than their forks (i.e. they are indifferent to their position at the table and to the number of their tablemates)

Most solutions based on more rudimentary synchronization mechanisms sacrifice at least one of these points. Another concern in concurrent programming is to prevent starvation, which means (taken literally in this example) making sure that every philosopher gets a chance to eat from time to time. While not apparent from the code, the new language can be implemented in such a way that execution is not prone to starvation-related problems (see Section 3.3.1.2).

```
new initPhilosopher: (Unit-->Void, Unit-->Void, String) --> Void;
def initPhilosopher(leftFork, rightFork, name):
{
  new takeForks: Unit --> Unit;
  new releaseForks: Unit --> Unit;

  def leftFork() & rightFork() & takeForks(): return();
  def releaseForks(): leftFork() & rightFork() & return();

  new eat: Unit --> Unit;
  new think: Unit --> Unit;

  def eat(): ...      # implement eating and thinking here
  def think(): ...

  new philosopher: Unit --> Void;

  def philosopher():
  {
    takeForks();
    eat();
    releaseForks();
    think();
    philosopher();
  }

  philosopher();
}

new forkA: Unit --> Void;
new forkB: Unit --> Void;
new forkC: Unit --> Void;
new forkD: Unit --> Void;

forkA() & forkB() & forkC() & forkD() &
initPhilosopher(forkA, forkB, "Descartes") &
initPhilosopher(forkB, forkC, "Sokrates") &
initPhilosopher(forkC, forkD, "Sartre") &
initPhilosopher(forkD, forkA, "Kant");
```

Figure 2.2: A solution to the Dining Philosophers problem.

2.1.1 Function Symbols and Function Bodies

We start by defining the behavior of a single philosopher as a recursive function:

```
new philosopher: Unit --> Void;

def philosopher():
{
  takeForks();
  eat();
  releaseForks();
  think();
  philosopher();
}
```

The first thing to mention here is the explicit division between function symbols and function bodies. Executing the statement in the first line creates a new function symbol and makes it accessible in the current scope under the identifier `philosopher`. Executing the second statement attaches a function body to the previously created symbol.

2.1.2 Function Bodies may Link Multiple Function Symbols

In the code above, subroutines are used to take and release two forks simultaneously. This is one of the nontrivial parts of traditional solutions where each fork is modeled by one synchronization primitive that allows only individual access control, but not in coaction with others. A naive implementation can lead to deadlock, e.g. when all philosophers hold only their respective left fork and wait for the other to be released.

Here, the language allows to circumvent this issue by declaring a function body that is only executed after multiple function symbols have been invoked:

```
new leftFork: Unit --> Void;
new rightFork: Unit --> Void;

new takeForks: Unit --> Unit;
new releaseForks: Unit --> Unit;

def leftFork() & rightFork() & takeForks(): return();
def releaseForks(): leftFork() & rightFork() & return();
```

As opposed to the function symbols labeled `leftFork` and `rightFork`, which are of type `Unit --> Void`, the functions `takeForks` and `releaseForks`² are of type `Unit --> Unit`. The type `Void` literally describes the *empty* type that does not contain any values. A call to a function declared to return a `Void` result will never terminate. The type `Unit`, however, contains the single value `()` and functions declared to return `Unit` (or any other type) *may* return control to their caller.

²Although technically imprecise, the term *function* will be used in the following to denote an identifier referring to a specific function symbol.

The meaning of the function body definitions is as follows: If there are calls to `takeForks`, `leftFork` and `rightFork` present at the same time, control will be returned to the caller of `takeForks`, while the pending calls are consumed. Since this is the only function body attached to the function `takeForks`, a call to `takeForks` will remain pending until both other functions are called, too. The same holds for calls to `leftFork` or `rightFork`: only after all three functions are called is the function body executed, returning control to `takeForks`'s caller. The order in which the functions are called is not important and is in no way related to the order in which the functions appear in the `def` statement.

The order of appearance in the `def` statement has a different effect, though: While it is possible to specify explicitly which of the listed functions are supposed to return (e.g. with `returnto.takeForks()`), the shorthand `return()` always refers to the rightmost function in the list.

In fact, `return` is just an automatically generated function of type `Unit --> Void` (because `Unit` is the return type of `takeForks`) that can be used like any other function, and `returnto` is nothing but a record of such functions labeled with appropriate names.

A call to `releaseForks` returns in any case, but produces calls to `leftFork` and `rightFork` *in parallel* to returning control.

2.1.3 Function Symbols are First-Class Values

So far, the code is concerned with only one philosopher. What remains to do is to differentiate between those items that are specific to one philosopher and those that are shared (i.e. the forks):

```
new initPhilosopher: (Unit --> Void, Unit --> Void, String) --> Void;
def initPhilosopher(leftFork, rightFork, name)
{
  new takeForks: Unit --> Unit;
  new releaseForks: Unit --> Unit;

  def takeForks() & leftFork() & rightFork(): return();
  def releaseForks(): leftFork() & rightFork() & return();

  ...
}
```

The language feature used here is that function symbols are first-class values: they can be given arbitrary local identifiers, they can be passed as parameters to function invocations and function bodies can return function symbols just like any string or integer. Furthermore, function bodies need not be defined together with function symbols in the

same syntactic block. Instead, new function bodies may also be attached to function symbols that were passed as parameters or returned from a function invocation.

Now, `leftFork` and `rightFork` refer to arguments passed to `initPhilosopher`. This way, the same forks can be passed to several philosophers, each of which in turn adds an additional function body that synchronizes with that philosopher's `takeForks` function. In parallel, initial calls must be made to the individual forks so they can be picked up by one of the philosophers:

```
new forkA: Unit --> Void;
new forkB: Unit --> Void;
new forkC: Unit --> Void;
new forkD: Unit --> Void;

forkA() & forkB() & forkC() & forkD() &

initPhilosopher(forkA, forkB, "Descartes") &
initPhilosopher(forkB, forkC, "Sokrates") &
initPhilosopher(forkC, forkD, "Sartre") &
initPhilosopher(forkD, forkA, "Kant");
```

The calls to `forkX` and `initPhilosopher` are delimited by the `&`-operator, which causes these statements to be executed in parallel.

Putting all these pieces together yields the code displayed in Figure 2.2. As already mentioned at the beginning of this section, the solution developed here has some desirable properties that are not as easily achieved using traditional synchronization primitives.

2.2 Advanced Concepts

In the previous Section 2.1, we have introduced function symbols and function bodies, as well as the fact that they need not be declared together. In the example, the focus has been mostly on concurrency, while in this section, we show how some well-known programming paradigms are expressed as special cases.

2.2.1 Object-Oriented Programming

Due to the fundamental distinction between function symbols and function bodies, calling a function symbol may result in the execution of different method bodies.

Thinking of object-oriented languages, this is not a new idea: If some class *overrides* a method declared in one of its superclasses, it effectively adds a new method body to the existing method symbol. Methods may also be *overloaded*, i.e. reimplemented for a different parameter list (see Figure 2.3³).

```
class Object
{
    String toString()
    {
        ...
    }
}

class Integer extends Object
{
    String toString()
    {
        ...
    }
}

class PrintWriter
{
    void println(Object x)
    {
        ....
    }

    void println(Integer x)
    {
        ...
    }
}
```

Figure 2.3: Overriding and overloading methods in Java: Multiple function bodies on function symbols `toString` and `println`.

Therefore, it is self-suggesting for the new language to support object-oriented programming in a similar manner. The approach we take differs from the one of languages like Java in an important detail, though: While dynamic method dispatch, i.e. function body selection at call-time, in traditional languages takes into account only the type of the first argument (the `this` object), our language dispatches dynamically on *all* the arguments.

³Focusing on the two methods chosen, we neglect the fact that in Java, the `extends Object` relationship is implicit and thus, `PrintWriter` would also be a subclass of `Object`

The behavior of traditional object-oriented languages often gives rise to confusion. Assigning an object of type `Integer` to a variable `anObject` declared as `Object` and then invoking `p.println(anObject)` on a `PrintWriter p` will, in Java, execute the more general method body (expecting an `Object`), even though the actual argument is of type `Integer`. Therefore, the actual `PrintWriter` implementation calls `toString` on the argument object in order to ensure proper textual output.

Another example which requires this double indirection is traversing a hierarchical data structure of heterogeneously typed objects in several functionally different passes, also known as the Expression Problem [Wadler, 1998]. The common solution to this problem is applying the Visitor Pattern [Gamma et al., 1995]: Data objects are requested to “accept” a visitor object and then call back the visitor to “visit” the data object, thus dispatching first on the target object’s type and then on the visitor’s. Like other design patterns, the visitor pattern is criticized as not being an example of best practice in software development but merely a workaround for shortcomings in the languages used [Bonniot, 2003b].

Considering the function body selection process in our language, it is clear that, when invoking a function symbol, the dispatch mechanism must take into account which other function symbols have been called at any rate. Otherwise, it would not be possible to determine which of the function bodies could be executed. Fortunately, it is only a small step from this basic mechanism to the object-oriented one, which guarantees to select the most specialized function body⁴, out of those that fit the given function arguments.

With this implementation of multiple dynamic dispatch (as opposed to single dispatch in the traditional case), all arguments are equal from a method body selection perspective. Consequently, there is no reason anymore why methods need to *belong* to classes. In fact, methods, which are also referred to as *multimethods* in this case, can be declared as generic functions outside of classes, even in unrelated places of a program.

In Figure 2.4, we show how the example from Figure 2.3 can be recast in the new language.

2.2.2 Functional Programming

Just like in functional programming languages, variable assignments are non-destructive in our language. That is, the meaning of the statement `a = 7` is not to alter a memory location, overwriting whatever previous value with the value 7, but to replace the name `a`

⁴In practice, there may be several ones that fit equally well.

```
type Object;
type Integer is Object;

new toString: Object --> String;

def toString(Object x):
{
  ...
}

def toString(Integer x):
{
  ...
}
```

```
type PrintWriter;

new println: (PrintWriter, Object) --> Unit;

def println(PrintWriter w, Object x):
{
  ...
}

def println(PrintWriter w, Integer x):
{
  ...
}
```

Figure 2.4: Generic functions in the new language.

with the value 7 in all following statements in the current code block. The language, however, is not generally free from side-effects. Clearly, function symbols represent mutable state, as invocations may remain pending until they are consumed by the execution of a function body. As a consequence, there is a priori no referential transparency as in pure functional languages.

Nonetheless, we can implement common functional programming idioms easily. Anonymous functions (λ -abstractions) e.g. can be written as `fn arg:body`. In Figure 2.5, we show some simple non-anonymous recursive functions with pattern matching. Algebraic datatypes, as another example, can be constructed by creating a type hierarchy with a common supertype. Functions can then dispatch on the argument types, using pattern matching like in a traditional functional language.

Moreover, there are additional benefits from the combination of object-oriented and functional features, especially with regard to algebraic data types. By introducing new subclasses, we can add new alternatives, and by implementing new methods, we can add new behavior. Simpler object oriented languages only allow subclassing (but no

```

new fac: Int --> Int;

def fac(0): return(1);
def fac(n): return(n * fac(n-1));

new even: Int --> Bool;
new odd:  Int --> Bool;

def even(0): return(true);
def even(n): return(odd(n-1));

def odd(0): return(false);
def odd(n): return(even(n-1));

```

Figure 2.5: Implementing simple recursive functions with pattern matching.

adding of methods to classes) and traditional functional languages only allow adding new functions, but not adding alternatives to option types (which would require adding new cases to existing functions). But from a modularity and composability point of view, both approaches supplement each other.

As a matter of fact, we use a well-known concept from functional programming to provide a more appealing syntax for object-oriented programs. Since methods are nothing but generic functions, without special syntax, we would have to write `toString(anObject)` instead of `anObject.toString()`. Since the `.`-operator is already reserved for record-field access, we define `->` to be the *apply-partial* operator, which we can even do in the language itself (see Figure 2.6).

```

new '->';

def '->' (a, m):
{
  new app;

  def app():          return m(a);
  def app(x):        return m(a, x);
  def app(x, y):     return m(a, x, y);
  def app(x, y, z):  return m(a, x, y, z);
  def app(x, y, z, w): return m(a, x, y, z, w);

  ...

  return app;
}

```

Figure 2.6: Methods as value-dispatching functions in the new language.

This way, we can write `anObject->toString`, denoting the anonymous function that will call `toString(anObject ...)` with `...` replaced by its own actual parameters. Now, `anObject->toString()`, which is exactly the C++ notation, will work just as expected. But what is more, the partially applied function `anObject->toString` is itself a first-class value.

2.2.3 Mutable State

As mentioned above, there are no mutable variables in our language. By making use of the state-fullness of function symbols, however, we can easily implement them in the language itself.

```
new initMutableVar;
def initMutableVar(x):
{
  new cell.get: Unit --> Object;
  new cell.set: Object --> Unit;

  new state: Object --> Void;

  def state(x) & cell.set(y): state(y) & return();
  def state(x) & cell.get(): state(x) & return(x);

  return(cell) & state(x);
}
```

Figure 2.7: A mutable memory cell.

The program snippet in Figure 2.7 implements a simple mutable memory cell, by combining a never-returning function `state` with a getter and a setter function. Note that the value returned by the function `initMutableCell` is not an object of any user-defined type but a record with two function-value fields `get` and `set`. Wrapping up mutable variables in a class hierarchy would not be a big undertaking, though.

The memory cell presented in Figure 2.7 is of course a very simple model that would suffer from all problems commonly associated with mutable state. By extending the interface a little bit, however, we can build more advanced models. As one example, we can design a software-transaction model, where mutable cells keep a version number that is incremented on every change. Variables must then be *entered* into a transaction, saving their current version number and yielding a copy that is only valid within the transaction. Upon *committing* a transaction, an attempt is made to lock all entered variables, succeeding only if each variable's original has not been altered concurrently (the

version number is used to check that). If all variables can be locked successfully, they are unlocked, taking the new value from the temporary copy and updating the version tag if the new value differs from the old. If any variable has been modified in the meantime the locking fails, all other variables are unlocked without modification and the commit operation signals a failure.

The regular way to handle such a failure is to retry the operation in a new transaction. In fact, we can define a higher-order function `atomic` that does just that, repeatedly executing another function with a fresh transaction until the commit succeeds (for the sake of brevity, we omit the code here).

2.2.4 Generators

Generators [Kiselyov, 2004, Liskov, 1993] are a feature that have found their way into popular languages not long ago. Conceptually, generators are functions that produce a sequence of values, yielding control to a receiver each time a new value is generated, in most languages with a special `yield` keyword and support from the runtime system. After the receiver has consumed the new value, the generator resumes after the `yield` statement. This way, sequential output from complex data structures can be produced in a forward manner, being consumed on the fly, without intermediate data-structures.

Due to the concurrent nature of our language, generators (and coroutines, as the implementation mechanism [Conway, 1963]) are a natural fit. Instead of introducing a special keyword, we can yield control from any function, just by calling another function received as a parameter. Additional syntax, though, is provided on the consumer side to facilitate generator comprehension. Figure 2.8 shows a simple generator implementation, and different syntax for printing the generated values to standard output. There is also a built-in syntax for generator literals, which allows the `genTest` generator to be written as `[1,2,3]`.

<pre>new genTest; def genTest(yield): { yield(1); yield(2); yield(3); return(); }</pre>	<pre>genTest(print); genTest->foreach(print); print x for x in genTest; for x in genTest: print x;</pre>
--	--

Figure 2.8: Generator definition (left) and usage (right).

Generators are especially powerful when combined through higher-order functions, and many of those that commonly appear in functional languages can be implemented straightforwardly. In Figure 2.9, we show a simple range operator that enables expressions like `3..7`, and the higher-order function `zip`, which synchronizes multiple (in this case: two) source generators. Other generator combinators can be written accordingly, either from scratch or by composing existing ones with the help of generator comprehension statements (`for ... in ...`).

```

new '..';
def '..'(x, y):
{
  new gen;
  def gen(yield):
  {
    new iter;
    def iter(n, false):
    {
      returnto.gen();
    }
    def iter(n, true):
    {
      yield(n);
      iter(n+1, n+1<=y);
    }
    iter(x, x <= y);
  }
  return gen;
}

```

```

new zip;
def zip(g1, g2):
{
  new gen;
  def gen(yield):
  {
    new r1;
    new r2;

    def r1(a1) & r2(a2):
    {
      yield(a1, a2);
      returnto.r1() &
      returnto.r2();
    }
    g1(r1) & g2(r2);
    return();
  }
  return gen;
}

```

```

map = fn gen, f: (f(x) for x in gen);
zipWith = fn (g1, g2), f: zip(g1, g2)->map(f);
cartesian = fn a, b: (x, y for x in a for y in b);

```

Figure 2.9: Some common higher-order generator functions.

Figure 2.10 illustrates some example uses of generators. An important thing to remember is that generators are in fact functions (and thus first-class values), so generator comprehension expressions, too, can be treated as such. Another powerful abstraction are reduction operations (like `sum`, `max` or `all`) that are also common in functional programming languages. Together with the lazy nature of generators, they provide for a natural formulation and efficient implementation of solutions to non-trivial problems such as whether two trees have exactly the same inorder-sequence of leaves (known as the “Samefringe-Problem” [Gabriel, 1991])⁵.

⁵Assuming in this example that the leave-generators produce a special end-of-fringe marker to guarantee termination.

```

# generators as values
oneToTen = 1..10;
oneFourNine = [1,4,9];
oddNumbers = i if i%2 == 1 for i in 1..10;

# reduction functions
sumOfSquares = sum(x*x for x in 1..10);
sameFringe = all((treeA->leaves(),treeB->leaves())->zipWith('=='));

# printing square numbers: 1, 4, 9
for x, y in zip(1..3, 1..3):
  print(x * y);
(1..3, 1..3)->zipWith('*')->foreach(print);
[1,2,3]->map(fn x: x*x)->foreach(print);

# printing cartesian product: (1,1),(2,1),(1,2),(2,2),(1,3),(2,3)
for y in 1..3:
  for x in 1..2:
    print(x, y);
cartesian(1..2, 1..3)->foreach(print);

```

Figure 2.10: Example generator usage.

By nature of their construction, generators operate in “push” mode, proactively producing values and handing them off to a receiver. In some cases, though, it is desirable to have “pull” access to a data source, e.g. when the pattern of consumption is non-regular as is the case in Figure 2.11. For those cases, we can define a generalized pull-handler that inverts the generator logic, providing an iterator-style `next` function for any given generator (see Figure 2.11).

Along the same lines, with just a little more code, we can also build a fully updatable cursor that provides `prev`, `update` and `delete` functions that operate without stateful modification of the underlying generator, implementing a very basic instantiation of the Zipper approach [Huet, 1997].

Last but not least, we can build generators that yield values in parallel instead of in sequence, by issuing statements like `yield(1) & yield(2) & yield(3)`; or using the shorthand syntax `[1 & 2 & 3]`. For the most part, higher-order functions need not be

```
new pull;

def pull(gen):
{
    new yield;
    new next;

    def yield(n) & next(r):
    {
        returnto.yield(r) &
        returnto.next(n);
    }

    gen(yield) & return(next);
}

next = pull(1..100);

print next();

next();
next();

print next();

next();
next();
next();
next();

print next();
```

Figure 2.11: Inverting generator logic: “pull” mode.

changed to accommodate for parallel generators. The functions `map` and `zip`, for example, carry over unmodified, as well as the `for`-comprehension syntax.

2.3 Type System

In the majority of examples above, we have provided type annotations in `new` statements. The type system of the new language, however, is a dynamic one, at least for the time being. In those cases where explicit types are given, runtime checks will produce error messages when type assertions are violated, but no type annotations are required.

The reasons we have chosen the route of dynamic typing were for the most part pragmatic. We acknowledge that static typing has undeniable benefits but implementing a suitable static type system was considered beyond the scope of this work.

The main reason why programs in our language are hard to type statically is that our model of object-orientation requires open subtyping and at the same time, we need non-subtype polymorphism for implementing functional features (e.g. `map`). Being forced to write out complex parametric type declarations is a pain, so an advanced system of type inference would be called for. Open dynamic subtyping, however does not play well with type inference, at least not without major help from the programmer. When attaching a function body to a function symbol, for example, one would require that the type of values the function body accepts is compatible with the function symbol's argument type, and, more importantly, that all method bodies return subtypes of the function symbol's return type. With simple statically declared types this would be no problem, but for the more complex types we require, pattern implication would have to be calculated which is a computationally hard problem.

Another point is that, in theory, it would be desirable to check that there is always a most specific method implementation and that no accidental or parasitic non-deterministic conflicts are introduced by new method bodies. With function bodies attached to multiple function symbols, however, it may well arise the case that there are several equally specific method bodies, of which only one at a time is enabled at runtime (because all of them depend on other calls), so there would be in fact always a most specific one. The compiler cannot verify this by just looking at the types of the function symbols. Instead, it would have to check all possible combinations of values that might be present on the involved function symbols.

Along the same lines, it would be nice to check that for all arguments, there will eventually be an appropriate method body. But with our model, method bodies may be defined later on, after some call has already been made. Again, we would like to know in advance, if at any time in the program's course, a method body would be set up to handle this value. At this time, we do not even know if this question is decidable at all.

Checking these properties, even in a conservative manner, would require a sophisticated flow-analysis, which is something a conventional static type system cannot provide. So the actual value of such a type system to our language may well be questioned.

2.4 Related Work

Even in earlier days of programming, languages have provided concurrency constructs, so the idea of a concurrent programming language is not at all new. In this section, we review the concurrency features of some languages that are either of some general interest or in some way related to our work.

Speaking of the early days, there are Simula [Dahl and Nygaard, 1967] (coroutines), Concurrent Pascal [Brinch Hansen, 1975] (monitors), Modula 3 [Cardelli et al., 1992] and of course Ada [Taft and Duff, 1997]. As one of the more widespread languages in industry, up to today, Ada provides concurrent tasks and supports synchronization by explicit rendezvous. Care must be taken though, because in many cases, undisciplined use of rendezvous incurs the risk of deadlock.

In C/C++ [Kernighan and Ritchie, 1978, Ellis and Stroustrup, 1990], there is no language support for concurrency altogether, so the programmer has to rely on libraries like Pthreads [Butenhof, 1997] or MPI [Gropp et al., 1994]. As an alternative approach, there are language extensions like OpenMP [Chandra et al., 2001], but like MPI, OpenMP is targeted at scientific computing tasks which do require efficient parallelization but are not concerned much with interactivity.

Java [Gosling et al., 1996] and C# [Microsoft Corporation, 2001] both provide threads with shared memory and synchronization using monitors. Examining the implementation of monitors in Java, however, Brinch Hansen comes to the conclusion that “Java ignores the last twenty-five years of research in parallel programming languages” [Brinch Hansen, 1999]. Starting with version 1.5, Java comes with an additional class-library for common concurrent programming idioms like queues, locks and thread pools.

For some functional languages, concurrent variants have been proposed, too. Examples include Concurrent ML [Reppy, 1999] and Concurrent Haskell [Peyton Jones et al., 1996]. Concurrent Haskell departs from traditional lock-based synchronization by providing transactional variables [Harris et al., 2005]. Modification of those variables can happen only inside a transaction monad, a property which is ascertained by Haskell’s type-class system.

In the engineering community, a popular tool is LabView [Kodosky, 2006], a graphical programming environment that allows the programmer to explicitly define the data-flow between program components. LabView “source code” thus resembles circuit plans, and the execution environment executes independent program fragments concurrently.

The Mozart/OZ system [van Roy, 2005] takes a radically different approach and provides concurrency by means of logic programming and single-assignment variables that may trigger actions when becoming bound to a value.

2.4.1 Join-Calculus Inspired Languages

There are a number of languages that are more or less closely based on the Join Calculus, which is the theoretical model underlying the $m : n$ relation between function bodies and function symbols, where function bodies are executed according to “join rules” between the function symbols. Among those languages are the original Join language [Fournet and Maranget, 1997], JoCaml [Le Fessant, 2001], Funnel [Odersky et al., 1999] as well as the language variants Join Java [von Itzstein, 2004] and Polyphonic C#/C ω [Benton et al., 2004].

Due to their same theoretic background and opposed to our design, these languages all have in common that related function bodies and function symbols must be declared together. To the design of our language, Funnel had the most direct influence, because it also tried to embrace object-oriented as well as functional designs, with concurrency as the central concept. Funnel’s approach, however, is a different one, representing objects as records with function-valued fields, in contrast to our model of methods as open generic functions. Funnel also lacks support for pattern matching: The method of choice to implement algebraic datatypes is a Visitor-Pattern-like double indirection, the data object calling functions on a visitor object, which takes the role of a pattern matching expression.

Since Funnel does not seem to be developed actively any further, and Scala (see below), its successor language, is moving in a rather different direction with no language-level concurrency support, it has been a major motivation for this work to carry on along the lines of Funnel and push the existing concepts a little further.

2.4.2 Actor Languages

Languages based on the Actor model [Hewitt et al., 1973, Agha, 1986] are not among our language’s closer relatives, but since there is a lot of development going on in this branch at the time, an overview of concurrent languages would be incomplete without mentioning the Actor languages.

The most prominent language of this family and one of those that have been specifically designed for concurrent programming is Erlang [Armstrong et al., 1996]. Erlang is based on a model of higher-order actor processes, that themselves behave as functional programs but

can receive messages on their respective *mailboxes* and send messages to other processes. Sending is asynchronous, and selecting what kind of messages to receive is accomplished by specifying a message-pattern that must be matched for the receive operation to succeed.

Despite its origin as a “research language”, Erlang has been actually used in a number of large projects in the telecommunications industry with good success. Assessing the runtime performance of Erlang’s process system shows that concurrency can be implemented more efficiently than with OS threads: Erlang excels not only in process-creation time [Halen et al., 1998], but benchmarks also show that under heavy load, the Erlang-based web server YAWS outperforms Apache significantly [Ghodsi and Armstrong, 2001].

Erlang does have similarities to the language developed in this thesis, e.g. the asynchronous communication model, receiving values based on pattern matching, and the single-assignment semantics of sequential subprograms. In our language, however, there is no dedicated receive operation. Instead, message receivers are defined by setting up join rules which, in addition, may take values at once from multiple channels (= function symbols).

Scala [Odersky, 2006], the successor language of Funnel, has been progressing in Erlang’s direction, too. While Scala itself does not include any language-level concurrency support (other than that provided by the Java VM), there is a sophisticated Scala actors library which brings a lot of Erlang-like functionality to the Java platform. Due to Scala’s flexible syntax and powerful type system, the actor abstractions blend in with the rest of the language quite nicely.

Scala actors can be either thread-based (one thread per actor) or event-based. With event-based actors, like in Erlang and in our language, programmers can benefit from a high scalability without the drawback of an inverted control structure. Thread-based actors, on the other hand, can safely use blocking library code.

Nonetheless, the approach of Scala is very different from ours. With the goal of being a better alternative to Java for general-purpose programming on the Java VM (or C# on the Microsoft CLR), the aim of Scala is to provide better concurrency *on the JVM*, within the constraints set forth by the nature of the existing virtual machines.

2.4.3 Other Influences

Other languages that have been influential to our design despite their lack of dedicated concurrency features are Python [van Rossum, 1994] and Ruby [Matsumoto, 1997] on the one hand, because of their appealing syntax and their handling of generators and blocks.

Both languages are quite *en vogue* at this time, and despite their origin as scripting languages, programmers are building serious software systems in both of them.

On the other hand, languages featuring multimethods have been an influence, among which are CLOS (the Common Lisp Object System) [DeMichiel and Gabriel, 1987], Dylan [Shalit, 1996] and Nice [Bonniot, 2003a, 2005]. Nice has also been used as implementation language for the compiler and interpreter developed in the course of this thesis.

Chapter 3

Core Language and Execution Model

In this chapter, the execution model of the new language is described in detail. To keep the presentation flexible and to the point, and to allow for future extensions of the model, a number of entities such as the nature of runtime values will remain abstract. For the same reasons, it is a simplified core language (presented in Definition 3.2.11) whose semantics are formally defined and in terms of which more advanced language features will be expressed in Chapter 4.

The execution system is described by means of generalized Petri Nets, incorporating ideas from the Join Calculus. The historic development of these two lines of concurrency models is reviewed in Section 3.1, providing the foundation from which, in Section 3.2, our operational Petri Net based semantics is derived.

This operational model is refined in Section 3.3 to a description based on interactive components, which is close to a blueprint for the actual language runtime implementation. Refinements of this model, targeting distributed computation and multi-processor hardware, are discussed in Section 3.4.

Programming Languages		Petri Nets		Join Calculus
Function Symbols	=	Places	=	Channels
Function Bodies	=	Transitions	=	Rules

Figure 3.1: Equivalence of terminology from different domains.

Since there will be references to Petri Net and Join Calculus formalisms throughout this chapter, terminology will be used interchangeably. Figure 3.1 summarizes and identifies the most important terms.

3.1 Background — Formal Models of Concurrency

While programmers have been stuck with multithreading (see Section 1.3.1), theorists have been proposing formal models of concurrency for a long time. In fact, threads were not even designed up front as a means to structure concurrent programs but merely evolved out of operating system processes, dropping the address-space barrier for reasons of reduced communication overhead. The notion of a process in turn was conceived as a hardware abstraction that would allow an operating system to run several programs side by side on a single computer (called *multiprogramming* in those early days).

3.1.1 Petri Nets

One of the early and still most widely used formal concepts is that of Petri Nets [Petri, 1962]. Numerous variants of Petri Nets have been devised over the years, and many have been successful in the industry for describing the behavior of systems and reasoning about a variety of properties such as the absence of deadlock conditions.

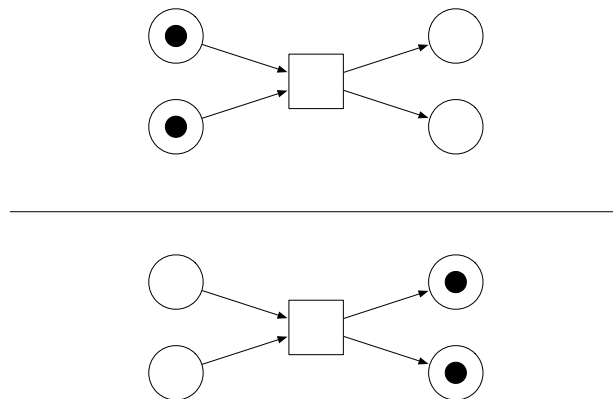


Figure 3.2: Transition in a classic Petri Net.

As a starting point, we consider Place/Transition Nets, which have places with unlimited capacity and arc weights of 1. This is no limitation, since all nets with limited output capacity can be translated to nets without.

Following Reisig [1985] but dropping arc weights and place capacities, a Place/Transition Net N is defined as $N = (P, T, F, M)$ by a set of places P , a set of transitions T (with P and T disjoint), a flow relation $F \subseteq (P \times T) \cup (T \times P)$ and a marking $M : P \rightarrow \mathbb{N}$.

A transition may *fire* if and only if (henceforth: *iff*) tokens are present on all its input places. In the course of firing, one token is removed from each of the transition's input places and one token is written to each of its output places. The firing of a transition thus results in a new marking M' . A simple case of a firing transition is shown in Figure 3.2.

3.1.2 Colored Petri Nets

Advanced Petri Net models, among which Colored Petri Nets [Jensen, 1992] are the most well-known, extend the notion of incomparable tokens to distinguishable, individual data values. A marking M is thus not anymore a mapping from P to numbers of tokens but to multisets of data values: $M : P \rightarrow \mathcal{M}(V)$ for some set of *colors* or data values V . To operate on the data values, the incoming arcs of a transition are annotated with *bindings* that attach an identifier to each incoming value. The outgoing arcs are inscribed with *formulae*, that denote primitive functional operations on the input values. See Figure 3.3 for an example.

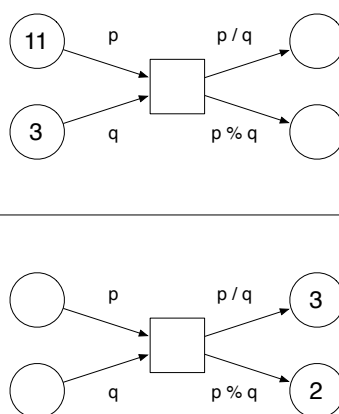


Figure 3.3: Distinguishable tokens and transitions with input bindings and output formulae (e.g. division and modulo) in a Colored Petri Net.

3.1.3 Process Calculi

Apart from Petri Nets, there is the algebraic approach of describing concurrency and computation as calculi: Here, processes expressed as mathematical terms are the basic abstractions and their compositional and behavioral properties are governed by algebraic laws. Fundamental developments in this branch have been the work on Communicating Sequential Processes (CSP) [Hoare, 1985] and the language Occam as well as the Calculus of Communicating Systems (CCS) [Milner, 1982].

In the CCS formalism, processes interact by synchronous (blocking) send and receive operations on local channel symbols. The communication primitives, however, cannot effect channel symbols that have not been declared in the same scope. Therefore, the calculus applies only to a limited range of systems whose communication structure do not change over time, something which is also true of the basic Petri Net models.

The π -calculus [Milner et al., 1992] builds upon the CCS but strives to better accommodate for dynamism. Send and receive operations may take arguments: The argument to the send operation is, not surprisingly, the data to be transferred (which may itself be a channel symbol), and in the receive case, the argument is the name the retrieved data will become bound to. By allowing processes to communicate through channels they have received through other channels, the described system’s communication topology is very flexible. The π -calculus has synchronous and asynchronous formulations. The asynchronous variant is mostly used though, and can also express the synchronous case. Around the π -calculus, a large theoretical framework has been developed for reasoning about concurrent communication patterns.

The π -calculus also forms the basis of the language PICT [Pierce and Turner, 1997]. Experience with PICT suggests however, that the π -calculus may be in a sense too powerful for being suited as a foundation for programming languages. Especially when distributed computing is aimed for, the possibility of receive operations on a single channel being present simultaneously at different locations poses serious implementation challenges.

3.1.4 The Join Calculus

The Join Calculus [Fournet and Gonthier, 1996, 2002] was developed explicitly for describing distributed systems. As opposed to the π -calculus, where any process can receive data on a channel if the channel symbol has been sent to that process before, the Join Calculus prohibits arbitrary receive operations. Instead, groups of channel symbols and the receivers of messages on these channels are defined in a single statement. In spirit of the original semantics that is expressed by means of a “chemical” machine [Berry and Boudol, 1992], these “reaction rules” define the possible ways that messages (envisioned as “molecules”) can react with each other. Defining a rule that has multiple input channels effectively sets up a message rendezvous scheme.

Defining channel symbols and associated reaction rules in the Join Calculus is syntactically reminiscent of function definitions in sequential programming languages. Not surprisingly though, the Join Calculus appears to many as a natural way to extend exist-

ing languages with concurrent features. Along these lines, a number of language variants have been developed, some of which are discussed in Section 2.4.1.

3.1.5 Functional Nets

Functional Nets are a fusion of Petri Nets and the Join Calculus, bringing the dynamic, evolutionary aspect of Join-terms to the so far fixed communication topology of Colored Petri Nets. The issues of dynamism were addressed by Asperti and Busi [1996], who translate Join Calculus ideas such as channel mobility into the Petri Net formalism.

The fundamental observation is that the firing of a transition in a Petri Net (precondition: all incoming places contain a non-zero amount of tokens) and the rewriting of a Join term according to some reaction rule (precondition: all channels on the left-hand side of the rule carry messages) are basically the same operation.

Asperti and Busi describe a hierarchy of net models with increasing dynamism and give encodings of the more powerful models into the simpler ones. In their terms, Mobile Nets extend Colored Petri Nets with the ability to treat places as values, such that the output places of a transition are no longer statically fixed. Dynamic Nets add to this the ability to create new transitions and places during the course of a transition’s firing. The notion of Dynamic Nets contains the Join Calculus as a special case, while Dynamic Nets do not impose locality restrictions, i.e. they do not require places and their associated transitions to be created together.

Inspired by this model, Odersky et al. [1999] also give a reinterpretation of the Join Calculus (without the added features of Dynamic Nets) and propose Functional Nets as an abstraction general enough for functional, imperative and concurrent programming [Odersky, 2000].

With the aim of a *programming* abstraction, Odersky identifies function symbols with Petri Net places and Join Calculus channels, as well as function bodies with Petri Net transitions and Join Calculus rules. The semantics of Functional Nets are not presented anymore in terms of a chemical machine, but as a reduction system, where the basic step of computation consists in rewriting function applications to function bodies, just like in functional programming based on the λ -calculus.

Functional Nets have been employed as a foundation for the language Funnel [Odersky et al., 1999], which also has been influential to this work (see Section 2.4).

3.2 Core Semantics I — Operational Execution Model

In this section, we describe our operational execution system based on a generalized Petri Net model. Section 3.2.1 gives an informal description, with the formal definitions following in Section 3.2.2.

3.2.1 Informal Description

3.2.1.1 From Colored to Dynamic Nets

Starting with Colored Nets as reviewed in Section 3.1.2, our first step is to add mobility, i.e. to make the set of places a subset of the set of data values: $P \subseteq V$. Since the aim is to have the output places of a transition depend on the *values* received from the input places, all the arrows $(T \times P)$ are removed from the flow relation F that had previously been a subset of $(P \times T) \cup (T \times P)$. In general, the concrete output places of a transition cannot anymore be known statically.

Instead, each transition is endowed with a program that contains explicit output statements, where the output places are given by identifiers that are evaluated at runtime. Of course, a place must be known to a program under some identifier if it is supposed to receive any output from that program. In Figure 3.4, the example from Figure 3.3 is recast in this manner. Figure 3.5 gives an actual example of mobility, i.e. a place received as input value serving as an output place.

Continuing the separation of input and output, the firing of transitions is split in two phases. In the first one, the input data is removed from the input places and stored in an *activation record* according to the input bindings. The second step consists in evaluating the transition's program in the context of this activation. So instead of one atomic firing process, there are now two of them¹. Since a transition can have multiple concurrent activations, the activations give rise to a marking on the transitions (see Figure 3.6).

Programs are made even more powerful by providing instructions to set up new transitions. This way, not only the markings of places and transitions change as the network evolves, but the set of transitions can grow, too. Programs can also create new places, so the result of executing a program is in fact a new net. In the terminology of Asperti and Busi [1996], this type of nets is referred to as Dynamic Nets. Figure 3.7 gives an example.

¹Refined models presented later relax the atomicity of program execution (see Section 3.3.1.3).

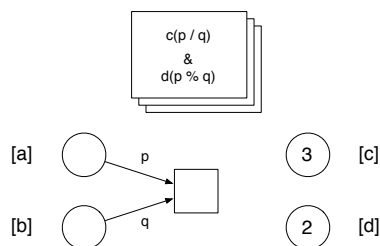
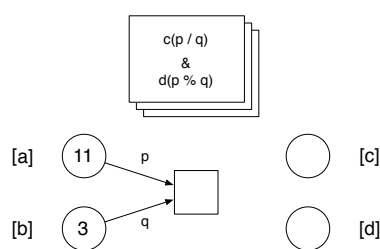


Figure 3.4: Replacing outgoing arrows and output formulae by programs with explicit output statements. Square brackets denote the value of the enclosed identifier.

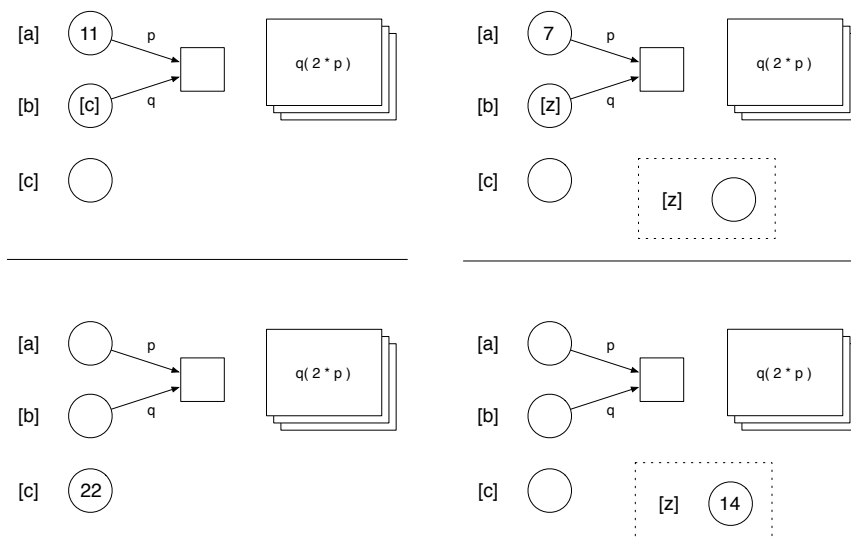


Figure 3.5: Places as values in Mobile Nets: Output places of a transition are no longer statically fixed but may depend on dynamic input (the dashed box indicates that the place [z] is possibly located in a distant part of the net).

3.2.1.2 Patterns and Predicates

With the various kinds of values a place possibly contains, it becomes necessary to somehow restrict the set of values a transition may be activated with. Therefore, each transition has an associated pattern that must be matched before the transition can be activated

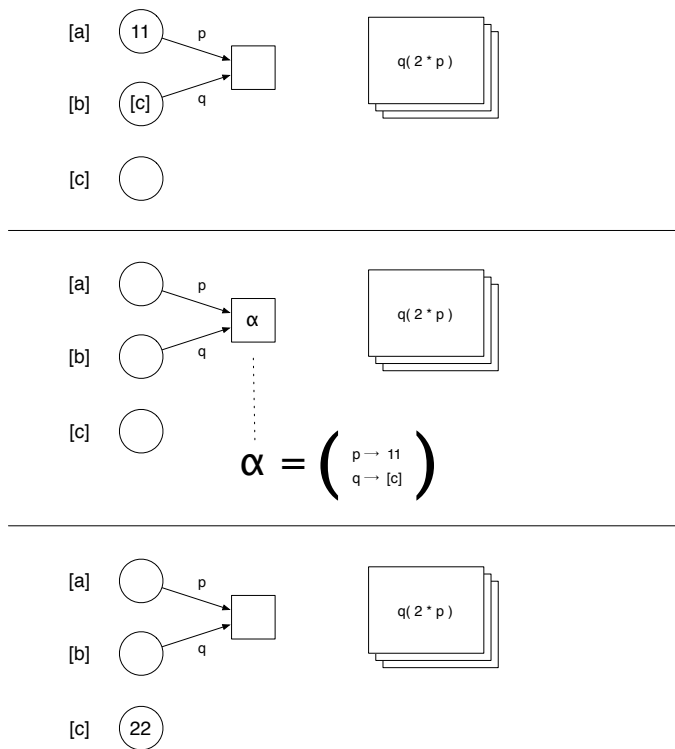


Figure 3.6: Firing of a transition split into activation and program execution.

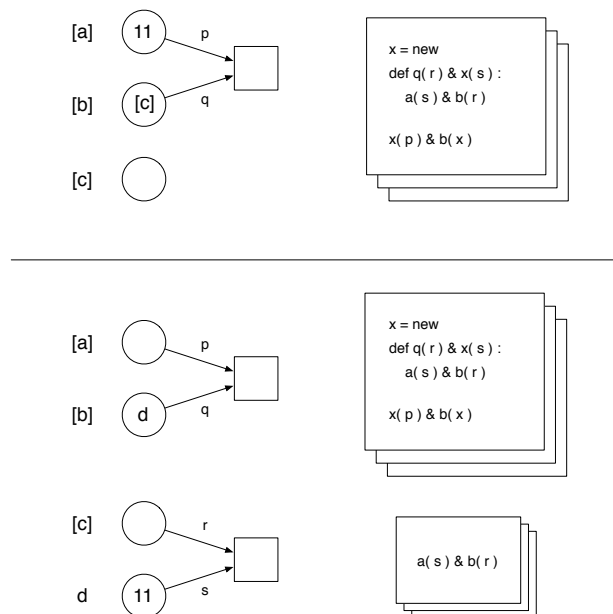


Figure 3.7: Creating new net components in a Dynamic Net. Note that identifiers are lexically scoped and that *d* is the new place *itself* and not an identifier.

(see Figure 3.8). The pattern can thus be seen as a predicate on a set of possible input values, which must hold as a necessary condition for activating the rule. But fulfilling this predicate is not a sufficient condition: The second role the pattern plays is that of a measure on how good a set of values fits a transition's input requirements. If there is a conflict between otherwise enabled transitions, the conflict must be resolved according to the *best-match* principle, i.e. it is one of those transitions that match their input values the most closely that will fire. Below are several examples of this: In Figure 3.9, a very basic case is shown, while Figures 3.10, 3.11 and 3.12 describe the allowed activations in slightly more contrived conflict situations.

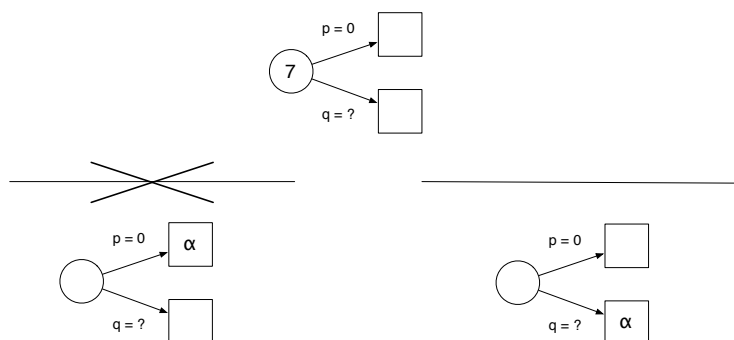


Figure 3.8: Only enabled transitions may be activated (a transition is enabled iff matching values are present on all its input channels).

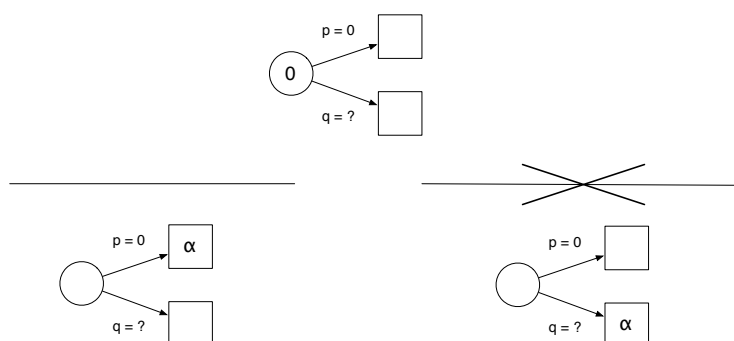


Figure 3.9: An enabled transition may not be activated if its activation would prevent a better matching rule from being activated with overlapping values.

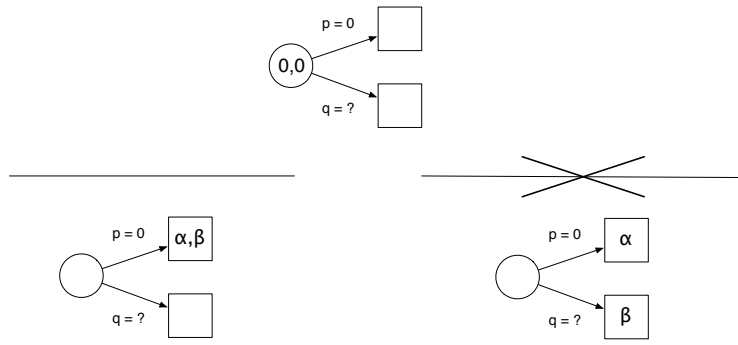


Figure 3.10: Identical values must adhere to the same matching rules, regardless of multiplicities.

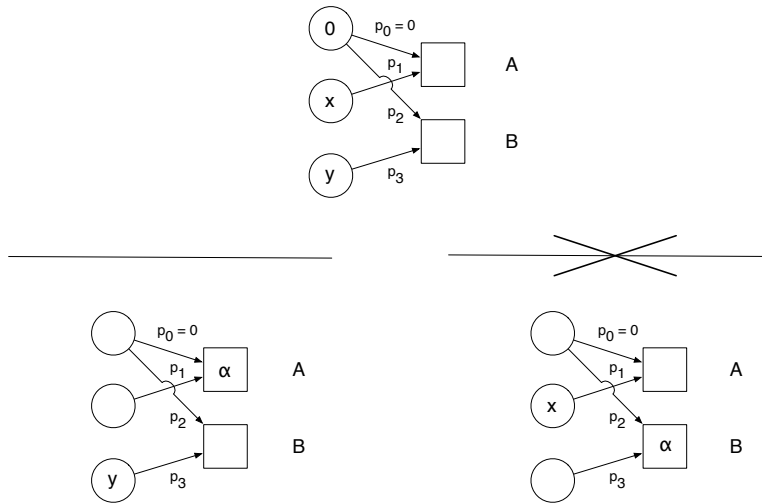


Figure 3.11: Both A and B are enabled, but B may not be activated because A matches $(0, x)$ strictly better than B matches $(0, y)$.

3.2.2 Definitions

3.2.2.1 Notational Conventions

On the way to a formal definition of our execution model, we first elucidate our symbolics. In several places, the notation \overline{x}_i will be used to denote a sequence of values, indexed by i . Note that depending on the context, \overline{x}_i itself is not necessarily a mathematical object but rather a shorthand notation or *macro* for the longer x_0, \dots, x_n (for some n). Used as such, $\{\overline{x}_i\}$ represents a set, $\{\overline{x}_i \mapsto \overline{y}_i\}$ denotes a function, and $\overline{x}_i \in \overline{X}_i$ is the proposition that for all i , x_i is a member of X_i .

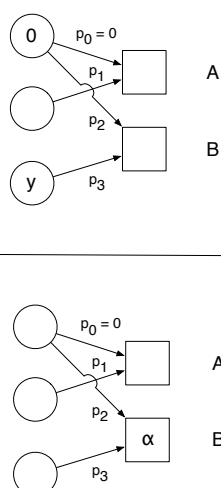


Figure 3.12: Here, B may be activated because activating B does not prevent activation of A since A is not enabled.

Multisets over a set X are assumed to be defined in the standard way, as functions that map the elements of X to a non-negative count of occurrences. Addition and subtraction of multisets is defined as

$$(A \pm B)(x) = A(x) \pm B(x) \text{ for all } x \in X.$$

Defined in this way, the difference of two multisets may contain negative coefficients and thus, it is not necessarily a multiset but a more general *formal sum*. Because of this, multisets will be treated as a special case of formal sums in the following, and the formal sum notation $x_1 + x_3 + 2x_4$ will be used to denote the multiset $\{x_1 \rightarrow 1, x_2 \rightarrow 0, x_3 \rightarrow 1, x_4 \rightarrow 2\}$. The set of all multisets over X is denoted by $\mathcal{M}(X)$.

If not noted otherwise, the lowercase letters i to n will denote non-negative integers.

3.2.2.2 Basic Abstract Data Types

In order to keep the core system as flexible as possible, it has been designed around a number of abstract data types. The concrete structure of values, channels, and identifiers is not important here, so it suffices to assume their existence.

Assumption 3.2.1. *Identifiers*, *Channels* and *Values* are infinite sets where

$$\text{Channels} \subseteq \text{Values}, \quad \text{Identifiers} \cap \text{Values} = \emptyset.$$

Patterns are another abstract data type. In addition to the patterns themselves, there are some required operations that must be defined on concrete pattern objects. These operations determine the set of values a pattern matches and each match's quality.

Assumption 3.2.2. For any integer $n > 0$, $Patterns_n$ is the set of patterns with arity n , and $Matches$ is a partially ordered set, \sqsubseteq being its partial order. There are functions dom and match :

$$\begin{aligned} \text{dom} &: Patterns_n \rightarrow \mathcal{P}(Values^n) \\ \text{match} &: Patterns_n \times Values^n \rightarrow Matches \end{aligned}$$

With the above sets and functions at hand, one can give meaning to the statement that a pattern does or does not match some values and if it does, how close the match is in comparison to others.

Definition 3.2.1. A pattern $\pi \in Patterns_k$ is said to *match* a tuple of values $\bar{x}_i \in Values^k$ iff $\bar{x}_i \in \text{dom}(\pi)$. If a pattern $\pi_A \in Patterns_k$ matches some values \bar{x}_i , a pattern $\pi_B \in Patterns_l$ matches values \bar{y}_i and iff $\text{match}(\pi_A, \bar{x}_i) \sqsubset \text{match}(\pi_B, \bar{y}_i)$, then π_A matches \bar{x}_i *strictly better* than π_B matches \bar{y}_i .

In Section 3.3.2.4, additional criteria for classifying patterns and matches will be introduced.

3.2.2.3 Programs, Rules and Nets

Building upon the abstract data types defined above, a number of concrete types can be assembled. Perhaps the most important objects are programs, expressed in the core-language grammar that is given later in Definition 3.2.11.

Definition 3.2.2. The set *Programs* of core-language programs is the language generated by the core-language grammar.

The evaluation of programs takes place in the presence of an activation record (or environment) that maps identifiers to values.

Definition 3.2.3. The set *Activations* of activation records is the set of all functions

$$\alpha : X \rightarrow Values$$

where $X \subseteq Identifiers$.

In the description of nets, the transitions play an important part. In contrast to traditional Petri Nets, where transitions (like places) are abstract objects, transitions are defined as structured tuples here.

Definition 3.2.4. A *rule* r is a tuple $r = (\alpha, \bar{a}_i, \bar{p}_i, \pi, \theta)$ where for some $n > 0$:

- α is an activation, called the *enclosing* activation
- $\bar{a}_i \in \text{Channels}^n$ is a tuple of channels
- $\bar{p}_i \in \text{Identifiers}^n$ is a tuple of pairwise different binding identifiers
- $\pi \in \text{Patterns}_n$ is a pattern that can determine matches on n -tuples of values
- $\theta \in \text{Programs}$ is a program

The set of all rules is denoted by *Rules*.

As one can see, rules are fairly self-contained objects. The actual definition of nets is thus comparatively simple.

Definition 3.2.5. A Dynamic Functional Net (or net for short) is a tuple $N = (P, T, V, E)$ where:

- $P \subset \text{Channels}$ is the set of places
- $T \subset \text{Rules}$ is the set of rules
- $V : \text{Channels} \rightarrow \mathcal{M}(\text{Values})$ is the *channel marking*, with $V(a) = \emptyset$ for $a \notin P$
- $E : \text{Rules} \rightarrow \mathcal{M}(\text{Activations})$ is the *rule marking*, with $E(a) = \emptyset$ for $a \notin T$

It is important to note that not all possible nets are consistent in an informal sense. The above definition e.g. does not enforce that all channels referenced by some of the net's rules are members of the net's set of places. Therefore, a stricter definition of *legal* nets is given in Definition 3.2.14.

3.2.2.4 Enabledness, Conflicts and Activation

On the way of defining the circumstances under which a rule may be activated, the first step is the following necessary condition:

Definition 3.2.6 (enabled). Let $r = (\alpha, \bar{a}_i, \bar{p}_i, \pi, \theta)$ be a rule and $V \supseteq \{\overline{a_i \rightarrow X_i}\}$ a channel marking. The rule r is *enabled* under V for values \bar{x}_i iff each x_i is present on the rule's i th input channel and iff the tuple \bar{x}_i is matched by the rule's pattern π :

$$\begin{aligned} (X_i - \bar{x}_i \parallel a_i) &\in \mathcal{M}(\text{Values}) \quad \text{for all } i \\ \bar{x}_i &\in \text{dom}(\pi) \end{aligned}$$

where $\bar{x}_i \parallel a_i$ is the *projection* of the input values onto the channel a_i as defined by

$$\bar{x}_i \parallel a_i = \sum_k \delta_{a_i, a_k} x_k$$

and δ is the discrete Kronecker delta.

The projection operation is necessary because a rule's input channels are not required to be unique and thus, multiple values may be required to be present on a single channel for a rule to be enabled. In the case of pairwise different channels \bar{a}_i , the projection $\bar{x}_i \parallel a_i$ obviously simplifies to \bar{x}_i .

But as stated earlier, being enabled is not sufficient for a transition to fire, because the rule may be in a conflict situation with a better matching one. To define the notion of conflict, the weaker notion of static or potential conflict is introduced first.

Definition 3.2.7 (potential conflict). Let $r_A = (\alpha, \bar{a}_i, \bar{p}_i, \pi_A, \theta_A)$ and $r_B = (\beta, \bar{b}_i, \bar{q}_i, \pi_B, \theta_B)$ be rules. The rules r_A and r_B have a *potential conflict* regarding values \bar{x}_i and \bar{y}_i (at channel c) iff $\exists k, l$:

$$\begin{aligned} a_k = c = b_l \\ \bar{x}_i \in \text{dom}(\pi_A) \quad \wedge \quad \bar{y}_i \in \text{dom}(\pi_B) \\ \bar{x}_i[x_k \rightarrow y_l] \in \text{dom}(\pi_A) \quad \vee \quad \bar{y}_i[y_l \rightarrow x_k] \in \text{dom}(\pi_B) \end{aligned}$$

Combining enabledness with a potential conflict then leads to an actual conflict.

Definition 3.2.8 (conflict). Two rules r_A and r_B have a *conflict* regarding values \bar{x}_i and \bar{y}_i under a channel marking V iff r_A, r_B have a potential conflict regarding \bar{x}_i, \bar{y}_i and r_A resp. r_B are enabled with \bar{x}_i resp. \bar{y}_i under V .

Summing up the above definitions, one can now give necessary and sufficient preconditions for possible rule activation.

Definition 3.2.9 (may be activated). If r_A is a rule then r_A *may be activated* with values \bar{x}_i under a channel marking V iff r_A is enabled with \bar{x}_i under V and there exists no rule r_B such that r_A has a conflict with r_B regarding \bar{x}_i, \bar{y}_i under V and r_B 's pattern matches \bar{y}_i strictly better than r_A 's matches \bar{x}_i .

In other words, if $r_A = (\alpha, \bar{a}_i, \bar{p}_i, \pi_A, \theta_A)$ and $r_B = (\beta, \bar{b}_i, \bar{q}_i, \pi_B, \theta_B)$ have a conflict regarding \bar{x}_i, \bar{y}_i , then r_A may not be activated if

$$\text{match}(\pi_B, \bar{y}_i) \sqsubset \text{match}(\pi_A, \bar{x}_i).$$

That is, $\text{match}(\pi_A, \bar{x}_i)$ must be minimal (in terms of the partial order \sqsubset) among the set of matches generated by conflicting rule/value combinations.

With the preconditions of activation set up, it is time to define the effects:

Definition 3.2.10 (activation). Let $N = (P, T, V \cup \{\overline{a_i \rightarrow X_i}\}, E \cup \{r \rightarrow A\})$ be a net and let $r = (\alpha, \bar{a}_i, \bar{p}_i, \pi, \theta) \in T$ be a rule that may be activated under V for some values \bar{x}_i . Then $N' = (P, T, V \cup \{\overline{a_i \rightarrow (X_i - \bar{x}_i || a_i)}\}, E \cup \{r \rightarrow (A + \alpha [\overline{p_i \rightarrow \bar{x}_i}])\})$ is called the (*direct*) *successor* of N under activation of r with \bar{x}_i .

3.2.2.5 Program Syntax and Execution

Programs have remained abstract entities in the description of rule activation. Turning toward the second stage of the execution model, we introduce the actual core language syntax.

Definition 3.2.11 (program syntax). The core language (i.e. the set of core language programs) is defined as the language generated by the following grammar:

$$x, y \in \text{Identifiers}, \quad v \in \text{Values} \setminus \text{Channels}, \quad \Phi_i : \text{Values}^i \rightarrow (\text{Values} \setminus \text{Channels})$$

$$\begin{aligned} \text{ProgramDef} &\rightarrow \text{VarDef}^* \text{RuleDef}^* \text{Call}? \\ \text{VarDef} &\rightarrow x = \text{Expr} \\ &\quad | \quad x = \Phi_i \text{Expr}^i \\ &\quad | \quad x = \mathbf{new} \\ \text{RuleDef} &\rightarrow \mathbf{def} \text{Expr } x \ [\& \text{Expr } y]^* \mathbf{where} \ \text{PatternExpr}: \text{ProgramDef} \\ \text{Call} &\rightarrow \text{Expr } \text{Expr} \ [\& \text{Expr } \text{Expr}]^* \\ \text{Expr} &\rightarrow x \\ &\quad | \quad v \end{aligned}$$

A program consists of a number of single-assignment local variable definitions and a list of rule definitions possibly followed by a call instruction. The right-hand side of a variable assignment can be either a single expression, a primitive operator applied to some expression arguments, or a fresh channel. Since identifiers will be evaluated in the context of an activation, both parameters and local variables can be accessed, as well as those declared in enclosing programs.

A (parallel) call instruction is the application of channels to values, and a rule definition is the setup of another program to be invoked when values matching the given pattern are present on the given channels. Those values become the actual parameters of the program on invocation. The definition of pattern expressions is intentionally left open, as is the definition of patterns itself.

Definition 3.2.12 (program semantics). For each of the different syntactic elements of programs as defined in Definition 3.2.11, a semantics function is defined with a signature as given below (where $\Sigma = \text{Activations} \times \text{Nets}$):

$$\begin{aligned} \mathcal{E} &: \text{Expr} \rightarrow (\text{Activations} \rightarrow \text{Values}) \\ \mathcal{G} &: \text{PatternExpr} \rightarrow (\text{Activations} \times \text{Identifiers}^k \rightarrow \text{Patterns}_k) \\ \mathcal{S} &: (\text{VarDef} \cup \text{RuleDef} \cup \text{Call}) \rightarrow (\Sigma \rightarrow \Sigma) \\ \mathcal{P} &: \text{ProgramDef} \rightarrow (\Sigma \rightarrow \Sigma) \end{aligned}$$

The semantics functions are indeed only partial functions, and program statements (and thus whole programs) outside their domains are considered semantically illegal.

First, the semantics of individual expressions is defined:

$$\begin{aligned} \mathcal{E} \llbracket x \rrbracket (\alpha \cup \{x \rightarrow v\}) &= v \\ \mathcal{E} \llbracket v \rrbracket (\alpha) &= v \end{aligned}$$

Semantics of pattern expressions remain abstract.

Below are the semantics for the different variable declarations:

$$\begin{aligned} \mathcal{S} \llbracket x = e \rrbracket (\alpha, N) &= (\alpha [x \rightarrow \mathcal{E} \llbracket e \rrbracket (\alpha)], N) \\ \mathcal{S} \llbracket x = \Phi \bar{e}_i \rrbracket (\alpha, N) &= \left(\alpha \left[x \rightarrow \Phi \left(\overline{\mathcal{E} \llbracket e_i \rrbracket (\alpha)} \right) \right], N \right) \\ \mathcal{S} \llbracket x = \mathbf{new} \rrbracket (\alpha, (P, T, V, E)) &= (\alpha [x \rightarrow a], (P \cup \{a\}, T, V, E)) \\ &\text{where } a \in \text{Channels} \setminus P \end{aligned}$$

For rule definitions (requiring that the \bar{p}_i are pairwise distinct):

$$\begin{aligned} \mathcal{S} \llbracket \mathbf{def} \overline{a_i} \overline{p_i} \mathbf{where} \pi : \theta \rrbracket (\alpha, (P, T, V, E)) &= (\alpha, (P, T \cup \{r\}, V, E)) \\ &\text{where } r = (\alpha, \overline{b_i}, \overline{p_i}, \mathcal{G} \llbracket \pi \rrbracket (\alpha, \overline{p_i}), \theta) \\ &\text{and } \overline{b_i} = \overline{\mathcal{E} \llbracket a_i \rrbracket (\alpha)}, \quad \overline{b_i} \in \overline{P} \end{aligned}$$

And for call instructions:

$$\begin{aligned} \mathcal{S}[\overline{a_i x_i}] (\alpha, (P, T, V \cup \{\overline{b_i \rightarrow Y_i}\}, E)) &= \left(\alpha, \left(P, T, V \cup \{\overline{b_i \rightarrow (Y_i + \overline{y_i} \| b_i)}\}, E \right) \right) \\ \text{where } \overline{b_i} &= \overline{\mathcal{E}[\overline{a_i}] (\alpha)}, \quad \overline{b_i} \in \overline{P} \\ \text{and } \overline{y_i} &= \overline{\mathcal{E}[\overline{x_i}] (\alpha)} \end{aligned}$$

With the statement semantics set up, the denotation of a program is defined as the functional composition of its statements' semantics:

$$\mathcal{P}[\overline{v_i r_i c}] = \mathcal{S}[c] \circ \mathcal{S}[r_n] \circ \cdots \circ \mathcal{S}[r_0] \circ \mathcal{S}[v_n] \circ \cdots \circ \mathcal{S}[v_0]$$

Summing up the above, we can define the effects of executing an activated rule with regard to the enclosing net.

Definition 3.2.13 (execution). Let $N = (P, T, V, E \cup \{r \rightarrow A\})$ be a net, let $r \in T$ be a rule and let $\alpha \in A$. Then N' with $(\alpha', N') = \mathcal{P}[\theta] (\alpha, (P, T, V, E \cup \{r \rightarrow (A - \alpha)\}))$ is called the (*direct*) *successor* of N under execution of r with α .

The evolution of nets can now be sufficiently described by means of rule activation and program execution. Therefore, the range of nets that will be considered legal can be restricted to those that evolved out of a single program.

Definition 3.2.14 (legal net, generate). A *legal net* N is either the result of executing some program θ on an empty net or it is the successor of some other legal net as the result of activation or execution. In the first case, the program θ is said to *generate* N .

Such legal nets, by construction, satisfy a number of properties.

Observation 3.2.1. *In a legal net, no rule and no activation references channels outside the net. Activations of a rule r only exist because r was activated before and they are exactly r 's enclosing activation extended with r 's binding identifiers mapped to the actual parameters r was activated with.*

3.3 Core Semantics II — Component-Based Execution Model

In Section 3.2, an operational execution model for core language programs has been defined in terms of rule activations and program executions that may or may not happen depending on specified conditions. No laws are given though, that, e.g., would ever *require* any activation to occur, so this choice is entirely non-deterministic.

Non-determinism in general is a desirable property because it yields opportunities for parallel execution. On the other hand, any concrete execution strategy will need to make deterministic choices at times. Furthermore, each source of non-determinism in a formal model increases the system’s state-space and thus makes it harder to reason about its behavior. The aim of this section is thus to devise an execution model that is further geared towards an actual implementation, isolating and making explicit the inherent non-determinism while introducing deterministic rules in some places.

As a cornerstone, it was chosen to maintain the invariant that there are no rules allowed in the net that may be activated but are not—i.e. as soon as a rule may be activated, the system will indeed activate it. This invariant ensures that activation happens in a timely manner and thus eliminates one source of possible starvation.

Section 3.2.1 introduces the new model in some more detail, with the formal definitions following in Section 3.3.2.

3.3.1 Informal Description

3.3.1.1 Separating Concerns

The revised execution model is expressed in terms of software components, interconnected by *FIFO*-channels with unbounded capacities. Components communicate by sending asynchronous messages, and sequences of transferred messages are envisioned as data streams [Broy and Stølen, 2001].

As shown in Figure 3.13, the execution environment is modeled as a system of three components, connected in a feedback loop: the *executor*, the *scheduler*, and the *activator* component.

On system startup, the executor component receives a single program from the “outside”, from which the system’s initial net will be built. After the system has thus started running, the executor receives a stream of program/activation pairs. The executor evaluates the individual program statements in the context of the activation that was input

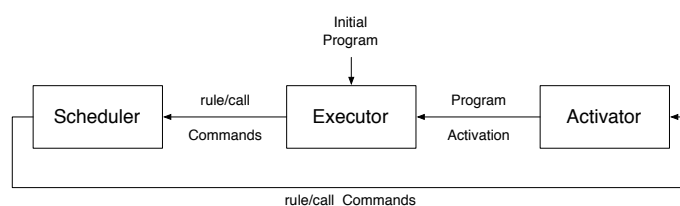


Figure 3.13: Building blocks of the component-based execution model.

together with the program (or the empty activation on initialization). As a result of evaluating rule definition and call statements, the executor outputs a sequence of *rule* and *call* commands that is sent to the scheduler. Note that the executor outputs a *sequence* of commands for each single input program, so the overall output in response to a stream of input programs is a stream of command sequences.

The scheduler component receives this stream of command sequences and multiplexes the sequences into a stream of commands, according to a suitable scheduling strategy. The only requirement is that the order of commands within every individual input sequence must be preserved in the flattened output stream, i.e. no command may “overtake” another one that appeared in a previous position in the same input sequence.

The activator in turn handles the commands from the scheduler one by one, managing the net’s rules and channel marking, as well as identifying rules that may be activated. The output of the activator component is again a stream of activated programs (program/activation pairs) that are fed back into the executor, causing the resulting command sequences to be scheduled for execution by the scheduler component.

As depicted in Figure 3.14, the underlying net’s constituents P, T, V and E are spread among distinct components. The executor manages the set of places P , the activator’s job is to manage the set of rules T and the channel marking V , and the rule marking E manifests itself in the scheduler’s I/O-behavior. This separation of concerns is the basis for our loosely-coupled implementation.

3.3.1.2 Taming Non-Determinism

In order to maintain the system invariant formulated above, the activator must always check for rules that may be activated. After setting up a new rule, it is only this rule that may possibly be activated (though possibly several times), but after writing data values to channels, it is in general sets of rules that must be checked and it is possible that a

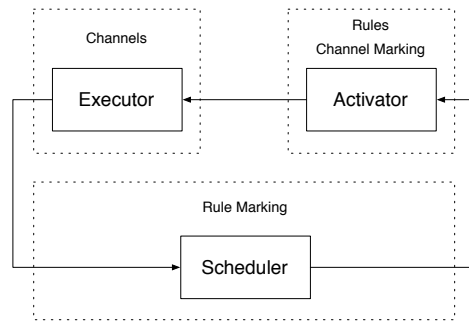


Figure 3.14: Individual components manage different parts of the underlying net.

multi-element set of rules may be activated. Of course, these rules may have conflicts and thus may not always be activated *together*. In any case, the activator must generate a sequence of activated programs until no further rule may be activated (see Section 3.3.2 for the actual strategy used).

With the system invariant preventing delays in the activation of rules, there are only two sources of non-determinism left in the system: the scheduler, which is responsible for selecting activated programs for execution, and the activator, choosing a sequence of rules to be activated from a set of transitions if (and only if) there are more than one that may be activated at the same time.

The selection of rules can easily be influenced by the programmer since it depends directly on the structure of the net and on the patterns assigned². In addition, implementations might choose to introduce some uniformity by selecting rules at random.

In the choice of scheduling strategy, it is certainly desirable that some kind of fairness guarantee can be made, i.e. that the execution of individual programs or program statements is not delayed for prolonged intervals of time. This, however, is not required by the system design. In fact, the other components are completely agnostic to the actual scheduler implementation.

Seen from the opposite direction, the scheduler implementation directly determines the runtime properties of the whole system. If a setup requires strict *FIFO* execution order or a less strict form of immunity against starvation, this can easily be achieved by choosing an appropriate scheduler. The scheduler component will also be the place where load-balancing is handled in the case of an implementation targeting multi-processor hardware (see Section 3.4).

²The activator's choice will actually be biased towards better-matching rules (see Section 3.3.2).

3.3.1.3 Differences from the Petri-Net Model

The alert reader may have noticed that there is one aspect in which the above description contradicts the previous model: In Section 3.2.2, program executions were required to take place atomically. That is, while a program would be executed, no other change might happen to the network, including rule activation.

Here, rule definitions and call statements are individual activator-commands that are executed atomically on their own but not in conjunction with others. That is, evaluation of a rule definition statement may cause an activation of the new rule before the effects of the following rule definitions and the program's call statement are taken into account.

This decision was made deliberately, and although otherwise complicating the model a bit, no technical reasons would prohibit a program-atomic realization (e.g. by means of an additional *commit* command). Atomicity of program execution makes perfect sense in the light of a Petri Net based formalism, but from the perspective of high-level programs (see Section 4.2), situations are likely to arise where programs behave in subtly surprising ways. It can not, however, be demanded from the programmer to be fully aware of each program's net representation, which might not be easily deducible because of syntactic transformations.

As an example consider the programs shown in Figure 3.15. Without being too much concerned about the actual high-level syntax in anticipation of Section 4.2, assume that the value 2 is of type `Int`, which is a subtype of `Real`, and that because of the subtype relation, a pattern `Real x` will match the real number 3.5 strictly better than the integer 2. In the program shown on the left, there are two parallel execution paths: in the first path, the integer 2 is written to the channel `a`, and a real number to the channel `b`. In the second path, there is a very long delay at the beginning, after which a rule is set up on both `a` and `b` and a real number is written to channel `a`. Because of the initial delay, one can assume that the values are already written to the channels when the rule is created (although this is not a guarantee). If statements are always executed independently one after the other, the rule will immediately be activated, consuming values previously written to `a` and `b`. If, however, programs are executed atomically, rule activation may take place only after the second value is written to `a` and the resulting conflict regarding the single value at channel `b` *must* be resolved in favor of the real number at `a`.

This behavior alone might give rise to confusion. The situation gets worse, however, if the program is only slightly modified, such as shown on the right in Figure 3.15. Surprisingly at first sight, the rule definition and the invocation of `a` do not happen in the

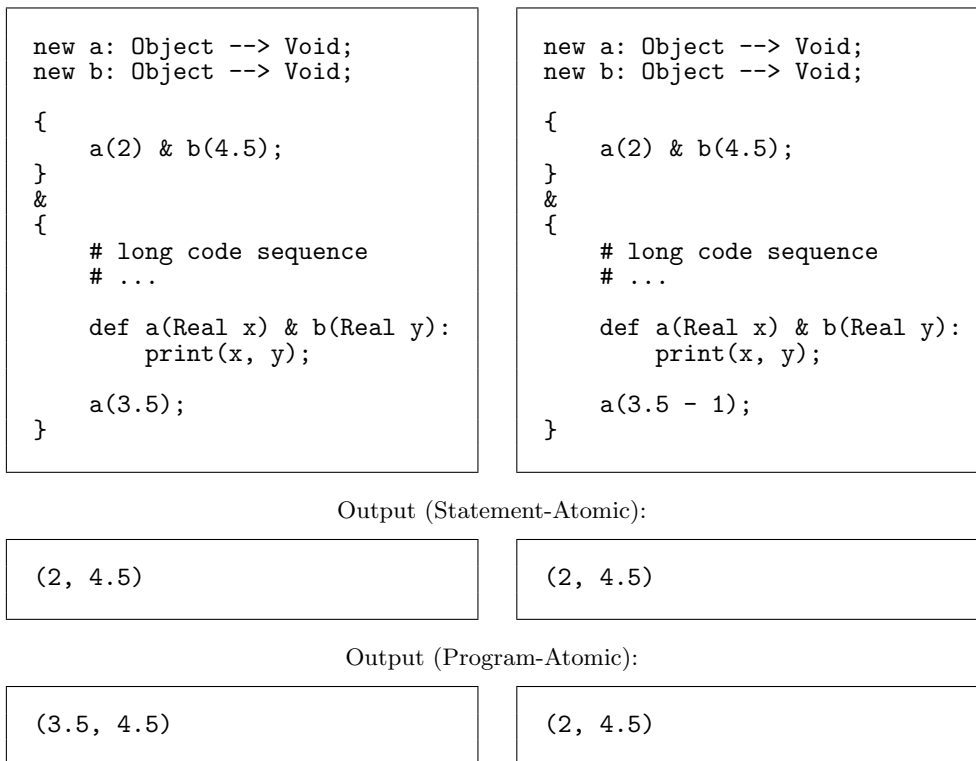


Figure 3.15: Fully-atomic core-program execution causes subtle differences in the behavior of very similar high-level programs (assuming `3.5-1` is translated to a function call).

same core program anymore. The reason is that during the translation from high-level syntax to a core-language representation, the minus operator might be implemented as an additional function invocation³. Since the call to `a` can only happen after the result of the arithmetic expression is evaluated, it ends up in a different core program, and thus the rule definition will *always* be evaluated first.

Explaining why these nearly identical programs consistently produce different output is only possible on the basis of a detailed knowledge of the program translations taking place, and this can not be considered an example of good language design. The semantics of statement-by-statement execution, however, is something every programmer is with no doubt acquainted with.

While other examples (see Figure 3.16) might argue in favor of program-atomicity and indicate that the lack thereof also introduces unexpected behavior, the seeming inconsistency is easily explained in terms of statement-by-statement execution and related programming errors are thus far more easily resolved by the programmer.

³Possibly depending on the compiler's optimization level, which in general should not affect a program's output.

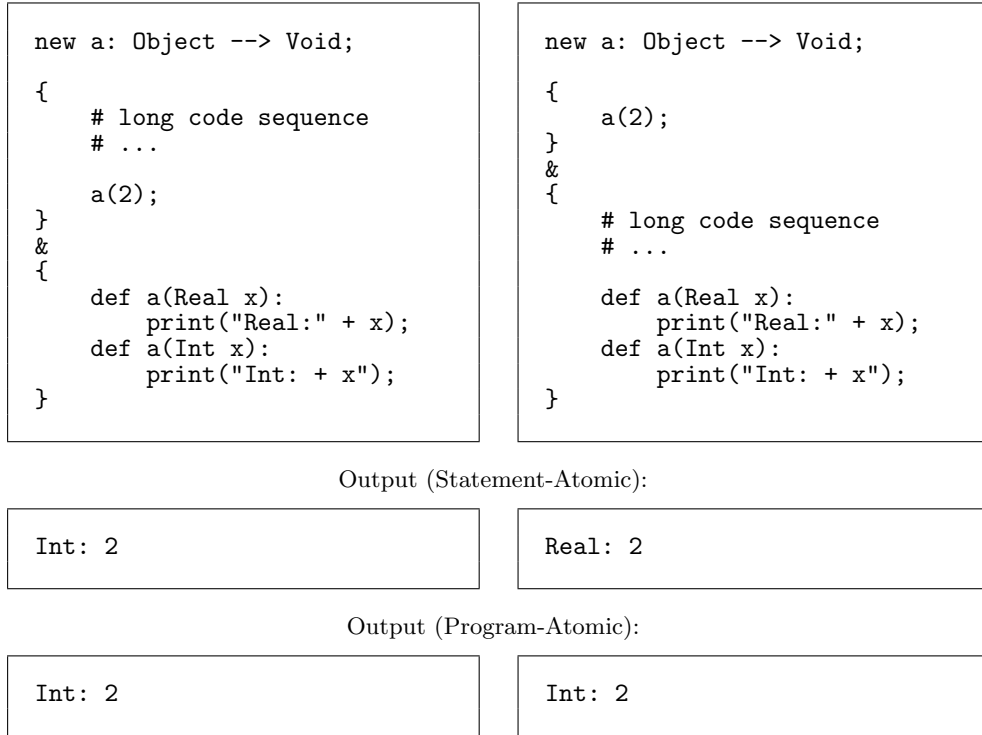


Figure 3.16: Statement-atomic program execution may also cause output that is unexpected at first sight, but behavior can be explained in high-level terms.

3.3.2 Definitions

3.3.2.1 The Scheduler Component

The scheduler's task is to multiplex a stream of command sequences into a flattened stream of commands. The structure of these rule and call commands is defined below:

Definition 3.3.1 (*commands*).

$$\begin{aligned}
 \text{Commands} = & \{ \text{call } (\overline{a}_i, \overline{x}_i) \mid (\overline{a}_i, \overline{x}_i) \in \text{Channels}^n \times \text{Values}^n \} \\
 & \cup \{ \text{rule } (\alpha, \overline{a}_i, \overline{p}_i, \pi, \theta) \mid (\alpha, \overline{a}_i, \overline{p}_i, \pi, \theta) \in \text{Rules} \}
 \end{aligned}$$

Since the actual scheduling strategy is to remain abstract, the definition of the scheduler component is vastly underspecified, i.e. only basic properties are asserted. We use a helper function *filter* (defined in the usual way) to express the property of preserving the individual input sequences' order in the output stream.

Definition 3.3.2 (*scheduler*). Component signature:

$$\text{sched} : (\text{Commands}^*)^\omega \rightarrow \text{Commands}^\omega$$

Behavior (assuming no command x_i appears more than once in the whole input):

$$\text{filter}(\text{sched}(\langle \dots, \langle \bar{x}_i \rangle, \dots \rangle, \{\bar{x}_i\})) = \langle \bar{x}_i \rangle$$

As another consequence of this definition, a scheduler instance must not hold back commands for infinitely large amounts of time. In other words, any scheduler implementation must guarantee that there will always be progress, provided that there is further input.

The assumption that no command appears more than once in the input is no limitation. An implementation can always guarantee this by means of memory addresses and for a formal description, commands can be made unique by “tagging” them with their position in the input.

3.3.2.2 The Executor Component

The purpose of the executor component is to capture the essence (regarding the effect on the underlying net) of a program’s statements by converting the program into a sequence of commands.

Up to now, the semantics of core-language programs (see Definition 3.2.12) have been defined directly in terms of net-transformations. Here, we modify this semantics by capturing the impending transformations in command-form, suitable to hand them over to the activator (the intermediary scheduler will take care of this). The semantics of expressions and patterns carry over unmodified, only those of statements and whole programs are redefined.

Definition 3.3.3 (program semantics). Function signatures for statement and program semantics:

$$\mathcal{S} : (\text{VarDef} \cup \text{RuleDef} \cup \text{Call}) \rightarrow (\Sigma \rightarrow \Sigma)$$

$$\mathcal{P} : \text{ProgramDef} \rightarrow (\Sigma \rightarrow \Sigma)$$

$$\text{where } \Sigma = \text{Activations} \times \mathcal{P}(\text{Channels}) \times \text{Commands}^*$$

Semantics of variable declarations:

$$\mathcal{S} \llbracket x = e \rrbracket (\alpha, P, C) = (\alpha [x \rightarrow \mathcal{E} \llbracket e \rrbracket (\alpha)], P, C)$$

$$\mathcal{S} \llbracket x = \Phi \bar{e}_i \rrbracket (\alpha, P, C) = \left(\alpha \left[x \rightarrow \Phi \left(\overline{\mathcal{E} \llbracket e_i \rrbracket (\alpha)} \right) \right], P, C \right)$$

$$\mathcal{S} \llbracket x = \mathbf{new} \rrbracket (\alpha, P, C) = (\alpha [x \rightarrow a], P \cup \{a\}, C)$$

$$\text{where } a \in \text{Channels} \setminus P$$

Rule definitions ($\overline{p_i}$ pairwise distinct):

$$\begin{aligned} \mathcal{S}[\text{def } \overline{a_i} \overline{p_i} \text{ where } \pi : \theta] (\alpha, P, C) &= (\alpha, P, C \triangleright \text{rule}(r)) \\ &\text{where } r = (\alpha, \overline{b_i}, \overline{p_i}, \mathcal{G}[\pi] (\alpha, \overline{p_i}), \theta) \\ &\text{and } \overline{b_i} = \overline{\mathcal{E}[a_i]} (\alpha), \quad \overline{b_i} \in \overline{P} \end{aligned}$$

Call instructions:

$$\begin{aligned} \mathcal{S}[\overline{a_i} \overline{x_i}] (\alpha, P, C) &= (\alpha, P, C \triangleright \text{call}(\overline{b_i}, \overline{y_i})) \\ &\text{where } \overline{b_i} = \overline{\mathcal{E}[a_i]} (\alpha), \quad \overline{b_i} \in \overline{P} \\ &\text{and } \overline{y_i} = \overline{\mathcal{E}[x_i]} (\alpha) \end{aligned}$$

Program semantics:

$$\mathcal{P}[\overline{v_i} \overline{r_i} c] = \mathcal{S}[c] \circ \mathcal{S}[r_n] \circ \cdots \circ \mathcal{S}[r_0] \circ \mathcal{S}[v_n] \circ \cdots \circ \mathcal{S}[v_0]$$

Just like in the Petri Net based model, the semantics of a program is the functional composition of its statements' semantics. In effect, this concatenates the commands resulting from the statements' evaluation. The only state the modified semantics affects is the net's channels.

On top of this semantics definition, we specify the executor component.

Definition 3.3.4 (executor). Component signature:

$$\text{exec} : \mathcal{P}(\text{Channels}) \rightarrow (\text{Activations} \times \text{Programs})^\omega \rightarrow (\text{Commands}^*)^\omega$$

Component behavior:

$$\begin{aligned} \text{exec}(P) ((\alpha, \theta) \triangleleft S) &= C' \triangleleft \text{exec}(P') (S) \\ &\text{where } (\alpha', P', C') = \mathcal{P}[\theta] (\alpha, P, \diamond) \end{aligned}$$

Receiving an initial program from the “outside” is omitted from the component definition. The desired effect can easily be achieved by a utility prefix-component that prepends the initial program to the executor's input stream, together with an empty activation.

3.3.2.3 The Activator Component

Determining combinations of rules and values that may be activated “from the bottom up” is a complicated task. Conflicts must be detected and resolved in favor of the best matching rule/value combinations, with the additional burden that, in general, there may be multiple ways to do so.

The activator component follows a simpler algorithm and always activates the globally best-matching enabled rule/value combination (or one of them, if there are several). At first sight, this might seem highly inefficient: after all, a huge number of rule/value combinations exists. In practice however, only a fraction of them must be checked.

Before defining the activator behavior (see Definition 3.3.7), some straightforward definitions formalize “best” matches.

Definition 3.3.5 (*best match for channel*). Let $r = (\alpha, \bar{a}_i, \bar{p}_i, \pi, \theta)$ be a rule, V a channel marking, and X the set of all \bar{y}_i that r is enabled with under V . Then \bar{x}_i is a *best match for r* under V iff $\bar{x}_i \in X$ and $\text{match}(\pi, \bar{x}_i)$ is minimal among $\{\text{match}(\pi, \bar{y}_i) \mid \bar{y}_i \in X\}$.

In addition to the best values for a given rule, the best overall rule/value combination is of interest.

Definition 3.3.6 (*global best match*). If R is a set of rules, V a channel marking, and \bar{x}_i a best match under V for a rule $r_A \in R$, then the pair \bar{x}_i, r_A is called a (*global*) *best match* in R under V iff there exists no rule $r_B \in R$ with a best match \bar{y}_i under V such that r_B 's pattern matches \bar{y}_i strictly better than r_A 's matches \bar{x}_i .

It is easily seen that the definition of a global best match is just the definition of “may be activated” without the conflict.

Observation 3.3.1. *If \bar{x}_i, r is a best match in R under V then r may be activated with \bar{x}_i under V .*

As mentioned above, the activator modifies its internal state and produces output as a response to rule or call commands. The actual component definition is given below:

Definition 3.3.7 (*activator*). Component signature:

$$\begin{aligned} \text{act} : & (\text{Channels} \rightarrow \mathcal{M}(\text{Values})) \times (\text{Channels} \rightarrow \mathcal{P}(\text{Rules})) \times \mathcal{P}(\text{Rules}) \\ & \rightarrow \text{Commands}^\omega \rightarrow (\text{Activations} \times \text{Programs})^\omega \end{aligned}$$

Reaction to call command:

$$\begin{aligned} & \text{act} \left(V \cup \{\overline{a_i \rightarrow X_i}\}, A \cup \{\overline{a_i \rightarrow R_i}\}, \emptyset \right) (\text{call}(\bar{a}_i, \bar{x}_i) \triangleleft C) \\ & = \text{act} \left(V \cup \{\overline{a_i \rightarrow (X_i + \bar{x}_i \| a_i)}\}, A \cup \{\overline{a_i \rightarrow R_i}\}, \bigcup \bar{R}_i \right) (C) \end{aligned}$$

Reaction to rule command:

$$\begin{aligned} & \text{act} \left(V, A \cup \{\overline{a_i \rightarrow R_i}\}, \emptyset \right) (\text{rule}(\alpha, \bar{a}_i, \bar{p}_i, \pi, \theta) \triangleleft C) \\ & = \text{act} \left(V, A \cup \{\overline{a_i \rightarrow (R_i \cup \{r\})}\}, \{r\} \right) (C) \\ & \quad \text{where } r = (\alpha, \bar{a}_i, \bar{p}_i, \pi, \theta) \end{aligned}$$

Internal work loop (with activation):

$$\begin{aligned} & \text{act} \left(V \cup \{ \overline{a_i \rightarrow X_i} \}, A, R \right) (C) \\ &= (\alpha [\overline{p_i \rightarrow x_i}], \theta) \triangleleft \text{act} \left(V \cup \{ \overline{a_i \rightarrow (X_i - \overline{x_i} \| a_i)} \}, A, R' \right) (C) \end{aligned}$$

where R' is the restriction of R to rules that are enabled for some $\overline{y_i}$ under $V \cup \{ \overline{a_i \rightarrow X_i} \}$ and $r = (\alpha, \overline{a_i}, \overline{p_i}, \pi, \theta) \in R'$, with $r, \overline{x_i}$ a best match in R' under $V \cup \{ \overline{a_i \rightarrow X_i} \}$

Internal work loop (without activation):

$$\begin{aligned} & \text{act} (V, A, R) (C) = \text{act} (V, A, \emptyset) (C) \\ & \text{if no rule in } R \text{ is enabled for any } \overline{y_i} \text{ under } V \end{aligned}$$

As one can see, the component's direct reaction to any command is to pass all the rules that are potentially affected by the command (and thus must be checked for activation) to an internal work loop, which in turn repeatedly activates one best match until there are no enabled rules left.

Observation 3.3.2. *The activator component maintains the system invariant, i.e. if there is no rule that may be activated before receiving a command, there will again be no rule that may be activated after completely handling the command.*

Moreover, the system invariant allows a further reduction of the search space for a best match when handling call commands: in addition to the set of rules, which is already tightened somewhat in the above definition, the values present on the channels need consideration. If there was no enabled rule before receiving the command, then no rule can suddenly become enabled with values that have *all* been present before. Utilizing this observation, implementations can further optimize the work loop.

3.3.2.4 A Taxonomy of Patterns

Another source of optimization is knowledge about the individual patterns. In this section, some properties that are of interest are given a closer look.

The most useful patterns, from an optimization standpoint, are those that are indifferent, i.e. those that do not care about the actual values matched.

Definition 3.3.8 (indifferent). A pattern $\pi \in \text{Patterns}_k$ is called *indifferent* iff there exists $\mu \in \text{Matches}$ (called the *tolerance* of π) such that $\mu = \text{match}(\pi, \overline{x_i})$ for all $\overline{x_i} \in \text{dom}(\pi)$.

When facing a rule with an indifferent pattern, it suffices to search for *any* matching tuple of values in order to find a best match.

Observation 3.3.3. *Let $r = (\alpha, \overline{a}_i, \overline{p}_i, \pi, \theta)$ be a rule, and let π be indifferent. If r is enabled with some \overline{x}_i under a channel marking V , then \overline{x}_i is a best match for r under V .*

Another category are the separable patterns, which can be matched individually for each channel and do not pose inter-channel constraints.

Definition 3.3.9 (separable). A pattern $\pi \in \text{Patterns}_k$ is called *separable (into $\overline{\pi}_i$)* iff there exist k patterns $\pi_i \in \text{Patterns}_1$ and a component-wise monotonic function $f : \text{Matches}^k \rightarrow \text{Matches}$ such that

$$\begin{aligned} \overline{x}_i \in \text{dom}(\pi) &\iff \overline{x_i} \in \text{dom}(\pi_i) \\ \text{match}(\pi, \overline{x}_i) &= f\left(\overline{\text{match}(\pi_i, x_i)}\right). \end{aligned}$$

Observation 3.3.4. *All patterns $\pi \in \text{Patterns}_1$ are trivially separable.*

Observation 3.3.5. *Let $r = (\alpha, \overline{a}_i, \overline{p}_i, \pi, \theta)$ be a rule, and let π be separable into $\overline{\pi}_i$. If for all i , x_i is a best match for $r_i = (\alpha, a_i, p_i, \pi_i, \theta)$ under a channel marking V , then \overline{x}_i is a best match for r under V .*

A very desirable property is the comparability of patterns regardless of any values.

Definition 3.3.10 (more precise). Let $\pi_A \in \text{Patterns}_k$ and $\pi_B \in \text{Patterns}_l$ be patterns. Then π_A is called *more precise than π_B* iff π_A matches \overline{x}_i better than π_B matches \overline{y}_i for all $\overline{x}_i \in \text{dom}(\pi_A)$ and $\overline{y}_i \in \text{dom}(\pi_B)$.

Observation 3.3.6. *Let π_A, π_B be indifferent patterns and let μ_A, μ_B be their tolerances. Then π_A is strictly more precise than π_B iff $\mu_A \sqsubset \mu_B$.*

Observation 3.3.7. *Let r_A, r_B be rules and let the pattern of r_A be strictly more precise than the pattern of r_B . If \overline{y}_i, r_B is a best match under a channel marking V , then r_A cannot be enabled under V for any \overline{x}_i .*

That is, if a pattern's precision can be determined easily (e.g. computed statically by the compiler), rules can be pre-sorted according to their pattern's precision before searching the values for a best match. If a match is found for a highly precise pattern, the search can stop and safely ignore the less precise rules.

Time Complexity of Pattern Matching

Determining best matches for a rule in the most general case of patterns amounts to checking every combination of input values, which requires $O\left(\prod |X_i|\right)$ operations, where X_i is the set of values present on input channel i . In other words, the time complexity is exponential in the number of the rule's channels, and proportional to each channel's count of values.

If, however, the rule's pattern is separable, each input channel can be checked independently, thus requiring only $O\left(\sum |X_i|\right)$ operations. Using efficient data structures that facilitate value lookup for the specific pattern, the term $|X_i|$ may even be reduced to logarithmic time. Weight-balanced trees, for example, have been shown to provide good speedup in Colored Petri Net implementations [Mortensen, 2001].

Indifferent patterns, by contrast, have the same worst-case complexity as their non-indifferent counterparts, whether separable or not. If there is no match, all possible combinations of values have to be checked nonetheless. But in case there is a match, the search can stop immediately, as any match must also be a best match.

Pattern Examples

With the above classifications on patterns, a number of common cases in high-level programs can be implemented efficiently. Writing a constant number of values to a channel that no multi-channel rule is defined on, for example, which is the equivalent of invoking an overridden or overloaded method, can always be completed in constant time since only new values need to be considered in finding a best match.

As another example, consider the rule definition

```
def a(x) & b(y):
{
    ...
}
```

The pattern is both indifferent and separable. Therefore, searching for a best match amounts to checking that both channels contain any value. Hence, writing to channel `a` or `b` is always a constant time operation.

If modified slightly, the rule definition's pattern is still indifferent but no longer separable:

```
def a(x) & b(x):  
{  
  ...  
}
```

Writing to either `a` or `b` is now linear in the number of values present on the other channel.

3.4 Advanced Refinements

The component-based execution model, as described so far, is sufficiently abstract to allow for interesting refinements, two of which are sketched below.

3.4.1 Distributed Model

In the context of interactive systems, distributed implementations that span multiple otherwise independent computing sites are of interest. Conceptually, each computing site possesses its own triplet of executor, scheduler and activator components. The computing sites' executors, however, are now required to have disjoint channel reservoirs to create new channels from, and to mark each channel with its "home" site on creation.

In addition, there are links from every executor to all the other sites' schedulers, as shown in Figure 3.17. This way, commands are always handled by the affected channels' home sites. When issuing a call command that would affect channels with different home sites, the executor just splits the command. Rule commands, by contrast, are not allowed to affect channels with different home sites.

With this simple extension, both remote procedure calls (RPC) and mobile code are covered: Writing to a remote channel moves the argument values to the channels home site, possibly causing rule activation there, while setting up a rule on some distant channels will transfer the rule-body code to the remote site.

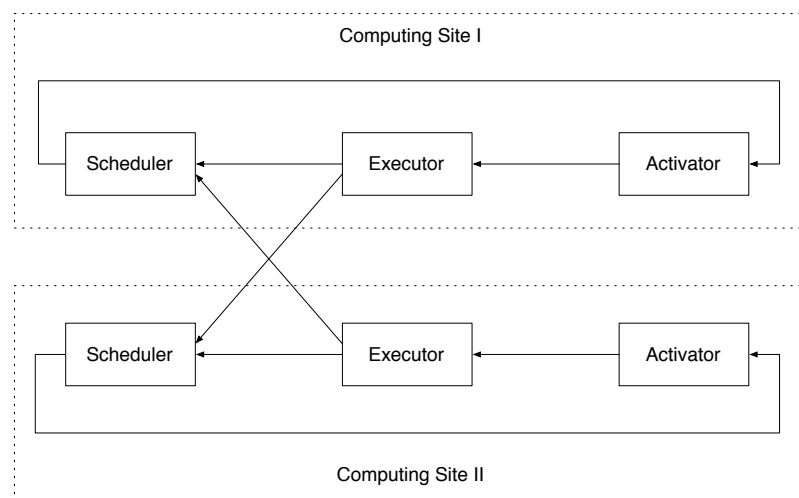


Figure 3.17: Distributed implementations target commands to the affected channels' home sites.

3.4.2 Concurrent Model

Another possible refinement is concerned with the efficient utilization of multi-processor hardware. In a first refinement step, the system is still envisioned as three components, but with several independent computation loops operating in parallel (see Figure 3.18).

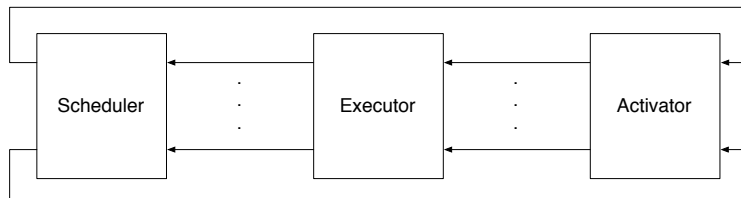


Figure 3.18: Black-box view of concurrent implementation.

By further refinement (see Figure 3.19), the “parallelized” components’ boundaries are dissolved where possible, making apparent the inherent computational dependencies. For efficiency reasons, the scheduler and the executor components exchange their positions: Now it is entire programs that are scheduled, not individual commands.

The single parallelized scheduler can be refined to a number of run queues (one for each processor), that are operated on in a work-stealing manner [Blumofe and Leiserson, 1999], which also is the method of choice in several other toolkit or language (variant) implementations such as Cilk [Blumofe et al., 1996]. The principle is that each activator pushes newly activated programs onto its own associated run queue. If the run queue gets empty at some time, an attempt is made to “steal” a queued activation from one of the other run queues, which is selected at random. That is, as long as each processor is producing enough new activations to prevent its run queue from draining, no shared data needs to be accessed. Even the load-balancing case can be implemented efficiently using only lock-free data structures [Blumofe and Papadopoulos, 1998].

While the executors do not need to share any state, the activators certainly do. In the refined model, the shared state is factored out into a single *net* component, that is accessed by all activators. Serializing access by some kind of locking strategy, although self-suggesting, may not necessarily be a good choice. A better approach would be to employ an optimistic transactional strategy, attaching version numbers to each channel and checking for concurrent modifications upon committing changes to a channel’s data, in which case the transaction would have to be retried. The reason why an optimistic approach seems far more promising than explicit locking is the conjecture that in many cases, writing to a channel will cause immediate activation of some rule (like in a regular

function call), such that no shared state is actually modified and hence, no locking is necessary.

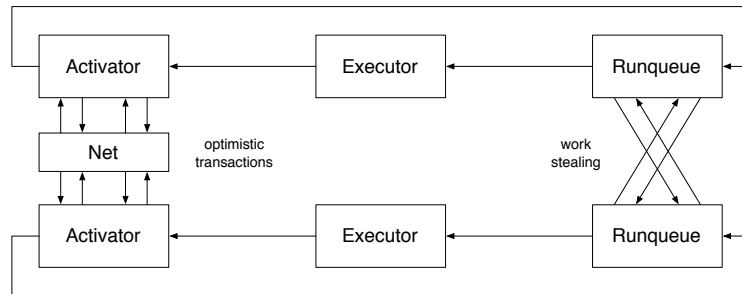


Figure 3.19: Lock-free concurrent refinement: Net state is managed using software transactions and computation load is balanced between processors by work stealing.

Chapter 4

High-Level Language

While the previous Chapter 3 was concerned with the formal semantics of programs expressed in a small core language, it is the purpose of this chapter to clarify how higher-level language constructs such as those introduced in Chapter 2 can be expressed in terms of this model.

In Section 4.1, we revisit those aspects of the system that have remained abstract so far and describe the instantiation that is used by the high-level language. Section 4.2 is concerned with the language syntax, explaining the translation of high-level syntax to the core language.

As opposed to the description of the core execution model, the treatment of the instantiation and the syntax encoding will be less formal. It was felt that a rigorous treatment would not offer much benefit for the reader, while being considerably harder to follow than a by-example explanation of the main concepts.

4.1 Instantiation of the Execution Model

In this section, we will reconsider the abstract data types of the core execution model (as defined in Section 3.2.2.2) and the syntactic entities that have been left open in the definition of the core language syntax and semantics (see Section 3.2.2.5).

4.1.1 Values

In the specification of the execution model, the existence of an abstract set *Values* of possible runtime values is assumed. Since the reference implementation is designed to run on the Java platform, *Values* may actually contain any Java object. In addition to the regular Java class libraries among which are basic types like `Integer` or `String`, the language runtime provides specialized classes for the following data types:

- The unit value `()`
- Immutable tuples of a fixed length
- Immutable records with a fixed set of named fields
- Channels

In addition, we treat a record as an object of a user-defined type t if the record has a field `_type` with value t . User-defined types may either be Java class-objects or records containing a field `_sym`, which identifies the channel used to create new objects, and an optional field `_super`, which defines the type's super type.

With rule definition and call statements, as well as channel creation expressions in variable declarations, there is explicit syntactical support for channel-related operations in the core language (see Definition 3.2.11). For values other than channels, there are no predefined operations but the core language allows any built-in function Φ_i to occur on the right-hand side of a variable declaration, provided that Φ_i maps tuples of values with a fixed arity i to a single result value.

$$\text{VarDef} \rightarrow x = \Phi_i \text{ Expr}^i$$

where $\Phi_i : \text{Values}^i \rightarrow (\text{Values} \setminus \text{Channels})$

The core language definition excludes channels from the set of possible return values. This is a technical restriction to prevent premature access to a function symbol that may later be the result of evaluating a `new`-expression. In the actual implementation, evaluating `new` will always return a fresh object, so this restriction can in fact be dropped.

In order to facilitate the assembly and access of structured data, several such built-in operations are provided:

- Tuple formation
- Tuple access at a statically defined numeric index
- Record extension (contains record formation as a special case)
- Record field access

An example of the corresponding syntax is shown in Figure 4.1.

```
tuple   = 7, "foo";
seven   = tuple[0];

record1 = {} ++ name: "John", age: 42;
record2 = record1 ++ job: "BigBoss", salary: 1.0e6;

john    = record2.name;
million = record2.salary;
```

Figure 4.1: Construction and access of structured data (tuples and records).

There is also a *tolerant* variant of record field access (written `a.b?`), which denotes the field `b` of record `a` if the field exists and the empty record if not. The purpose of this variant is explained in Section 4.2.3.

In addition to the structured data operations, there is a built-in function to type-check another expression. For example, `check Int x` will evaluate to `x` iff `x` is of type `Int` and produce a runtime error otherwise.

Arithmetic and logical operations, however, are not implemented as built-in functions. Because of overloading reasons (`String + String` vs. `Double + Integer`), arithmetic operations are implemented as calls to native functions.

4.1.2 Patterns

The relevant items in the execution model are the set $Patterns_n$, $n > 0$ of patterns, and the functions `dom` and `match`, as well as the partial order \sqsubseteq on the codomain of `match`.

$$\begin{aligned} & Patterns_n, Matches, \sqsubseteq \\ \text{dom} : & Patterns_n \rightarrow \mathcal{P}(Values^n) \\ \text{match} : & Patterns_n \times Values^n \rightarrow Matches \end{aligned}$$

In the core language, pattern expressions appear inside rule definition statements, with their own definition remaining abstract:

$$\begin{aligned} RuleDef &\rightarrow \mathbf{def} Expr\ x\ [\& Expr\ y]^* \mathbf{where} PatternExpr: ProgramDef \\ & PatternExpr \rightarrow ? \end{aligned}$$

We instantiate pattern expressions as conjunctions of atomic constraints, with empty conjunctions denoting the *match-all* pattern. A constraint can either assert an equality, such as `x = 7`, or a structural property, such as `x @ Tuple(2)`. In a rule definition, an example pattern expression might look as follows:

```
def a(x) where x @ Tuple(2), x[0] == 7:
{
  ...
}
```

In any case, the left hand side of a constraint specifies what to match and the right hand side defines what to match against. On the left, an identifier `x` that occurs as a binding in the pattern's enclosing `def` expression denotes the value received on the corresponding channel. Nested tuple indexing and record field access is also allowed here, like in value expressions (`x[i]` resp. `x.key`). The number of tuple or record access operations on the left hand side of a constraint is referred to as the *depth* of the constraint.

On the right hand side, equational constraints may refer to atomic values (`x = 3`), to other parts of the values that are tested for matching the pattern (`x[1] = y[0]`) or to variables defined outside the pattern (`x = $y`).

Structural constraints may either demand that the value in question is an *i*-tuple (`x @ Tuple(i)`), a record containing a field `key` (`x @ Record(key)`), or an object of a

user-defined type ($x @ \text{Object}(\text{type})$) where type may be a constant value or an identifier $\$t$ that refers to the scope outside the enclosing def statement.

The so-far abstract semantics of pattern expressions have been defined as follows:

$$\mathcal{G} : \text{PatternExpr} \rightarrow \left(\text{Activations} \times \text{Identifiers}^k \rightarrow \text{Patterns}_k \right)$$

$$\mathcal{G}[\pi] (\alpha, \bar{p}_i) = ?$$

In evaluating a pattern expression, outside identifiers such as $\$y$ are looked up in the given activation record α and replaced by the value found. Apart from that, the internal pattern representation (the set Patterns_k) is very similar to the structure of pattern expressions in the core language.

4.1.3 Matching

For most types of constraints, the requirements they impose on matching values should be clear from the description above. If there are no constraints in a pattern, the pattern will match any value. Regarding $x @ \text{Object}(t)$ constraints, not only objects whose direct type is t should match, but also objects that are of a subtype of t . That is, the chain of an object's `__super` links must be followed in order to determine if an object matches. Deciding whether two types are the same is done by comparing their `__sym` fields for equality.

In the evaluation of how good a pattern matches a set of values, one is interested in keeping the dependency on the actual values to a minimum (see Section 3.3.2.4). Therefore, we define a number of criteria after which we order patterns statically, shown from top to bottom in decreasing order of importance. On each level, a greater number of constraints means a more precise pattern:

1. Number of channels
2. Number of non-separable constraints (e.g. $y = x$)
3. Number of value constraints (e.g. $x = 3$ or $x == \$y$)
4. Number of other separable equality constraints (e.g. $x[1] = x[0]$)
5. Number of $x @ \text{Object}$ constraints
6. Number of $x @ \text{Tuple}$ and $x @ \text{Record}$ constraints

Only when two or more patterns are equally precise according to this static ordering, dynamic ordering must take place regarding the actual values. In this case, the sum of `...super` steps between the actual type of `x` and the type `t` in all `x @ Object(t)` constraints is the deciding factor, where a smaller number of steps means a more precise pattern.

4.2 High-level Syntax Encoding

We introduce the higher-level language constructs in a bottom-up fashion, building upon the core language as specified in Definition 3.2.11 on page 47 and successively adding more expressive features.

4.2.1 Sequential and Parallel Composition

One important aspect of the core language is that function call statements may occur only at the very end of a program. For a concrete programming language, this is of course a serious limitation, because there is no notion of returning a value from a function or of calling a number of functions in sequence. The solution is to convert the high-level program to continuation-passing style by introducing an additional function argument (the “continuation”) that represents the *remainder* of the computation [Appel and Jim, 1989].

<pre> new fun: Unit --> Int; def fun(): { return(7); } x = fun(); ... </pre>	<pre> new fun: (Int-->Void) --> Void; def fun(return): { return(7); } new c: Int --> Void; def c(x): { ... } fun(c); </pre>
--	---

Figure 4.2: Continuation-passing transform of a function call (left: high-level syntax, right: translation).

Figure 4.2 shows how function calls that appear on the right hand side of a variable declaration are transformed in this way. The intuitive meaning of the high-level program is to bind the value returned from the invocation of `fun` to the local name `x` and then continue with the next statement after the declaration. During the continuation-passing (CPS) transform, the implicit `return` function becomes an explicit parameter with a `Void` return type and an argument type equal to the return type of the untransformed function. The function symbol and function body declarations change accordingly. The call site is also transformed, supplying a newly created function as continuation parameter whose function body contains the remainder of the computation (i.e. all statements after the

declaration in question) and x as the name of its formal parameter. In this way, the behavior of the code is the same as before (the function `fun` is invoked, and upon invoking `return`, the function result is bound to the name x), but the function call appears in a tail position, making the code a legal core language program.

Function calls that appear in the middle of a sequential block without being part of a declaration (such as `print(...)` calls) may be handled in the same way, using a fresh name in place of the identifier from the declaration.

Since we are designing a concurrent language, the parallel composition of statement sequences is also of interest. Using the transformation shown in Figure 4.3, a parallel composition of sequential blocks may again be part of a statement sequence.

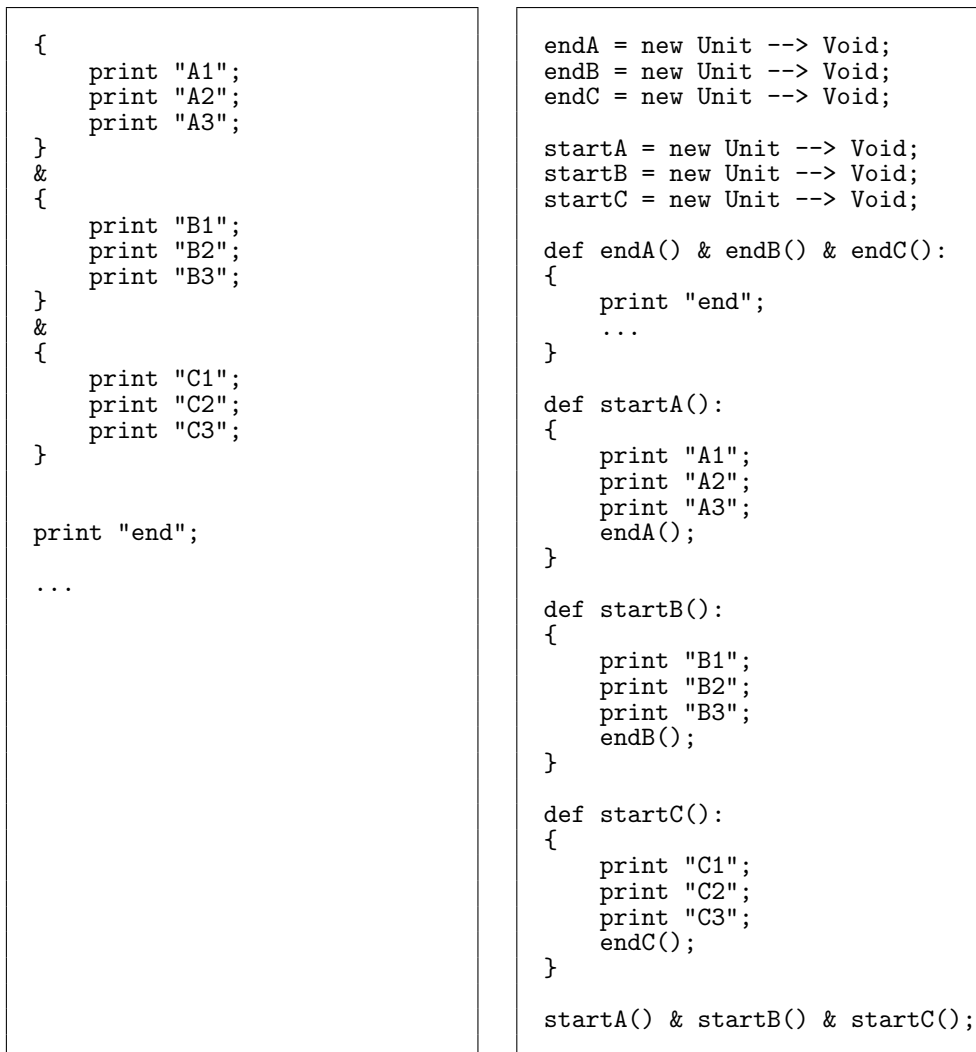


Figure 4.3: Transformation of a parallel composition of sequential blocks (transformation of inner statement sequences not shown).

In this transformation, we make use of the fact that the core language allows us to issue several function calls in parallel and that we can define function bodies on multiple function symbols. By turning each of the parallel blocks into a function of its own, we can easily start parallel execution of the blocks by calling the resulting functions in parallel. In order to be able to continue with the next sequential statement after the parallel blocks are done, we move the remainder of the computation into another function body, which is defined across several function symbols, one for each block. Inserting one of these function symbols at the very end of each block yields the desired behavior, as the continuation is activated only after all parallel blocks have run to completion.

Summing up the results of this section, we have extended the core language in the following way:

- Function calls may appear on the right-hand side of variable declarations.
- Parallel statements may not only contain function calls, but also sequential blocks.
- Sequential blocks in turn may contain function calls and parallel statements at any position, not just at the very end.

4.2.2 Value Expressions

In the core language, any value object that is not a channel may appear as a constant expression. The high-level syntax provides literals for the most important datatypes:

- The unit value (`()`)
- Boolean values (`true`, `false`)
- Integers
- Floating point numbers
- Strings

Another restriction of the core language is that only variables and constant values may be arguments to function calls or to built-in primitives. For high-level programs, this restriction does not exist (see Figure 4.4). All intermediate values that are results of composite expressions are assigned to fresh auxiliary variables during the translation phase. On the same run, an α -conversion takes place that assigns each variable a unique name. This way, syntactic blocks of statements inside other statement sequences are absorbed into the outer ones.

<pre>x = f(g(h(7, "foo"))); { x = 9; } x = 1 + 2 * x; print(x);</pre>	<pre>x0 = 7, "foo"; x1 = h(x0); x2 = g(x1); x3 = f(x2); x4 = 9; x5 = 2 * x3; x6 = 1 + x5; x7 = print(x6);</pre>
---	---

Figure 4.4: Intermediate values are given distinct names. Sequential blocks inside statement sequences are thus absorbed.

In Section 4.1, we have endowed the core language with some built-in functions that facilitate the assembly and access of structured data, namely tuples and records. Not all of these built-in functions are available for use by high-level programs in pure form. Tuple formation is available but no indexed tuple access. Extension of arbitrary records with new fields is not possible, only record *formation*, i.e. extension of the empty record. Record access by field is available, though. The reason for this decision is that the unavailable operations would be clumsy to implement in high-level syntax and that a more straightforward way to express the same is through pattern matching (see Section 4.2.3).

4.2.3 Pattern Expressions in Assignments

In the high-level language, the left-hand side of variable declaration statements is given far more power, allowing access to tuple fields and the extension of records through pattern expressions. Figure 4.5 gives some examples of how these pattern expressions translate to name-binding core language declarations. We can see that the statement `a.b.c = ...` translates to `x1 = a.b?`; `x1.c = ...` which is the only way a tolerant field access like `a.b?` (which evaluates to the empty record if `a` has no field named `b`) will find its way into a core program.

<pre> u, v, w = 1, 2, 3; a.name = "foo"; a.b.c.d = "bar"; name: foo, b: record = a; onetwo = one: 1, two: 2; </pre>	<pre> x0 = 1, 2, 3; u = x0[0]; v = x0[1]; w = x0[2]; a = {} ++ name: "foo"; x1 = a.b?; x2 = x1.c?; x3 = x2 ++ d: "bar"; x4 = x1 ++ c: x3; a = a ++ b: x4; foo = a.name; record = a.b; onetwo = {} ++ one: 1, two: 2; </pre>
--	---

Figure 4.5: Translation of tuple and record patterns.

4.2.4 Function Definitions

Function-symbol definition statements in the high-level language take an optional type annotation parameter, which is translated to explicit runtime type-check instructions (see Figure 4.6). The low-level representation of user-defined function symbols distinguishes the inner function symbol `__inner`, which is used for attaching function bodies, from the outer one (`__sym`) used in function calls. This way, it is the outer function symbol's responsibility to check pre- and postconditions upon execution of any user-defined function body.

As we will see in Section 4.2.5, this choice of representing user-defined function symbols as records allows user-defined types to be treated as value constructing functions, too.

In function body definitions, patterns are expressed in a structured form, as opposed to the core language notation of atomic constraints. Both are closely related, though, as is

shown in Figure 4.7. Moreover, any of those pattern expressions described in Section 4.2.3 can also appear in place of a name-binding expression in function patterns.

We have already seen in Figure 4.2 that the `return`-function becomes an explicit parameter in the course of the CPS-transform. Figure 4.8 explains the early syntactic translation and refers to the multi-channel case.

<pre> new fun: (Int, Int) --> String; def fun(x, y): { return("A"); } val = fun(7, 3); </pre>	<pre> fun = __type: Function, __inner: new; __sym: new; def fun.__sym(x0): { x1 = check (Int,Int) x0; x2 = fun.__inner(x1); x3 = check String x2; return x3; } def fun.__inner(x, y): { return("A"); } val = fun.__sym(7,3); </pre>
--	--

Figure 4.6: Translation of type-annotated function-symbol definitions.

<pre> def a(x) & b(x): ... def a(x) & b(y): ... def a(key: 3, val: x): ... def a(x, 3): ... def a(Int x): ... def a(Foo(val: 7)): ... def a(key: x.y): ... </pre>	<pre> def a(u) & b(v) where v = u:... def a(u) & b(v): ... def a(u) where u @ Record(key), u @ Record(val), u.key = 3: ... def a(u) where u @ Tuple(2), u[1] = 3: ... def a(u) where u @ Object(Int): ... def a(u) where u @ Object(Foo), u @ Record(val) u.val = 7: ... def a(u) where u @ Record(key): { x = {} ++ y:u.key; ... } </pre>
--	--

Figure 4.7: Translation of function patterns.

<pre>def a(x) & b(y): { def c(z): return(2 * z); returnto.a(x) & return(c(y)); }</pre>	<pre>def a(x)-->retA & b(y)-->retB: { returnto.a = retA; returnto.b = retB; return = returnto.b; def c(z)-->retC: { returnto.c = retC; return = returnto.c; return(2 * z); } returnto.a(x) & return(c(y)); }</pre>
---	--

Figure 4.8: Naming of continuation arguments.

4.2.5 Type Definitions

Definition of user-defined types is explained in Figure 4.9. Since types are, like functions, represented as records containing a channel-value `__sym` field, types can be used as value constructors in the same way as any other function is invoked. Due to the lack of an `__inner` field, no user-defined function bodies can be defined on type function symbols. The only accessible function body is created by default, taking a record as parameter and overriding that record's `__type` field after calling the super-type constructor.

4.2.6 Compilation Units

So far, the presentation has been concerned with only one single source program. Clearly, this is not a viable solution for any high-level language. Thus, program statements from several compilation units (i.e. source files) can be composed by means of `import`-clauses in the header of each file (see Figure 4.10). After resolving all dependencies, the compilation units are concatenated in topologically sorted order such that no file is included more than once. Imported files are specified relative to a root directory, which may be given by any of these three constants:

- `sys`: implementation-supplied libraries
- `proj`: working directory the compiler is invoked from
- `native`: native Java class (see Section 4.2.7)

If no import root is given, the import path is taken relative to the current file.

<pre> type Expr; type Const is Expr; type If is Expr; # evaluation code below new eval: Expr --> Value; def eval(Const ex): return(ex.val); def eval(If ex): { if eval(ex.cond): return(eval(ex.yes)); else return(eval(ex.no)); } print eval If(cond: Const(val: true), ifTrue: Const(val: 1), ifFalse: Const(val: 2)); # output: 1 </pre>	<pre> Expr = __type: Function, __sym: new, def Expr.__sym(rec): { o = rec; o.__type = Expr; return(o); } Const = __type: Function, __sym: new, __super: Expr; def Const.__sym(rec): { o = Expr(rec); o.__type = Const; return(o); } If = __type: Function, __sym: new, __super: Expr; def If.__sym(rec): { o = Expr(rec); o.__type = If; return(o); } # evaluation code below ... </pre>
--	---

Figure 4.9: Translation of type-definition statements.

4.2.7 Integration with Platform Code

As we will see in the code examples in Chapter 5, we can write code in our language that looks surprisingly similar to regular Java code. This is made possible by the fact that we can import any public Java class as if it were a regular compilation unit (see Figure 4.11).

For each group of imported Java methods with the same name, a function symbol is created and wrapper function bodies are attached for all imported non-static method implementations. The function bodies, set up with appropriate patterns, call through to the native implementation. Constructor functions are available under the name of the class itself.

In addition, there is a record for each class which enables access to the Java class's static and final fields as well as the static methods. For Java interfaces, a constructor function is created that takes a record with appropriately named function-value fields and returns an implementation of the corresponding interface, wrapping the provided functions.



Figure 4.10: Compilation of file `./main`. Imported files are prepended in topologically sorted order.

4.2.8 Syntactic Sugar

Figure 4.12 displays a summary of the syntax-level expressions in our language, such as the operator symbols and their precedence rules. Internally, operators are implemented as native functions, and their function symbols can be accessed directly (e.g. `'x'`). Conditionals, i.e. `if/else` expressions, are translated to functions that pattern-match on `true` and `false` values (see Figure 4.13). Anonymous functions (lambda-expressions) and generator-comprehensions are translated according to Figures 4.14 and 4.15. The translation of generator-literal expressions is shown in Figure 4.16.

<pre> # Class java.foo.FooBar package java.foo; public class FooBar { public static final FOO=... public FooBar(int x) { ... } public String toString() { ... } public static void bar() { ... } } </pre>	<pre> # Class java.lang.Object Object = new; ... toString = new; def toString(java.lang.Object o): { return(java.foo.FooBar. toString(o)); } ... # Class java.foo.FooBar FooBar = new; FooBarClass.FOO = java.foo. FooBar.FOO; def FooBar(java.lang.Integer x): { return(java.foo.FooBar. <init>(unbox(x))); } def toString(java.foo.FooBar o): { return(java.foo.FooBar. toString(o)); } def FooBarClass.bar(): { return(java.foo.FooBar. bar()); } # File ./main ... </pre>
<pre> # File ./main import native java.foo.FooBar; ... </pre>	

Figure 4.11: Access to native Java classes.

Operator Symbol	Description
<code>if cond then ... else ...</code>	conditional
<code>... for x in gen</code>	for-comprehension
<code>x, y, z</code>	tuple formation
<code>a:x, b:y, c:z</code>	record formation
<code>fn x: ...</code>	anonymous function
<code> </code>	logical <i>or</i>
<code>&&</code>	logical <i>and</i>
<code>==, !=</code>	equality, inequality
<code><, <=, >, >=</code>	comparison
<code>..</code>	range
<code>+, -</code>	addition, subtraction
<code>*, /, %</code>	multiplication, division, remainder
<code>+x, -x, !x</code>	unary identity, negation, inverse
<code>**</code>	exponentiation
<code>-></code>	partial application
<code>.</code>	record field access
<code>f x</code>	function application (right-associative)
<code>(...), [...]</code>	grouping, generator expression

Figure 4.12: Operators and precedence rules (stronger binding at the bottom).

<pre> if c: { ... } else: { ... } r = if c then a else b; </pre>	<pre> new x0; def x0(true): { ... return(); } def x0(false): { ... return(); } x0(c); new x1; def x1(true): return a; def x1(false): return b; r = x1(c); </pre>
---	--

Figure 4.13: Translation of if/else statements and expressions.

```
f = fn x: y;
```

```
new x0;
def x0(x): return(y);

f = x0;
```

Figure 4.14: Translation of lambda expressions.

```
f = x for i in g
```

```
for x in f:
  print x;
```

```
new x0;
def x0(yield):
{
  new x1;
  def x1(i):
  {
    return yield(x);
  }
  return g(x1);
}

f = x0;

new x1;
def x1(x):
{
  print x;
  return();
}

f(x);
```

Figure 4.15: Translation of for-comprehension statements and expressions.

```
f = [a, b, c];
```

```
f = [a & b & c];
```

```
new x0;
def x0(yield):
{
  yield(a);
  yield(b);
  yield(c);

  return();
}

f = x0;

new x1;
def x1(yield):
{
  yield(a) &
  yield(b) &
  yield(c);

  return();
}

f = x1;
```

Figure 4.16: Translation of generator expressions.

4.3 Reference Implementation

The reference implementation that has been created as part of this work consists of an interpreter for the core language described and a compiler translating the high-level syntax to executable core language code according to the rules defined in Section 4.2. Some details are implemented in a slightly more machine-oriented way though, such as removing the need to lookup identifiers by name from the interpreter by replacing named identifiers with numeric indices into lexically nested environment records. Some other simple optimizing transforms are also carried out during the compiler passes.

The system provides both an interactive shell, inside which short code snippets may be entered and executed directly, as well as a batch mode that reads programs from disk files. There is a third operation mode, which is aimed at testing the toolchain itself as well as individual programs. In this test mode, a list of programs can be run one after another, suppressing any visible output but doing a comparison with expected reference output that may be specified as a tagged comment in each of the source files.

The system itself is implemented in the language Nice [Bonniot, 2003a], which compiles directly to Java byte-code. Nice is a Java-like language with multimethods, which has proven a valuable tool in the course of the development. The source code is organized into six packages, as outlined in Figure 4.17.

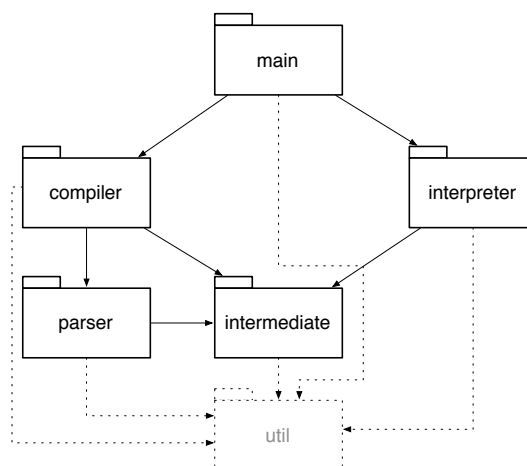


Figure 4.17: Implementation source packages and dependencies.

Some care has been taken to structure the package dependencies in a way that is amenable to a future extension of the system. As an example, all code transforms are contained in the package `intermediate`, that has no external dependencies other than the

very small `util` package. Likewise, the `interpreter` depends only on the data structures representing program code provided by `intermediate`.

The `interpreter` package contains multiple scheduler implementations, which are largely independent from the code that interprets program statements. The scheduler framework transparently executes native code, too, which is indispensable for integrating with platform libraries. When Java classes are imported, wrapper byte-code is generated dynamically. Using Java reflection has proven too slow for this task.

The `compiler` package provides infrastructure code to locate declared imports and to initiate the chain of parsing, code transformation and error handling.

The `main` package controls the command line interface and provides the overall user interaction facilities, such as reporting errors and promoting configuration settings. Upwards communication from the lower-level modules is handled in such a way that `main` package code implements context interfaces published by the lower-level modules, thereby obviating the need for explicit upward dependencies.

4.3.1 Availability

The language tools and their source code have been made available under a permissive MIT-style license. Up-to-date source code can be downloaded anonymously from the project's subversion repository [Rompf, 2007] hosted at Google Code.

4.3.2 Performance Evaluation

Since the purpose of the provided implementation is to serve as a reference and as a “proof of concept”, the development focus was not on performance, but on soundness and code maintainability. Therefore, only little effort has gone into optimization, and absolute performance measurements are not very conclusive.

In the requirements set forth in Section 1.4, however, efficient execution on multi-processor hardware was demanded. Therefore, the implementation provides support for concurrent execution by means of the work-stealing and transaction based model described in Section 3.4.2. For scheduling purposes, the implementation makes use of the FJTask-toolkit [Lea, 2000].

As a benchmark for performance evaluation, a program that calculates Fibonacci numbers in a concurrent divide-and-conquer fashion was chosen (highly concurrent, with frequent and fine-grained synchronization). Performance has been measured on a 64-

processor Sun Fire 15k machine, running Solaris 9i and JDK 1.5. Numbers are shown in Figure 4.18, graphs in Figure 4.19. The code used is shown in Figure 4.20.

Processors	Time/s	Speedup
1	153.405	1
2	78.664	1,950
4	40.317	3.804
8	21.534	7.123
16	13.229	11.596

Figure 4.18: Running times of calculating $fib(25)$, using an increasing number of processors.

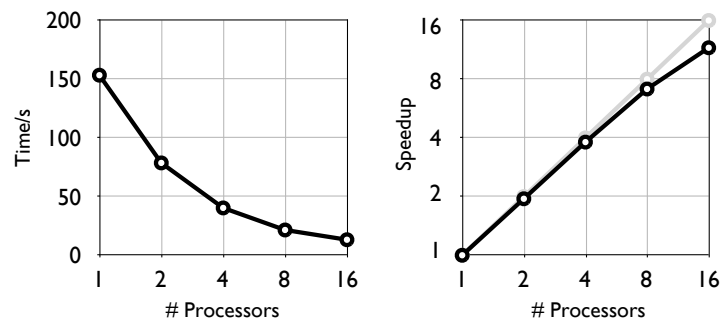


Figure 4.19: Running times of calculating $fib(25)$, using an increasing number of processors.

The speedup numbers for up to 16 processors are very satisfactory, but it has to be noted that for the Fibonacci benchmark, there is not much benefit in adding even more processors. Profiling has shown that with a higher number of processors, there is a rapid increase in the relative amount of time that is spent scanning for activations to be stolen.

This fact is not very astonishing, since running the Java-implemented Fibonacci benchmark program that comes with FJTask shows similar behavior and does not yield better speedups, unless one resorts to computing the lowest n Fibonacci numbers sequentially for a suitable n (refer to the discussion in Lea [2000] for more information).

Furthermore, not much attention has been paid to proper tuning of Java VM parameters, apart from increasing the heap size to several gigabytes of memory in order to prevent garbage collection artifacts (the young generation space was chosen especially large, since short-lived objects are allocated rapidly). Another fact is that the Java VM, operating in “server” mode on the target machine, optimizes for high throughput in long-running applications. It is not clear, whether any of the test runs took long enough to really profit

```
new fib;

def fib(0): return 0;
def fib(1): return 1;

def fib(n):
{
  new a;
  new b;

  def a(n1) & b(n2): returnto.fib(n1 + n2);

  a(fib(n-2)) & b(fib(n-1));
}
```

Figure 4.20: Fibonacci benchmark implementation.

from extensive, behind-the-scenes VM optimization. But the shorter the absolute running time of a program is, i.e. the higher the number of processors, the more time is relatively spent on “hot spot” bookkeeping.

Therefore, it should be safe to assume that the 16-processor limit is rather arbitrary and that near-linear speedups can be achieved for higher numbers of processors, too, depending on the characteristics of the benchmark program.

Chapter 5

Code Examples

In this chapter, we take up upon the use cases described in Section 1.3 and provide some non-trivial programming examples. Focusing on a declarative style of specifying interactive behavior and on integration with native Java code, we describe a method of data-flow based GUI programming in Section 5.1. In Section 5.2, we show how we can greatly simplify the animation code of a walking robot and distill a rather complex animation sequence out of simple keyframe-oriented scripts. Section 5.3 discusses the implementation of a web-application server, providing a synchronous I/O-layer on top of scalable, asynchronous platform I/O. In addition, we demonstrate the transparent use of continuations in cross-request control-flow by writing a little guess-a-number game.

5.1 Declarative Dataflow GUI Programming

In this section, we demonstrate how our language can be used in the creation of graphical user-interfaces (GUIs). After showing how to use the standard Java Swing framework [Hoy et al., 2002] directly, we present a declarative, dataflow-oriented abstraction layer on top of this.

5.1.1 Accessing Legacy Java Classes

Any Java class that is available on the runtime's classpath can be imported with an `import native` declaration. Moreover and thanks to the partial-application operator `->`, we can write code that, on the first glance, looks almost like regular Java code:

```
#
# Create a swing frame with a button inside that will print a message
# when clicked
#

import library.util;

import native javax.swing.JButton;
import native javax.swing.JLabel;
import native javax.swing.JFrame;
import native java.awt.FlowLayout;

import library.swingUtil;

button = JButton("Button");

button->addActionListener(ActionListener(actionPerformed:
                                fn e: Console.print("Click!")));

frame = JFrame("MyFrame");

frame->getContentPane()->setLayout(FlowLayout());
frame->getContentPane()->add(JLabel("Click here:"));
frame->getContentPane()->add(button);

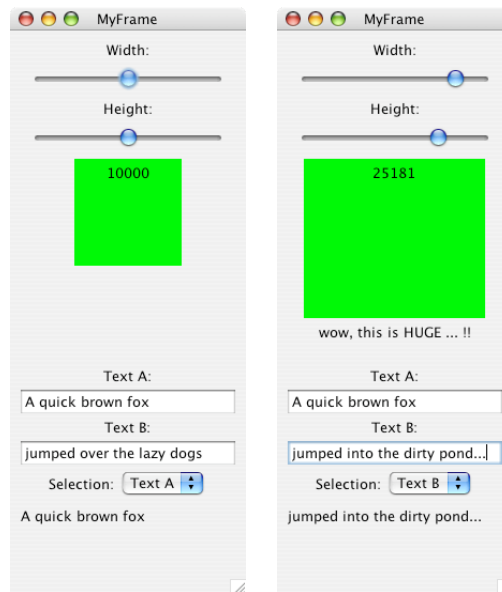
frame->pack();
frame->show();

stop();
```

In addition to calling Java methods ourselves, we can create objects that implement particular Java interfaces, as is the case with the above action listener. In fact, we can turn any record with appropriately named function-value fields into an implementation of a Java interface by passing the record to the interface constructor (in this case `ActionListener`).

5.1.2 Introducing Data Dependencies

Although feeling at home with the API may be a nice thing, it would not be much of an improvement, if writing Swing code "the old way" was the sole means of GUI construction in our language. To illustrate our more declarative approach to GUI development, which concentrates on the data dependencies rather than on state-changes, we are going to construct the following window:



Dragging one of sliders changes the width or height of the green rectangle, whose area is displayed as a numeric value inside. If the green rectangle grows larger than 20,000 square pixels, a text notice is shown. In the lower half of the window, the selection from the popup menu determines from which of the text fields the content of the label at the bottom is drawn. Editing that text field also changes the text below interactively.

In implementing this functionality, we make use of reference cells, another kind of advanced mutable variables we can implement in the language itself. Reference cells allow us to define data dependencies between variables and have state changes be tracked automatically.

In order to achieve this, the `set` operation on reference cells no longer takes a value that defines the contents of the variable, but a function that is executed to (re-)compute the actual value. Within this function, other variables may be referenced by calling their `subscribe` operation.

```

#
# Upper region of the window
#

sliderX = VSlider(0, 200);
sliderY = VSlider(0, 200);

areaValue = VCell
(
  fn t: sliderX.value.subscribe(t) * sliderY.value.subscribe(t)
);

areaLabel = VLabel("-");
areaLabel.value.set
(
  fn t: toString(areaValue.value.subscribe(t))
);

areaPanel = VFlowPanel[areaLabel];
areaPanel.comp->setBackground(Color(0, 255, 0));

areaPanel.size.set
(
  fn t: (sliderX.value.subscribe(t), sliderY.value.subscribe(t))
);

notice = VLabel("wow, this is HUGE ... !!");
notice.visibility.set
(
  fn t: areaValue.value.subscribe(t) > 20000
);

```

In constructing the lower half of the window, we make use of the fact that reference cells, too, can be treated as first class values. The value of the `selection` cell is one of the text fields, and the value of the `selectionText` label is defined as the value of that text field, which is the value of the `selection` cell.

```

#
# Lower half components
#

textA = VTextField("A quick brown fox");
textB = VTextField("jumped over the lazy dogs");

textA.size.set(fn t: (200, 22));
textB.size.set(fn t: (200, 22));

menu = VComboBox(["Text A", "Text B"]->toArray());

selectionText = VLabel("-");
selectionText.size.set(fn t: (200, 22));

selection = VCell
(
  fn t: (if menu.value.subscribe(t)->equals("Text A") then textA else textB)
);

selectionText.value.set
(
  fn t: (selection.value.subscribe(t)).value.subscribe(t)
);

```

To assemble components into panels, we make use of generator expressions:

```
#
# JFrame assembly
#

sliders = VFlowPanel
[
    VLabel("Width:"), sliderX,
    VLabel("Height:"), sliderY,
    areaPanel,
    notice
];

sliders.size.set(fn t: (220, 300));

texts = VFlowPanel
[
    VLabel("Text A:"), textA,
    VLabel("Text B:"), textB,
    VLabel("Selection:"), menu,
    selectionText
];

texts.size.set(fn t: (220, 250));

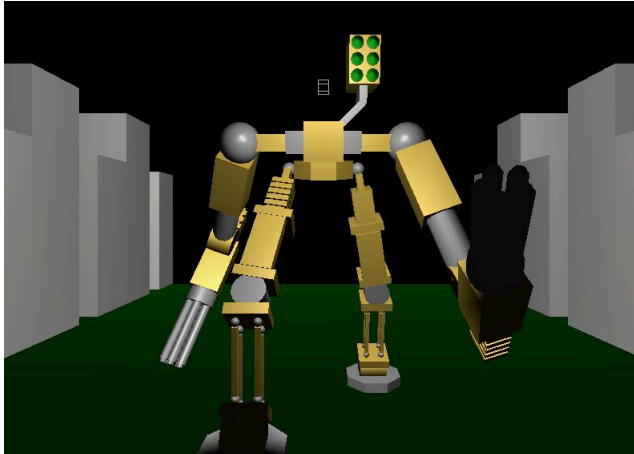
base = VFlowPanel [sliders, texts];

frame = JFrame("MyFrame");
frame->setContentPane(base.comp);
frame->setSize(220, 550);
frame->show();
```

Putting these code snippets together yields exactly the required functionality, without writing a single callback or update function ourselves. Moreover, the reference cell implementation is sophisticated enough to recompute just those cells that have actually changed, and to promote those changes in a topologically sorted way.

5.2 Walking Robot (“glutmech”) Animation

In this example we show how an animated character’s tight C code can be adapted in our new language to provide for modular, composable animation scripting.



The original glutmech [Parkinson-Bates, 2001] is one of the standard examples that come with most OpenGL installations. There are several keyboard commands that allow the user to move individual limbs of the robot, as well as a fairly smooth animation mode that makes the robot appear to be walking towards the user.

What we are interested in is how this animation is scripted and how it might be extended to make the robot do other interesting things. Looking at the original C code in Figure 5.1 reveals that most of the work is done by the function `animation_walk`, which is registered as an on-idle callback. Each time `animation_walk` is called, it updates the robot limbs’ positions and calls `glutPostRedisplay`, which triggers another callback to update the on-screen drawing.

5.2.1 Keyframe Animation

Taking a closer look at the code reveals that the animation consists of four keyframes that define the rotation parameters of the robot’s joints at four discrete moments in time. Between these keyframes, values are interpolated linearly.



By looking even closer, we can see that the animation sequence is in fact three animations taking place in parallel: moving the legs, the arms, and the torso.

```

if (frame > 0)
    elevation = -(3.2055 - (cos((M_PI / 180)
        * angle) * 3.2055));
else
    elevation = 0.0;
if (step == 1) {
    hip21 = hip21 - frame;
    elbow1 = heel1 = frame;
    heel2 = 15;
    ankle1 = frame;
    if (frame > 0)
        hip21 = angle;
    else
        hip21 = 0;
    ankle2 = -hip21;
    shoulder1 = 1.5 * frame;
    shoulder2 = -frame * 1.5;
} else {
    elbow1 = hip21 = -frame;
    elbow2 = heel2 = frame;
    heel1 = 15;
    ankle2 = frame;
    if (frame > 0)
        hip11 = angle;
    else
        hip11 = 0;
    ankle1 = -hip11;
    shoulder1 = -frame * 1.5;
    shoulder2 = frame * 1.5;
}
if (frame == 0.0)
    step++;
if (frame > 0)
    frame = frame - 3.0;
}
if (step == 4)
    step = 0;
distance += 0.1678;
glutPostRedisplay();
}

void
display(void)
{ // OpenGL drawing code
}

int
main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(600, 600);
    glutCreateWindow("glutmech: Vulcan Gunner");
    glutIdleFunc(animation_walk);
    glutDisplayFunc(display);
    glutMainLoop();
}

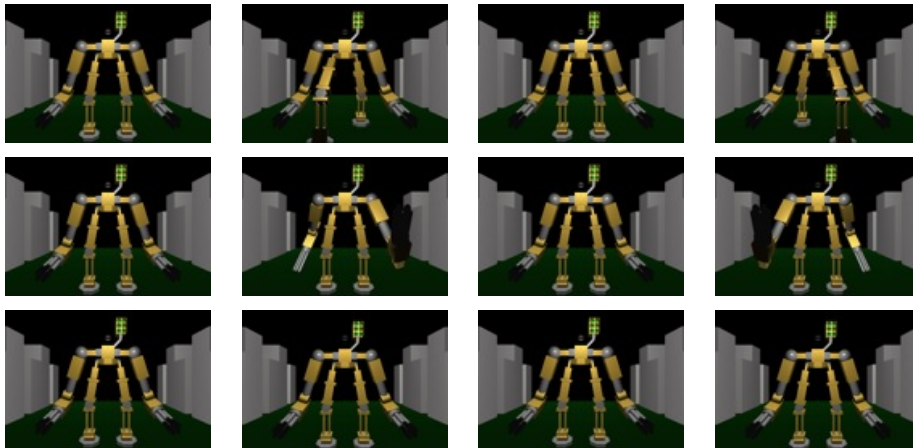
```

```

void
animation_walk(void)
{
    float angle;
    static int step;
    if (step == 0 || step == 2) {
        *cos(frame=3.0; frame<=21.0; frame=frame+3.0){ * /
        if (frame >= 0.0 && frame <= 21.0) {
            if (frame == 0.0)
                frame = 3.0;
            angle = (180 / M_PI) * (acos((cos((M_PI / 180) * frame)
                * 2.043) + 1.1625) / 3.2059));
            if (frame > 0) {
                elevation = -(3.2055 - (cos((M_PI / 180)
                    * angle) * 3.2055));
            } else
                elevation = 0.0;
            if (step == 0) {
                hip11 = -(frame * 1.7);
                if (1.7 * frame > 15);
                heel1 = frame * 1.7;
                heel2 = 0;
                ankle1 = frame * 1.7;
                if (frame > 0)
                    hip21 = angle;
                else
                    hip21 = 0;
                ankle2 = -hip21;
                shoulder1 = frame * 1.5;
                shoulder2 = frame * 1.5;
                elbow1 = frame;
                elbow2 = -frame;
            } else {
                hip21 = -(frame * 1.7);
                if (1.7 * frame > 15);
                heel1 = 0;
                heel2 = frame * 1.7;
                ankle2 = frame * 1.7;
                if (frame > 0)
                    hip11 = angle;
                else
                    hip11 = 0;
                ankle1 = -hip11;
                shoulder1 = -frame * 1.5;
                shoulder2 = frame * 1.5;
                elbow1 = -frame;
                elbow2 = frame;
            }
            if (frame == 21)
                step++;
            if (frame < 21)
                frame = frame + 3.0;
        }
    }
    if (step == 1 || step == 3) {
        * / * cos(frame=3.0; frame<=21.0; frame=frame+3.0){ * /
        if (frame <= 21.0 && frame >= 0.0) {
            angle = (180 / M_PI) * (acos((cos((M_PI / 180) * frame)
                * 2.043) + 1.1625) / 3.2029));
        }
    }
}

```

Figure 5.1: Original glutmech animation code.



5.2.2 Identifying the Building Blocks

With this knowledge, we can refactor the code quite a bit. Porting the whole code to our new language, however, is not necessary, since we can just as well leave the low-level drawing to native Java methods. Porting the drawing code from C to Java is fairly straightforward.

Isolating the three individual animation sequences and dropping the trailing redisplay trigger (the `glutPostRedisplay()` statement), we arrive at four methods (one for each pair of adjacent keyframes, in wrap-around style) per sequence. The parameter `k`, with a domain of `1..7`, defines the interpolated frame.

```
public class TestMech
{
    public void animation_legs0(int k)
    {
        float frame = 3.0f * k;
        float angle = (180 / M_PI) * ((float)Math.acos((((float)Math.cos(
            (M_PI / 180) * frame) * 2.043f) + 1.1625f) / 3.2059f));

        hip11 = -(frame * 1.7f);
        if (1.7f * frame > 15)
            heel1 = frame * 1.7f;
        heel2 = 0;
        ankle1 = frame * 1.7f;
        if (frame > 0)
            hip21 = angle;
        else
            hip21 = 0;
        ankle2 = -hip21;
    }

    public void animation_legs1(int k) {...}
    public void animation_legs2(int k) {...}
    public void animation_legs3(int k) {...}

    public void animation_arms0(int k) {...}
    public void animation_arms1(int k) {...}
    public void animation_arms2(int k) {...}
    public void animation_arms3(int k) {...}

    public void animation_body0(int k) {...}
}
```

```

    public void animation_body1(int k) {...}
    public void animation_body2(int k) {...}
    public void animation_body3(int k) {...}

    public void display()
    {
        // handle OpenGL drawing
    }
}

```

5.2.3 Re-assembling the Parts

Chaining the keyframe interpolation sequences together is done entirely in our new language. We parameterize over the mech object and, more importantly, over a function `nextFrame` that is called each time computation of a new animation frame is about to start.

```

new animateLegs;
new animateArms;
new animateBody;

def animateLegs(mech, nextFrame):
{
    # animate to keyframe 1

    for i in 1..7:
    {
        mech->animation_legs0(i);
        nextFrame();
    }

    # animate to keyframe 2

    for i in 1..7:
    {
        mech->animation_legs1(i);
        nextFrame();
    }

    # animate to keyframe 3

    for i in 1..7:
    {
        mech->animation_legs2(i);
        nextFrame();
    }

    # animate back to keyframe 0

    for i in 1..7:
    {
        mech->animation_legs3(i);
        nextFrame();
    }

    return();
}

def animateArms(mech, nextFrame): {...}
def animateBody(mech, nextFrame): {...}

```

Using the postfix for-loop notation, we can express the above code a little more terse:

```

new animateLegs;
new animateArms;
new animateBody;

def animateLegs(mech, nextFrame):
{
  mech->animation_legs0(i)->nextFrame() for i in 1..7;
  mech->animation_legs1(i)->nextFrame() for i in 1..7;
  mech->animation_legs2(i)->nextFrame() for i in 1..7;
  mech->animation_legs3(i)->nextFrame() for i in 1..7;

  return();
}

def animateArms(mech, nextFrame): {...}
def animateBody(mech, nextFrame): {...}

```

Now we can run any one of the three animations, but only one at a time.

In the original callback-oriented model, `nextFrame` would trigger a repaint (via `glutPostRedisplay`) and wait until the drawing has completed. We can achieve the same effect more straightforwardly, by just using the `mech`'s `display` method as `nextFrame`.

```

mech = TestMech();
(mech, mech->display)->animateLegs->loop();

```

5.2.4 Parallel Composition

The interesting part is now to combine the three basic animations to the original, composite one. With the help of the higher order function `animateParallel`, we can accomplish just that. Note how the given `nextFrame` function parameter is only called after each of the three animation sequences have called their `nextFrame` parameter:

```

new animateParallel;

def animateParallel(mech, a, b, c, nextFrame):
{
  new nfa;
  new nfb;
  new nfc;

  def nfa() & nfb() & nfc():
  {
    nextFrame();

    returnto.nfa() & returnto.nfb() & returnto.nfc();
  }

  mech->a(nfa) & mech->b(nfb) & mech->c(nfc);

  return();
}

```

With this utility function, which re-introduces a top-down control structure, we can run the combined animation sequence.

```

mech = TestMech();
(mech, animateBody, animateArms, animateLegs, mech->display)
  ->animateParallel->loop();

```

And we can build more complicated animations. Below we define another animation routine, `animateFire`, and integrate it with the other ones into a procedural animation script. The robot now does four regular steps, then waves its arms, does two little hops forward, fires its cannons, and starts over.

```

new animateFire;
def animateFire(mech, nextFrame):
{
  for i in 1..10:
  {
    mech->elbow1Subtract();
    mech->elbow2Subtract();
    mech->shoulder1Subtract() if i % 3 == 0;
    mech->shoulder2Subtract() if i % 3 == 0;
    nextFrame();
  }

  for i in 1..50:
  {
    mech->FireCannon();
    nextFrame();
  }

  for i in 1..10:
  {
    mech->elbow1Add();
    mech->elbow2Add();
    mech->shoulder1Add() if i % 3 == 0;
    mech->shoulder2Add() if i % 3 == 0;
    nextFrame();
  }

  return();
}

new script;
def script(nextFrame):
{
  mech->animateParallel(animateBody, animateLegs, animateArms, nextFrame);
  mech->animateParallel(animateBody, animateLegs, animateArms, nextFrame);

  mech->animateArms(nextFrame);
  mech->animateBody(nextFrame);

  mech->animateFire(nextFrame);

  script(nextFrame);
}

script(mech->display);

```

5.2.5 The Power of Generators

Taking a closer look at the nature of the `nextFrame` parameter each animation function takes, we can make another observation: The function `nextFrame` is invoked after one

animation frame is produced and returns control when computation of the next frame can commence, i.e. the frame has been fully rendered to the screen.

In its role, `nextFrame` is thus very similar to the `yield` parameter of generator functions: it yields control until a new value is to be produced. In fact, partially applied animation functions like `mech->animateLegs` can be used just like any other generator.

This fact obviates the need for a custom-made `animateParallel` function as we can use the library function `zip` instead. The original animation can now be expressed as:

```
mech = TestMech();
for frame in repeat zip(mech->animateLegs, mech->animateArms, mech->animateBody):
    mech->display();
```

In addition, we can use generator expressions to build animation scripts like the one presented above and iterate over the individual steps and animation frames by generator comprehension:

```
script =
[
    (mech->animateLegs, mech->animateArms, mech->animateBody)->zip(),
    (mech->animateLegs, mech->animateArms, mech->animateBody)->zip(),
    mech->animateArms,
    mech->animateBody,
    mech->animateFire
];

for step in script->repeat():
    for frame in step:
        mech->display();
```

In this simulation-oriented example, we have seen how our language can serve as a tool to coordinate native-code building-blocks on a higher level of abstraction. In addition, we have seen how generators can serve as a means to control synchronization and interactivity.

5.3 Web-Application Server

In this example, we describe the implementation of a simple web-application server. There are two things we are concerned with: First, we would like to use scalable, multiplexed, asynchronous I/O (through the Java NIO API [Hitchens, 2002]) with a syntax that resembles that of blocking (`readLine`-style) I/O. And second, we want to handle advanced application control flow across individual HTTP requests.

5.3.1 Main Server Loop

Basically, what an HTTP server does is the following: For each incoming connection on port 80, it reads requests and replies with responses until the connection is closed (see [Fielding et al., 1999]). We are going to use the following program code:

```
#
# Main program
#

for socketChannel in acceptConnections(80):
{
    Trace.print("Connect: $socketChannel");

    for req in readRequests(socketChannel):
    {
        res = handleRequest(req);

        writeResponse(socketChannel, res);
    }

    Trace.print("Disconnect: $socketChannel");
}
```

There are a number of things to be concerned with, though. Most importantly, no real HTTP server would implement this control loop in a sequential fashion, as there could be only one client connected at a time (or all connections would have to be closed after answering the first request). A number of real-world servers solve this problem by starting a new thread for each connection, but this leads to scalability problems if there are a lot of concurrently connected users.

The solution of choice is to limit the number of threads and to handle I/O asynchronously, multiplexing state-change events from several connections. The Java NIO package provides an efficient implementation of such a callback-based I/O library, but unfortunately, the API is far less easy to use than the customary, blocking, stream-interface.

Our goal is to use the above code as the server main loop and have it run fully asynchronous, multiplexed I/O.

5.3.2 Selectors and Callbacks

In the Java NIO API, there is a central `Selector` class, whose instances take care of tracking updates on a number of `SelectionKey` objects. We define a generator that repeatedly executes `select` on a given selector and yields the set of selected keys:

```
new selections;
def selections(selector):
{
  new gen;
  def gen(yield):
  {
    count = selector->selectNow();

    if count > 0:
      yield(selector->selectedKeys());

    gen(yield);
  }
  return gen;
}
```

In a second step, we define a function that creates a new selector object and, in parallel to returning the selector to the caller, starts a loop that iterates over the above selection-generator. For each set of selected keys, each key's attached callback handler is invoked and after that, the selection is cleared.

```
new createAsyncSelector;
def createAsyncSelector():
{
  selector = SelectorProviderClass.provider()->openSelector();

  {
    for keySet in selections(selector):
    {
      for k in keySet->iter():
      {
        Trace.print("Select: $k");

        handler = k->attachment();
        handler(k);
      }

      keySet->clear();
    }
  }
  &
  return selector;
}
```

5.3.3 Parallelizing Callbacks

So far, we have automated the execution of callback handlers, but we have not introduced more fine-grained concurrency.

What we are going to do now is to wrap up each callback handler in a generator that produces a value each time that specific callback is invoked. Implementing this in a generalized manner allows us to introduce the concurrency we are striving for: In parallel to the `yield` call, the callback returns, allowing other callbacks to be handled concurrently and new `select` invocations to occur. In order to prevent the same event from being handled multiple times, the corresponding selection key is deregistered from the selector until the callback is fully handled. In effect, invocations of each callback handler are serialized and executed in its own *conceptual* thread (which, of course, has nothing to do with how the runtime might utilize platform threads).

```

new callbacks;
def callbacks(channel, selector, ops):
{
  new gen;
  def gen(yield):
  {
    new callback;
    def callback(key):
    {
      key->interestOps(0);

      {
        yield(key);

        if key->isValid():
        {
          key->interestOps(ops);
          selector->wakeup();
        }
        else:
          returnto.gen();
      }
      &
      return();
    }

    selectionKey = channel->register(selector, ops, callback);
  }

  return gen;
}

```

5.3.4 Accepting Connections

With this callback generator at hand, we can implement the `acceptConnections` generator. Note the `fork yield` statement, which, by allowing the callback-generator to return, re-enables the flag that tells the server socket to listen for additional connection attempts.

```

new acceptConnections;
def acceptConnections(port):
{
  new gen;
  def gen(yield):
  {
    serverSocketChannel = ServerSocketChannelClass.open();

    serverSocketChannel->configureBlocking(false);

    isa = InetAddress(port);

    serverSocketChannel->socket()->bind(isa);

    for key in serverSocketChannel->callbacks(selector,
      SelectionKeyClass.OP_ACCEPT):
    {
      serverSocketChannel = key->channel();

      socketChannel = serverSocketChannel->accept();
      socketChannel->configureBlocking(false);

      (fork yield) (socketChannel);
    }
    return();
  }
  return gen;
}

```

5.3.5 Retrieving Data

In the main-loop description above, there is a function `readRequest` and a function `writeResponse`, as well as a function `handleRequest`. Since we do not really care about the internals of the HTTP protocol, we skip a detailed explanation of these methods. Instead, we take a closer look at a buffered reader implementation which provides a function `readLine`, which is central to the parsing of HTTP requests and a good example of avoiding the inversion of control usually incurred by asynchronous I/O.

Just like the generator that yielded incoming connections, we can implement a generator that yields on newly received data:

```

new readBytes;
def readBytes(socketChannel, bufSize):
{
  new gen;
  def gen(yield):

```

```

    {
        buffer = ByteBufferClass.allocateDirect(bufSize);
        for key in socketChannel->callbacks(selector,
                                           SelectionKeyClass.OP_READ):
            {
                count = socketChannel->read(buffer);
                if count < 0:
                    {
                        socketChannel->close();
                        return to.gen();
                    }
                buffer->flip();
                yield(buffer);
                buffer->clear();
            }
        return();
    }
    return gen;
}

```

With this generator, which is set up to repeatedly call the function `inputReceived`, we can handle buffered reading, using the following rendezvous scheme:

```

new BufferedDecoder;
def BufferedDecoder(socketChannel):
    return BufferedDecoder(socketChannel, 4096);
def BufferedDecoder(socketChannel, bufSize):
    {
        new bufferState;
        new inputReceived;

        def bufferState(charBuffer, waitingForInput) & inputReceived(byteBuffer):
            {
                # update charBuffer with data from byteBuffer, possibly
                # compacting/resizing
                # ...

                bufferState(charBuffer, false) & return();
            }

        new this.readLine;
        def bufferState(charBuffer, false) & this.readLine():
            {
                lineMatcher = PatternClass.compile("(.*\r?\n")->matcher(charBuffer);

                if lineMatcher->find():
                    {
                        line = lineMatcher->group(1);

                        bufferState(charBuffer, false) & return(line);
                    }
                else:
                    bufferState(charBuffer, true) & return(this.readLine());
            }

        bufferState(CharBufferClass.allocate(bufSize), true) &
        readBytes(socketChannel, bufSize)->foreach(inputReceived) &
        return this;
    }
}

```

Calls to `readLine` now behave in the well-known, synchronously blocking way one would expect, despite the underlying asynchronous implementation. A part of the request parsing function is shown below:

```
new readRequests;

def readRequests(socketChannel):
{
  reader = BufferedDecoder(socketChannel);

  new gen;
  def gen(yield):
  {
    # example first line: GET /foo/bar HTTP/1.1

    requestLine = reader.readLine()->split(" ")->asList();

    req.httpMethod   = requestLine->get(0);
    req.httpLocation = requestLine->get(1);
    req.httpVersion  = requestLine->get(2);

    # read headers and message body
    # ...

    yield(req);

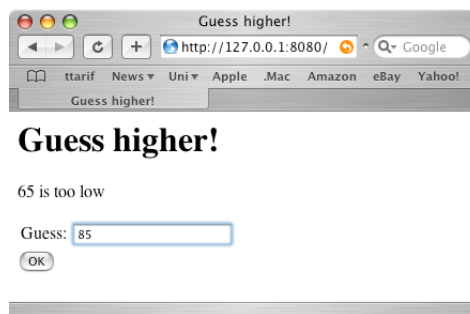
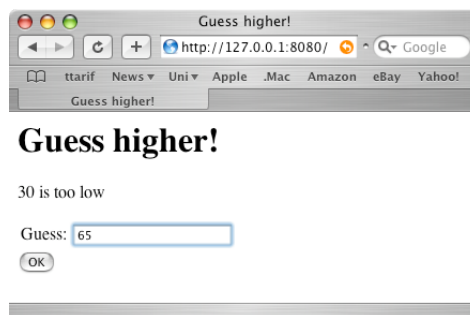
    gen(yield);
  }
  return gen;
}
```

5.3.6 Application-Level Control Flow

The stateless request/response nature of the HTTP protocol is well-suited for the exchange of independent units of information, but coordinating a multi-request user interaction is a challenge. Not to mention that the user may at any time open a new browser window, or hit the “back” or “reload”-button.

Adopting the (rather non-standard) approach of some web frameworks such as Seaside [Ducasse et al., 2004], we can provide server-side cross-request scripting with a blocking send-page-and-wait metaphor. Each time a page is sent to the client, the server application logic is interrupted, saving a continuation that may be resumed upon receiving a future request.

We illustrate this approach by implementing a little *guess-a-number* game, asking the user to guess a previously chosen number and giving hints on whether to guess lower or higher. Once the user guesses correctly, the application reports the number of guesses needed and starts over with a fresh random number.



... and some guesses later ...



The code we will use is very similar to an implementation targeting a text-based console with `println` and `readLine` calls:

```

numberGen = Random();
new playGame;
def playGame(s):
{
  num = numberGen->nextInt(100);

  s = s->sendHtmlFormPage("Guess my number!",
    "It's between 0 and 100", ["Guess"]);

  new getAnswer;
  def getAnswer(s, n):
  {
    x = IntegerClass.parseInt(s.req.params->get("Guess"));

    if (x < num):
    {
      s = s->sendHtmlFormPage("Guess higher!",
        "$x is too low", ["Guess"]);

      return getAnswer(s, n+1);
    }

    if (x > num):
    {
      s = s->sendHtmlFormPage("Guess lower!",
        "$x is too high", ["Guess"]);

      return getAnswer(s, n+1);
    }

    if (x == num):
    {
      s = s->sendHtmlFormPage("Correct!",
        "$x is the answer (it took you $n tries to find out)", []);

      return(s);
    }
  }

  s = getAnswer(s, 1);
  return playGame(s);
}

```

As we can easily see, the function `sendHtmlFormPage` plays a central part in the code above.

5.3.7 Infrastructure

To understand the underlying mechanism, we first take a look at how the function `handleRequest` (called from the main server loop) is implemented.

```

handlerMap = HashMap();
handlerMap->put("/", playGame);
new handleRequest;

```

```

def handleRequest(req):
{
    handlerID = if req.params->containsKey("conversationID")
                then req.params->get("conversationID")
                else req.httpLocation;

    handler = if handlerMap->containsKey(handlerID)
              then handlerMap->get(handlerID)
              else errorHandler;

    session = req: req, resume: return;

    handler(session);
}

```

The function `handleRequest` does never return by itself. Instead, the current continuation (`return`) is saved in a session record, together with the incoming request. Depending on the requested URL (`req.httpLocation`) or on a *conversation id*, if supplied, a request handler is chosen which is then called with the session record as parameter. So in order to have `handleRequest` return, which will trigger the delivery of actual page content to the client by means of `writeResponse`, the selected handler must, at some time, call `session.resume` with an appropriate response record.

This is exactly what `sendHtmlFormPage` does. In addition, it registers its own `return` continuation as a fresh request handler, saving the auto-generated ID as *conversation id* in a hidden form field. Upon submitting the form, it is the `return` continuation of this call to `sendHtmlFormPage` which will be invoked by `handleRequest`, resuming the interrupted program directly after the `sendHtmlFormPage` call.

```

new sendHtmlFormPage;

def sendHtmlFormPage(session, head, body, fields):
{
    targetID = registerHandler(return);

    inputFields = ""
    <tr>
      <td>${name}:</td>
      <td><input type="text" name="${name}" /></td>
    </tr>
    "" for name in fields;

    res.httpStatus = "200 OK";
    res.body = ""
    <html>
      <head>
        <title>${head}</title>
      </head>
      <body>
        <h1>${head}</h1>
        <p>${body}</p>
        <form method="POST">
          <input type="hidden" name="conversationID" value="${targetID}" />
          <table border="0">
            ${inputFields->strjoin()}
          </table>

```

```

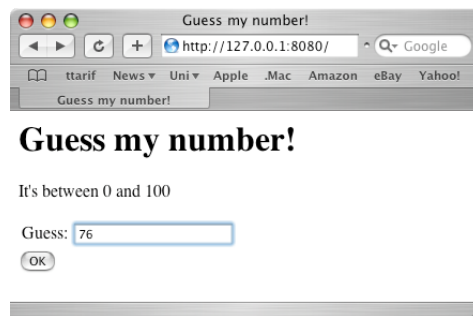
        <input type="submit" name="submit" value="OK" />
      </form>
    </body>
  </html>
  """;
  session.resume(res);
}

```

With this continuation-based programming model, opening multiple browser windows and using the back-button is supported automatically, since previously saved continuations can be resumed at any time and possibly more than once. At times, though, this may not be desired. In this number game, for example, we can cheat with the number of guesses by hitting the back-button and resubmitting the form:



... hitting the back-button a few times ...



Invalidating continuations is up to the application, though, as it is mainly a matter of business-logic. Implementation-wise, we could create *one-shot* continuations, or we could employ the software-transaction model mentioned in Section 2.2.3.

Chapter 6

Discussion

In the introductory sections, we have identified the lack of proper language support as a major hindrance to the development of concurrent interactive systems. Arguing that expressive languages with strong concurrency abstractions might help alleviate these problems, we set forth the goal to develop a language with concurrency as a fundamental feature but general enough to express common sequential programming in a natural way.

The chosen language design, which is closely related to the Join Calculus but with relaxed locality constraints and extended pattern matching that honors the *quality* of a match, is defined formally on a reduced core language. A formal semantics is given first in terms of generalized Petri Nets, then refined to interacting components. Keeping several aspects of the core model abstract makes the description amenable to possibly varying future instantiations.

Insight into which high-level concurrent programming abstractions are especially useful and valuable will only manifest by gaining experience with concurrent languages in actual software projects. The presented core language is sufficiently abstract to support a number of high-level constructs by encoding, including advanced object-oriented and functional approaches based on multimethods/open generic functions. Furthermore, we have shown syntactic sugar for handling sequential and parallel generators, which can be used not only for purposes of iteration but also for synchronizing interactive control-flow.

The full language and the supplied compiler/interpreter have been used to implement a number of non-trivial example programs with good success, expressing otherwise complicated solutions in a rather concise and modular way. The implementation, despite its reference character, shows good speedup numbers on multi-processor hardware and has reached a satisfying overall level of maturity.

6.1 Future Work

Nonetheless, designing let alone establishing a programming language and implementing industrial-quality tools is a massive undertaking that is clearly beyond the scope of a single thesis. Thus, there is a broad range of possible future work, both in practical and theoretical fields.

First, there are a number of desirable high-level enhancements, such as providing library code or IDE support (e.g. an Eclipse plug-in) to foster usability. On the syntax side, exception handling would be a desirable feature and it could be discussed whether the `&`-operator (which is only allowed in statements) could be dropped in favor of evaluating all `,`-delimited expressions in parallel.

Regarding the compiler implementation, it is always a good test for language expressiveness to implement the compiler in the language itself. This would not be such a big undertaking, as functionality could be carried over piece by piece. Since both Nice and our language provide multimethods, the code transformations would probably be the place to start. Along the same lines, it would be a natural advancement to generate Java byte-code for core language programs, employing native execution instead of interpretation.

A bit more removed would be the task of integrating our language with graphical software-design tools. By translating graphical models to our core language, which could be quite naturally done in the case of Petri Net based tools, we would effectively arrive at executable specifications.

Among the other issues that could be investigated is a persistence mechanism that would allow persistent storage of business processes like pending orders or shopping carts. Of interest would also be a security model, understanding function symbols as capabilities, like in the language E [Miller, 2006]. While it is clear that one can call a function symbol only if one has been sent the function symbol before, one can create new rules on it, too, so there would possibly be a need to prevent parasitic rule creation.

Regarding the use cases, we have left the field of real-time systems untouched. The first step in this direction would be to clarify the requirements and constraints of targeting embedded systems and coming up with an applicable instantiation of the core execution model. Another step would possibly consist in compiling to real machine code and building the execution framework as a native library.

A big step, both for type reconstruction/type inference (thus: safety) and program optimization (thus: execution speed) would be static program analysis. Static analysis is

difficult because of our language's non-determinism and higher-orderness. The classical way of first computing a program's control-flow graph and then its data-flow graph does not work here, because higher-order function symbols must be tracked as data values. This, of course, is not a new problem in the domain of dynamic languages and much could be borrowed from work on the analysis of Scheme or SmallTalk programs [Shivers, 1991, Agesen, 1995] and on compiling Python to C++ [Dufour, 2006].

Appendix A

Syntax Specification

A.1 Compilation Units

File → *Import** *Stm_{seq}**
Import → **import** *ImportRoot?* *ImportPath* ;
ImportRoot → *Identifier*
ImportPath → *Identifier* [*.* *Identifier*]*

A.2 Statements

Stm_{seq}	\rightarrow	<i>Block</i>	\rightarrow	$\{ Stm_{seq}^* \}$
		<i>Parallel</i>	\rightarrow	$[Stm_{par} \ \& \]^+ Stm_{parTail}$
		<i>TypeDef</i>	\rightarrow	type $Pat_{typedef}$ [is Val_{type}]?;
		<i>FunDef</i>	\rightarrow	new Pat_{fundef} [$: Val_{type} \rightarrow Val_{type}$]?;
		<i>RuleDef</i>	\rightarrow	def $Pat_{ruledef} : Stm_{ruledef}$
		<i>Declaration</i>	\rightarrow	$Pat_{dec} = Val_{dec}$;
		<i>For</i>	\rightarrow	for Pat_{for} in $Val_{forGen} : Stm_{compound}$
		<i>If</i>	\rightarrow	if $Val_{ifCond} : Stm_{compound}$
		<i>IfElse</i>	\rightarrow	if $Val_{ifCond} : Stm_{compoundIf}$ else $: Stm_{compound}$
		<i>PostfixFor</i>	\rightarrow	$Stm_{postfix}$ for Pat_{for} in Val_{forGen} ;
		<i>PostfixIf</i>	\rightarrow	$Stm_{postfix}$ if Val_{ifCond} ;
		<i>Call</i>	\rightarrow	$Val_{callFun}$ $Val_{callArg}$;
Stm_{par}	\rightarrow	<i>Block</i>	\rightarrow	$\{ Stm_{seq}^* \}$
		<i>Call</i>	\rightarrow	$Val_{callFun}$ $Val_{callArg}$
$Stm_{parTail}$	\rightarrow	<i>Block</i>	\rightarrow	$\{ Stm_{seq}^* \}$
		<i>Call</i>	\rightarrow	$Val_{callFun}$ $Val_{callArg}$;
$Stm_{ruledef}$	\rightarrow	<i>Block</i>	\rightarrow	$\{ Stm_{seq}^* \}$
		<i>Parallel</i>	\rightarrow	$[Stm_{par} \ \& \]^+ Stm_{parTail}$
		<i>Call</i>	\rightarrow	$Val_{callFun}$ $Val_{callArg}$;

$Stm_{compound}$	→	<i>Parallel</i>	→	$[Stm_{par} \&]^+ Stm_{parTail}$
		<i>Block</i>	→	$\{ Stm_{seq}^* \}$
		<i>For</i>	→	for Pat_{for} in $Val_{forGen} : Stm_{compound}$
		<i>If</i>	→	if $Val_{ifCond} : Stm_{compound}$
		<i>IfElse</i>	→	if $Val_{ifCond} : Stm_{compoundIf}$ else : $Stm_{compound}$
		<i>PostfixFor</i>	→	$Stm_{postfix}$ for Pat_{for} in $Val_{forGen} ;$
		<i>PostfixIf</i>	→	$Stm_{postfix}$ if $Val_{ifCond} ;$
		<i>Call</i>	→	$Val_{callFun} Val_{callArg} ;$
$Stm_{compoundIf}$	→	<i>Parallel</i>	→	$[Stm_{par} \&]^+ Stm_{parTail}$
		<i>Block</i>	→	$\{ Stm_{seq}^* \}$
		<i>For</i>	→	for Pat_{for} in $Val_{forGen} : Stm_{compoundIf}$
		<i>IfElse</i>	→	if $Val_{ifCond} : Stm_{compoundIf}$ else : $Stm_{compoundIf}$
		<i>PostfixFor</i>	→	$Stm_{postfix}$ for Pat_{for} in $Val_{forGen} ;$
		<i>PostfixIf</i>	→	$Stm_{postfix}$ if $Val_{ifCond} ;$
		<i>Call</i>	→	$Val_{callFun} Val_{callArg} ;$
$Stm_{postfix}$	→	<i>PostfixFor</i>	→	$Stm_{postfix}$ for Pat_{for} in Val_{forGen}
		<i>PostfixIf</i>	→	$Stm_{postfix}$ if Val_{ifCond}
		<i>Call</i>	→	$Val_{callFun} Val_{callArg}$

A.3 Value Expressions

Val_{dec}	\rightarrow	For	\rightarrow	$Val_{forBody}$ for Pat_{for} in Val_{forGen}
		$IfElse$	\rightarrow	if Val_{ifCond} then Val_{ifelse} else Val_{ifelse}
		$Tuple$	\rightarrow	Val_{tuple} [, Val_{tuple}] ⁺
		$Record$	\rightarrow	$Identifier$: Val_{record} [, $Identifier$: Val_{record}] ⁺
		$Lambda$	\rightarrow	fn Pat_{lambda} : Val_{lambda}
		$Disjunction$	\rightarrow	[$Disjunction$ $Val_{disjunction}$] $Val_{disjunction}$
		$Conjunction$	\rightarrow	[$Conjunction$ $Val_{conjunction}$] && $Val_{conjunction}$
		$Comparison$	\rightarrow	[$Comparison$ Val_{comp}] [= ! = < > < = > =] Val_{comp}
		$Range$	\rightarrow	[$Range$ Val_{range}] .. Val_{range}
		Sum	\rightarrow	[Sum Val_{sum}] [+ -] Val_{sum}
		$Product$	\rightarrow	[$Product$ $Val_{product}$] [* / %] $Val_{product}$
		$Negation$	\rightarrow	[- + !] $Val_{negation}$
		$Power$	\rightarrow	[$Power$ Val_{power}] ** Val_{power}
		$Partial$	\rightarrow	$Val_{partArg}$ -> $Val_{partFun}$
		$Call$	\rightarrow	$Val_{callFun}$ $Val_{callArg}$
		$Field$	\rightarrow	[$Field$ Val_{field}] . $Identifier$
		$Generator$	\rightarrow	[Val_{dec} [& Val_{dec}]*]
		$Group$	\rightarrow	(Val_{dec})
		$Atom$	\rightarrow	$Identifier$ $Literal$
Val_{type}	\rightarrow	$Field$	\rightarrow	Val_{field} . $Identifier$
		$Group$	\rightarrow	(Val_{dec})
		$Atom$	\rightarrow	$Identifier$ $Literal$

$Val_{forBody}$	\rightarrow	<i>For</i>	\rightarrow	$Val_{forBody}$ for Pat_{for} in Val_{forGen}
		<i>If</i>	\rightarrow	$Val_{forBody}$ if Val_{ifCond}
		<i>IfElse</i>	\rightarrow	if Val_{ifCond} then Val_{ifBody} else Val_{ifBody}
		<i>Tuple</i>	\rightarrow	...
		<i>Record</i>	\rightarrow	...
		...		
Val_{ifBody}	\rightarrow	<i>IfElse</i>	\rightarrow	...
		...		
Val_{tuple}	\rightarrow	<i>Lambda</i>	\rightarrow	...
		...		
Val_{record}	\rightarrow	<i>Lambda</i>	\rightarrow	...
		...		
Val_{lambda}	\rightarrow	<i>Disjunction</i>	\rightarrow	...
		...		
Val_{forGen}	\rightarrow	<i>Disjunction</i>	\rightarrow	...
		...		
Val_{ifCond}	\rightarrow	<i>Disjunction</i>	\rightarrow	...
		...		
$Val_{disjunction}$	\rightarrow	<i>Conjunction</i>	\rightarrow	...
		...		
$Val_{conjunction}$	\rightarrow	<i>Comparison</i>	\rightarrow	...
		...		
Val_{comp}	\rightarrow	<i>Range</i>	\rightarrow	...
		...		
Val_{range}	\rightarrow	<i>Sum</i>	\rightarrow	...
		...		
Val_{sum}	\rightarrow	<i>Product</i>	\rightarrow	...
		...		
$Val_{product}$	\rightarrow	<i>Negation</i>	\rightarrow	...
		...		
$Val_{negation}$	\rightarrow	<i>Power</i>	\rightarrow	...
		...		
Val_{power}	\rightarrow	<i>Partial</i>	\rightarrow	...
		...		

$$\begin{aligned}
Val_{partArg} &\rightarrow Partial \rightarrow Val_{partArg} \rightarrow Val_{partFun} \\
&| Call \rightarrow Val_{callFun} Val_{callArg} \\
&| Field \rightarrow \dots \\
&| \dots \\
Val_{partFun} &\rightarrow Field \rightarrow \dots \\
&| \dots \\
Val_{callFun} &\rightarrow Partial \rightarrow Val_{partArg} \rightarrow Val_{partFun} \\
&| Field \rightarrow \dots \\
&| \dots \\
Val_{callArg} &\rightarrow Call \rightarrow Val_{partFun} Val_{callArg} \\
&| Field \rightarrow \dots \\
&| \dots \\
Val_{field} &\rightarrow Group \rightarrow (Val_{dec}) \\
&| Atom \rightarrow Identifier | Literal
\end{aligned}$$

$$\begin{aligned}
Literal &\rightarrow Unit \rightarrow () \\
&| Bool \rightarrow \mathbf{true} | \mathbf{false} \\
&| Integer \rightarrow Digit^+ \\
&| Real \rightarrow Digit^+ . Digit^+ [[e | E] [- | +]? Digit^+]? \\
&| String \rightarrow " [Char^+ | \ Char | \$Identifier]^* " \\
&| """ [Char^+ | \ Char | EOL | " | \$Identifier | \$\{Val_{dec}\}]^* """
\end{aligned}$$

$$Identifier \rightarrow [[Letter | -] [Letter | - | Digit]^*] | [' Char^+ ']$$

A.4 Pattern Expressions

A.4.1 Patterns in Rule Definitions

$$\begin{aligned} Pat_{ruleDef} &\rightarrow Parallel \rightarrow [Pat_{ruleDefPar} \ \&]^+ Pat_{ruleDefPar} \\ &| Call \rightarrow Val_{callFun} Pat_{ruleArg} ; \end{aligned}$$

$$Pat_{ruleDefPar} \rightarrow Call \rightarrow Val_{callFun} Pat_{ruleArg} ;$$

$$\begin{aligned} Pat_{ruleArg} &\rightarrow Group \rightarrow (Pat_{ruleArgGroup}) \\ &| Field \rightarrow [Field | Identifier] . Identifier \\ &| Atom \rightarrow Identifier | Literal | ? \end{aligned}$$

$$\begin{aligned} Pat_{ruleArgGroup} &\rightarrow Tuple \rightarrow Pat_{ruleArgTuple} [, Pat_{ruleArgTuple}]^+ \\ &| Record \rightarrow Identifier : Pat_{ruleArgRecord} [, Identifier : Pat_{ruleArgRecord}]^+ \\ &| Typing \rightarrow Val_{type} Pat_{ruleArgTyping} \\ &| Group \rightarrow (Pat_{ruleArgGroup}) \\ &| Field \rightarrow [Field | Identifier] . Identifier \\ &| Atom \rightarrow Identifier | Literal | ? \end{aligned}$$

$$\begin{aligned} Pat_{ruleArgTuple} &\rightarrow Typing \rightarrow \dots \\ &| \dots \end{aligned}$$

$$\begin{aligned} Pat_{ruleArgRecord} &\rightarrow Typing \rightarrow \dots \\ &| \dots \end{aligned}$$

$$\begin{aligned} Pat_{ruleArgTyping} &\rightarrow Group \rightarrow \dots \\ &| \dots \end{aligned}$$

$$\begin{array}{l}
Pat_{for} \quad \rightarrow \quad Tuple \quad \rightarrow \quad Pat_{ruleArgTuple} [, Pat_{ruleArgTuple}]^+ \\
\quad \quad \quad | \quad Typing \quad \rightarrow \quad Val_{type} Pat_{ruleArgTyping} \\
\quad \quad \quad | \quad Group \quad \rightarrow \quad (Pat_{ruleArgGroup}) \\
\quad \quad \quad | \quad Field \quad \rightarrow \quad [Field | Identifier] . Identifier \\
\quad \quad \quad | \quad Atom \quad \rightarrow \quad Identifier | Literal | ?
\end{array}$$

$$\begin{array}{l}
Pat_{lambda} \rightarrow Tuple \quad \rightarrow \quad Pat_{ruleArgTuple} [, Pat_{ruleArgTuple}]^+ \\
\quad \quad \quad | \quad Typing \quad \rightarrow \quad Val_{type} Pat_{ruleArgTyping} \\
\quad \quad \quad | \quad Group \quad \rightarrow \quad (Pat_{ruleArgGroup}) \\
\quad \quad \quad | \quad Field \quad \rightarrow \quad [Field | Identifier] . Identifier \\
\quad \quad \quad | \quad Atom \quad \rightarrow \quad Identifier | Literal | ? \\
\quad \quad \quad | \quad Empty \quad \rightarrow
\end{array}$$

A.4.2 Patterns in Assignments

$$\begin{aligned}
 Pat_{dec} &\rightarrow Tuple \rightarrow Pat_{decTuple} [, Pat_{decTuple}]^+ \\
 &| Record \rightarrow Identifier : Pat_{decRecord} [, Identifier : Pat_{decRecord}]^+ \\
 &| Group \rightarrow (Pat_{dec}) \\
 &| Name \rightarrow [Name . Identifier] | Identifier
 \end{aligned}$$

$$\begin{aligned}
 Pat_{decTuple} &\rightarrow Group \rightarrow \dots \\
 &| Name \rightarrow \dots
 \end{aligned}$$

$$\begin{aligned}
 Pat_{decRecord} &\rightarrow Group \rightarrow \dots \\
 &| Name \rightarrow \dots
 \end{aligned}$$

$$Pat_{typedef} \rightarrow Name \rightarrow \dots$$

$$Pat_{fundef} \rightarrow Name \rightarrow \dots$$

Bibliography

- Adobe Systems. Adobe Flex, 2004. URL : <http://www.adobe.com/products/flex/>.
- O. Agesen. The cartesian product algorithm. In W. Olthoff, editor, *ECOOP '95: European Conference on Object-Oriented Programming, Århus, Denmark, August 1995. Proceedings*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 1995.
- G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.
- A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 11–13, 1989, Austin, TX*, pages 293–302. ACM Press, 1989.
- J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in Erlang*. Prentice Hall, 2nd edition, 1996.
- A. Asperti and N. Busi. Mobile Petri Nets. Technical Report UBLCS-96-10, Department of Computer Science, University of Bologna, 1996.
- B. V. Beneden. Examining the VxWorks AE 1.1 RTOS. *Dr. Dobb's Journal*, 27(11):66, 68, 70, November 2002.
- N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):769–804, 2004.
- G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
- R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

- R. D. Blumofe and D. Papadopoulos. The performance of work stealing in multiprogrammed environments. *ACM SIGMETRICS Performance Evaluation Review*, 26(1):266–267, 1998.
- R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.
- D. Bonniot. The Nice programming language, 2003a. URL : <http://nice.sourceforge.net/>.
- D. Bonniot. *Typage modulaire des multi-méthodes*. PhD thesis, École des Mines, Paris, 2005.
- D. Bonniot. Visitor pattern considered useless, 2003b. URL : <http://nice.sourceforge.net/visitor.html>.
- P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, 1975.
- P. Brinch Hansen. Java’s insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, 1999.
- M. Broy and K. Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Monographs in Computer Science. Springer, 2001.
- D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
- L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, 1992.
- R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.
- M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.
- F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *EW10: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop: Beyond the PC, Saint-Emilion, France, July 1, 2002*, pages 186–189. ACM Press, 2002.

-
- O. Dahl and K. Nygaard. *SIMULA — A language for programming and description of discrete event systems. Introduction and user's manual*. Norwegian Computing Center, 1967.
- L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *ECOOP' 87: European Conference on Object-Oriented Programming, Paris, France, June 15–17, 1987, Proceedings*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 1987.
- E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2): 115–138, 1971.
- E. W. Dijkstra. Cooperating sequential processes. In V. Genuys, editor, *Programming Languages*, pages 43–112. American Press, 1968.
- S. Ducasse, A. Lienhard, and L. Renggli. Seaside — a multiple control flow web application framework. In *Proceedings of 12th International Smalltalk Conference (ISC'04)*, pages 231–257, 2004.
- M. Dufour. *Shed Skin: An Optimizing Python-to-C++ Compiler*. Master's thesis, Parallel and Distributed Systems Group, Delft University of Technology, 2006.
- M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
- R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, 1999.
- C. Fournet and G. Gonthier. The Join Calculus: A language for distributed mobile programming. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures.*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2002.
- C. Fournet and G. Gonthier. The reflexive CHAM and the Join Calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.
- C. Fournet and L. Maranget. The Join Calculus language, 1997. URL : <http://join.inria.fr>.

- M. Fuchs. Escaping the event loop: an alternative control structure for multi-threaded guis. In L. J. Bass and C. Unger, editors, *Engineering for Human-Computer Interaction, Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction, Yellowstone Park, USA, August 1995*, volume 45 of *IFIP Conference Proceedings*, pages 69–87. Chapman & Hall, 1996.
- R. P. Gabriel. The design of parallel programming languages. In V. Lifschitz, editor, *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, pages 91–108. Academic Press, 1991.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
- A. Ghodsi and J. Armstrong. Apache vs. Yaws, 2001. URL : <http://www.sics.se/~joe/apachevsyaws.html>.
- Google. Google Docs & Spreadsheets, 2006. URL : <http://docs.google.com>.
- J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- P. T. Graunke and S. Krishnamurthi. Advanced control flows for flexible graphical user interfaces: or, growing guis on trees or, bookmarking guis. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 277–287. ACM Press, 2002.
- P. T. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In P. Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2618 of *Lecture Notes in Computer Science*, pages 238–252. Springer, 2003.
- W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.
- J. Halen, R. Karlsson, and M. Nilsson. Performance measurements of threads in Java and processes in Erlang. Technical Report ETX/DN/SU-98:024, Ericsson, 1998. URL : <http://www.sics.se/~joe/ericsson/du98024.html>.
- P. Haller and M. Odersky. Event-based programming without inversion of control. In D. E. Lightfoot and C. A. Szyperski, editors, *Modular Programming Languages, 7th*

-
- Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006 Proceedings*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.
- T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *PPoPP '05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM Press, 2005.
- R. Haux, C. Seggewies, W. Baldauf-Sobez, P. Kullmann, H. Reichert, L. Luedecke, and H. Seibold. Soarian — workflow management applied for health care. *Methods of Information in Medicine*, 24(1):25–36, 2003.
- C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In N. J. Nilsson, editor, *IJCAI '73: Third International Joint Conference on Artificial Intelligence*, pages 235–245. William Kaufmann, 1973.
- R. Hitchens. *Java Nio*. O'Reilly & Associates, 2002.
- C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
- M. Hoy, D. Wood, M. Loy, J. Elliot, and R. Eckstein. *Java Swing*. O'Reilly & Associates, 2002.
- G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
- K. Jensen. *Coloured Petri Nets. Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer, 1992.
- H. Kalogirou. Multithreaded game scripting with Stackless Python, August 2005. URL : <http://harkal.sylphis3d.com/2005/08/10/multithreaded-game-scripting-with-stackless-python/>.
- B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1978.
- O. Kiselyov. General ways to traverse collections, 2004. URL : <http://okmij.org/ftp/Scheme/enumerators-callcc.html>.
- J. Kodosky. Is LabVIEW a general purpose programming language?, 2006. URL : <http://zone.ni.com/devzone/cda/tut/p/id/5313>.

- Laszlo Systems. OpenLaszlo, 2005. URL : <http://www.openlaszlo.org>.
- F. Le Fessant. *JoCaml: conception et implémentation d'un langage à agents mobiles*. PhD thesis, École Polytechnique, Paris, 2001.
- D. Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43. ACM Press, 2000.
- E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
- B. Liskov. A history of CLU. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 133–147. ACM Press, 1993.
- Y. Matsumoto. The Ruby programming language, 1997. URL : <http://www.ruby-lang.org>.
- B. Meyer. The next programming frontier. *Bertrand Meyer's EiffelWorld column*, June 2005. URL : http://www.eiffel.com/general/monthly_column/2005/June.html.
- Microsoft Corporation. *C# Language Specification*. Microsoft Press, 2001.
- M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, John Hopkins University, Baltimore, 2006.
- R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1982.
- R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–40, 1992.
- K. H. Mortensen. Efficient data-structures and algorithms for a Coloured Petri Nets simulator. In K. Jensen, editor, *3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, volume DAIMI PB-554, pages 57–74. University of Aarhus, 2001.
- M. Odersky. Functional Nets. In G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 – April 2, Proceedings*, volume 1782 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.

-
- M. Odersky. The Scala experiment: can we provide better language support for component systems? In J. G. Morrisett and S. Peyton Jones, editors, *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 11-13, 2006*, pages 166–167. ACM Press, 2006.
- M. Odersky, G. Chen, C. Zenger, and M. Zenger. A functional view of Join. Technical Report ACRC-99-016, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, 1999.
- S. Parkinson-Bates. The glutmech OpenGL example, 2001. URL :
[http://www.opengl.org/resources/code/samples/glut_examples/demos/
glutmech/glutmech.c](http://www.opengl.org/resources/code/samples/glut_examples/demos/glutmech/glutmech.c).
- C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Universität Bonn, 1962.
- S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, 1996.
- B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI-476, Computer Science Department, Indiana University, 1997.
- D. Quan, D. Huynh, D. R. Karger, and R. Miller. User interface continuations. In *UIST '03: Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology*, pages 145–148. ACM Press, 2003.
- A. V. Ratzert, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN/Tools for editing, simulating, and analysing coloured petri nets. In W. M. P. van der Aalst and E. Best, editors, *Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003, Proceedings*, volume 2679 of *Lecture Notes in Computer Science*, pages 450–462. Springer, 2003.
- W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer, 1985.
- J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
- T. Rompf. The Vodka programming language project, 2007. URL :
<http://code.google.com/p/vodka/>.

- D. C. Sastry and M. Demirci. The QNX operating system. *Computer*, 28(11):75–77, 1995.
- A. Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison-Wesley, 1996. URL : <http://www.opendylan.org/books/drm/>.
- O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.
- J. Stier, J. Jahnke, and H. Müller. Geist3D, a simulation tool for geometry-driven Petri Nets. In S. Donatelli and P. S. Thiagarajan, editors, *Petri Nets and Other Models of Concurrency – ICATPN 2006, 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Turku, Finland, June 26-30, 2006, Proceedings*, volume 4024 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 2006. URL : <http://www.geist3d.org>.
- H. Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), March 2005.
- S. T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, volume 1246 of *Lecture Notes in Computer Science*. Springer, 1997.
- The Apache Software Foundation. The Apache Cocoon project — control flow, 2006. URL <http://cocoon.apache.org/2.1/userdocs/flow/continuations.html>.
- G. van Rossum. The Python programming language, 1994. URL : <http://www.python.org>.
- P. van Roy, editor. *Multiparadigm Programming in Mozart/OZ: Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.
- G. S. von Itzstein. *Introduction of High Level Concurrency Semantics in Object Oriented Languages*. PhD thesis, School of Computer and Information Science, University of South Australia, 2004.
- P. Wadler. The expression problem. Discussion on the Java-Genericity mailing list, December 1998.