Copyright © 2003, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

CAL LANGUAGE REPORT
Specification of the Cal Actor Language

by

Johan Eker and Jörn W. Janneck

Memorandum No. UCB/ERL M03/48

1 December 2003

CAL LANGUAGE REPORT
Specification of the Cal Actor Language

by

Johan Eker and Jörn W. Janneck

Memorandum No. UCB/ERL M03/48

1 December 2003

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CAL Language Report

Specification of the CAL actor language

Johan Eker
Jörn W. Janneck

language version 1.0 — document edition 1

ERL Technical Memo
University of California at Berkeley
December 1, 2003

Contents

Contents 3

1 **Introduction** 9
- 1.1 Actors and actor composition 9
- 1.2 Language design: goals and principles 11
- 1.3 Platform independence and compatibility 12

I Language description 15

2 **Introductory remarks** 17
- 2.1 Lexical tokens 17
- 2.2 Typographic conventions 19
- 2.3 Syntax rules 19
- 2.4 Notational idioms 19

3 **Structure of actor descriptions** 21
- 3.1 Namespaces and imports 22
- 3.2 Time 23

4 **Data types** 25
- 4.1 Objects, variables, and types 25
- 4.2 Type formats 27
- 4.3 Required types 28
- 4.4 Structured objects and their types 29
- 4.5 Mutable objects and their types 30
- 4.6 Type framework 31

5 **Variables** 33
- 5.1 Variable declarations 33
  - 5.1.1 Explicit variable declarations 34
- 5.2 Variable scoping 35

6 Expressions 41
6.1 Literals 42
6.2 Variable references 42
6.2.1 Old variable references 42
6.3 Function application 43
6.4 Field selection 45
6.5 Indexing 45
6.6 Operators 45
6.7 Conditional expressions 46
6.8 Introducing a local scope 46
6.9 Closures 46
6.9.1 Lambda-expressions and function closures 47
6.9.2 Proc-expressions and procedure closures 47
6.9.3 Function and procedure declarations 48
6.10 Comprehensions 48
6.10.1 Simple collection expressions 49
6.10.2 Comprehensions with generators 50
6.11 Type assertion 53

7 Statements 55
7.1 Assignment 55
7.1.1 Simple assignment 56
7.1.2 Field assignment 56
7.1.3 Assignment with indices 56
7.1.4 Assigning to and from mutable variables 57
7.2 Procedure call 58
7.3 Statement blocks (begin ... end) 59
7.4 If-Statement 59
7.5 While-Statement 59
7.6 Foreach-Statement 60
7.7 Choose-Statement 61

8 Actions 65
8.1 Input patterns, and variable declarations 66
8.1.1 Single-port input patterns 67
8.1.2 Multiport input patterns 68
8.1.3 Scoping of action variables 69
8.2 Output expressions 70
8.3 Delays 73
8.4 On action selection: guards and other activation conditions . . . 73
8.5 Initialization actions 74

9 **Action-level control structures** 77
  9.1 Action tags 78
  9.2 Action schedules 79
    9.2.1 Finite state machine schedules 79
    9.2.2 Regular expression schedules 80
  9.3 Priorities 82

II Semantics **85**

10 **Actor model** 86
  10.1 Preliminaries 86
  10.2 Time systems 87
  10.3 Actor transition systems 89

III Appendices **91**

A **CAL language syntax** 93
  A.1 Actor 93
  A.2 Expressions 94
  A.3 Statements 95
  A.4 Actions 96
  A.5 Action control 97

B **Keywords** 99

C **Basic runtime infrastructure** 103
  C.1 Operator symbols 103
  C.2 Basic data types and their operations 104
    C.2.1 `Collection[T]`—collections 105
    C.2.2 `Seq[T]`—sequences 107
    C.2.3 `Set[T] < Collection[T]`—sets 107
    C.2.4 `List[T] < Collection[T], Seq[T]`—lists 109
    C.2.5 `Map[K, V]`—maps 109
    C.2.6 `Number`—numbers 110
    C.2.7 `Character`—characters 110
    C.2.8 `String < List[Character]`—strings 110
    C.2.9 `Boolean`—truth values 111
    C.2.10 `Null`—the null value 111

Index 115

Bibliography 127

First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. Second, we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming-language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems.
[...]
Underlying our approach to this subject is our conviction that "computer science" is not a science and that its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is what might best be called *procedural epistemology*—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is." Computation provides a framework for dealing precisely with notions of "how to"

Harold Abelson, Gerald Jay Sussman
Structure and Interpretation of Computer Programs [2]

What a thing means is simply what habits it involves [...] there is no distinction of meaning so fine as to consist in anything but a possible difference in practice.

Charles S. Pierce [13]

Chapter 1

Introduction

This report describes CAL, an actor language created as a part of the Ptolemy II project [1] at the UC Berkeley. It is intended primarily as a repository for technical information on the language and its implementation and contains very little introductory material. After a short motivation, we will outline the goals and the guiding principles of the language design. We will also give a short outline of the actor model, and the context that the actors written in CAL are embedded into, describing the kinds of assumptions an actor may and may not, in general, make about it.

1.1 Actors and actor composition

Actors. The concept of *actor* as an entity that is composed with other actors to form a concurrent system has a rich an varied history—some important milestones include [6], [9], [3], [4], [5]. A formal description of the notion of *actor* underlying this work can be found in chapter 10, which is based on the work in [10] and [7]. Intuitively, an *actor* is a description of a computation on sequences of *tokens* (atomic pieces of data) that produces other sequences of tokens as a result. It has *input ports* for receiving its input tokens, and it produces its output tokens on its *output ports*.

The computation performed by an actor proceeds as a sequence of atomic steps called *firings*. Each firing happens in some actor *state*, consumes a (possibly empty) prefix of each input token sequence, yields a new actor state, and produces a finite token sequence on each output port.[1]

firing

Several actors are usually composed into a *network*, a graph-like structure (often referred to as a *model*) in which output ports of actors are connected to input ports of the same or other actors, indicating that tokens produced at those output ports are to be sent to the corresponding input ports. Such actor

composition of actors

---

[1] The notion of actor and firing is based on the one presented in [10], extended by a notion of state in [7].

networks are of course essential to the construction of complex systems, but we will not discuss this subject here, except for the following observations:

decoupling from communication model

- A connection between an output port and an input port can mean different things. It usually indicates that tokens produced by the former are sent to the latter, but there are a variety of ways in which this can happen: token sent to an input port may be queued in FIFO fashion, or new tokens may 'overwrite' older ones, or any other conceivable policy. It is important to stress that actors themselves are oblivious to these policies: from an actor's point of view, its input ports serve as abstractions of (prefixes of) input sequences of tokens, while its output ports are the destinations of output sequences.

decoupling from actor scheduling

- Furthermore, the connection structure between the ports of actors does not explicitly specify the order in which actors are fired. This order (which may be partial, i.e. actors may fire simultaneously), whether it is constructed at runtime or whether it can be computed from the actor network, and if and how it relates to the exchange of tokens among the actors—all these issues are part of the interpretation of the actor network.

communication + scheduling = model of computation

The interpretation of a network of actors determines its *semantics*—it determines the result of the execution, as well as how this result is computed, by regulating the flow of data as well as the flow of control among the actors in the network. There are many possible ways to interpret a network of actors, and we call any specific interpretation a *model of computation*—the Ptolemy project focuses on exploring the issues of models of computation and their composition, cf. [11, 12]. Actor composition inside the actor model that CAL is based on has been studied in [8].

As far as the design of a language for writing actors is concerned,the above definition of an actor and its use in the context of a network of actors suggest that the language should allow to make some key aspects of an actor definition explicit. These are, among others:

- The port signature of an actor (its input ports and output ports, as well as the kind of tokens the actor expects to receive from or be able to send to them.
- The code executed during a firing, including possibly alternatives whose choice depends on the presence of tokens (and possibly their values) and/or the current state of the actor.
- The production and consumption of tokens during a firing, which again may be different for the alternative kinds of firings.
- The modification of state depending on the previous state and any input tokens during a firing.

Actor-like systems. It is often useful to abstract a system as a structure of cooperating actors. Many such systems are *dataflow*-oriented, i.e. they consist of components that communicate by sending each other packets of information, and whose ability to perform computation depends on the availability of sufficient input data. Typical signal processing systems, and also many control system fall into this category.

Writing actors is hard. Writing an actor in a general-purpose programming language is of course possible, but most or all of the information that may be used to reason about its behavior is implicit in the program and can only be extracted using sophisticated analysis, if this is at all feasible.

Furthermore, actors often need to be run on different platforms. For instance, if actors are used in the design of an embedded system, they need to run in a modeling and simulation environment (such as Matlab or Ptolemy) as well as in the final product. Being able to use the same description of the actor functionality in both cases improves productivity and reduces the probability of errors.

1.2 Language design: goals and principles

Designing a programming language is an exercise in balancing a number of sometimes contradicting goals and requirements. The following were the ones that guided the design of CAL.

Ease of use. CAL is intended to be a programming language, not an intermediate format or a representation for automatically generated code. Since we want people to actually write code in it, the notation must be reasonably convenient to write, with consistent syntax rules, keywords, and structures. Because people make mistakes, it needs to be sufficiently redundant to allow effective error detection and localization, but simple and concise enough for frequent use, especially in frequently used areas.

Minimal semantic core. In spite of being a full-fledged programming language, we wanted to build CAL on a very small set of semantic concepts, for a number of reasons. First of all, being able to describe a large part of the full language through reductions to a smaller language makes the definition of language semantics much easier. From a practical perspective, this simplifies compiler construction—if there is a generic procedure that transforms any program into an equivalent program in the core language, then all it takes in order to compile the full language to any given platform is a code generator for the core language. This led to the design of a number of core languages of CAL (reflecting several levels of reduction) that much of the implementation of the language is based on, and which will be described in another place.

Focus and specificity. CAL is a domain-specific language that is aimed at providing a medium for defining actors. It was very important to draw a clear line between those pieces of functionality that were deemed to be part of an actor definition and those that were not. For example, in addition to clearly actor-specific structures such as actions and input/output patterns/expressions, expressions and statements were considered to be essential to defining an actor. On the other hand, there are many things that CAL explicitly does *not* contain, such as facilities for defining new types, concepts for connecting actors, or mechanisms to aggregate actors into composites. The fact that CAL is explicitly agnostic about these issues makes it possible to use the language in a wide variety of contexts, which may provide very different designs in those areas.

Implementation independence and retargetability. Even though our first target for CAL actors is the Ptolemy II platform, we want the language to be retargetable, in the following two senses: First, we would like to be able to take an actor written, say, for Ptolemy II and be able to compile it to some other platform, say to some C code that runs against a different API. Secondly, we would like to enable other people to embed CAL into entirely different, but still actor-like, contexts, which have different kinds of objects (and types), different libraries, different primitive data objects and operators. Here, we would not necessarily try to reuse the actor libraries written for other platforms (although some interesting subset might still be sufficiently generic to be reusable)—instead, we would reuse the CAL framework, i.e. its infrastructure such as parsers, transformers and annotators, verification and type checking, code generation infrastructure etc. This is why CAL does not have a type system of its own, but relies on the environment to provide one (cf. section 4 for more information). We hope that this will allow the use of CAL in a very wide range of contexts, from full-fledged component models (such as JavaBeans) to very resource-constraint embedded platforms.

Making relevant design knowledge explicit and manifest. The key goal of CAL is to enable the author of an actor to express some of the information about the actor and its behavioral properties that are relevant to using the actor (e.g. the verify its appropriate use inside a model, or to generate efficient code from it), but that would be only implicit in a description of the actor in a 'traditional' programming language such as C or Java.

1.3 Platform independence and compatibility

notions of *platform independence*

CAL is intended to be adaptable to a variety of different platforms. There are a number of ways to interpret the term 'platform independence', and since this is a very important aspect of the design of CAL, we will discuss our approach to this issue in this section.

code portability

For example, it could mean that *code* written in a language can be run on a variety of platforms (which is the interpretation chosen, e.g., in the implementation of the Java programming language). One common approach to achieve code independence would be to define a virtual platform that promises to be implementable in various environments. If this platform is a runtime platform (rather than a source-code level API), this makes not only the source code, but also of the compiled target portable across platforms. This has the obvious advantage of (at least potentially) making every piece of code that is ever written against the virtual platform portable to any implementation, increasing reuse and avoiding fragmentation of the user base. There are, however, downsides to this approach. First, it requires a delicate balance between including all possibly desirable features and selecting those that can be implemented on a wide variety of platforms. If the target platforms are reasonably similar, this may not be a problem, but to the extent that the targets vary, some of them may prevent the inclusion of features that would be very useful on others, resulting in a greatest common denominator design. Second, requiring code and APIs to be independent of any specific platform also makes it harder or impossible to take advantage of platform-specific features.

source vs target code portability

Another interpretation of the term focuses on the *language* and its concepts, rather than the code written in the language. The C language is an example of this: it provides a set of basic concepts, but it leaves many details open to specific implementations, such the sizes and representations of basic data types, and of course the set of library functions used to create programs. As a result, C code itself is not portable, but relies on the presence of a specific set of libraries, and may rely on a specific representation of the data objects. Of course, techniques exist to improve code portability, such as standardization of library sets, and abstraction mechanisms that deal with different data representations. But the general problem with this approach is, of course, that code written in a language is not automatically portable. The advantage, however, is that the language as well as code written in it may be tuned to exploit the specific features of a platform.

language portability

The design of CAL tries to realize this latter form of portability. The reason is that we intend the language to be used on wide variety of different platforms, and that we do not believe that there is a single abstraction that does justice to all of them—particularly because for some of them, performance of the generated code is a very high priority.

portability in CAL

Nonetheless, portability of source code as well as target code remains a concern. We intend to improve source code portability by defining *profiles* for certain classes of platforms, which define things like type systems, basic function and procedure libraries and the like. But these will be based on our experiences with the language in various scenarios, and thus are a second step in the development of the language and its environment. As for target code portability, it seems more reasonable to use existing infrastructure (such as the Java VM, or the Common Language Runtime) wherever possible, rather than developing one from scratch.

platform profiles

targeting existing virtual platforms

RATIONALE.
Throughout this report, selected aspects of the language design will be discussed in these boxes, presenting the rationale for the decisions made in the specific cases.
These discussions are mainly of interest to language implementors and people interested in the language design. Users only interested in the mechanics of the language may safely ignore them.

IMPLEMENTATION NOTE.
Similarly, discussions of implementation techniques, or aspects of the language that require particular consideration when implementing the language on some platform are set in these boxes.

Part I

Language description

Chapter 2

Introductory remarks

Throughout this part, we will present fragments of CAL syntax along with (informal) descriptions of what these are supposed to mean. In order to avoid ambiguity, we will now introduce a few conventions as well as the fundamental syntactic elements (lexical tokens) of the CAL language.

2.1 Lexical tokens

CAL has the following kinds of lexical tokens:

Keywords. Keywords are special strings that are part of the language syntax and are consequently not available as identifiers. See B for a list of keywords in CAL.

Identifiers. Identifiers are any sequence of alphabetic characters of either case, digits, the underscore character and the dollar sign that is not a keyword. Sequences of characters that are not legal identifiers may be turned into identifiers by delimiting them with backslash characters.

Identifiers containing the $-sign are *reserved* identifiers. They are intended to be used by tools that generate CAL program code and need to produce unique names which do not conflict with names chosen by users of the language. Consequently, users are discouraged from introducing identifiers that contain the $-sign.

Operators. Operators are written as any string of characters !, @, #, $, %,^, &, *, /, +, -, =, <, >, ?, ~, —. In addition to these strings, a few keywords are used as operators. The set of operators in CAL is extensible, different platforms may augment the set of operators. The language itself defines only a small set of operators—see appendix C.1 for a list of predefined operators.

Delimiters. These are used to indicate the beginning or end of syntactical elements in CAL. The following characters are used as delimiters: (,), {, }, [,], :.

Comments. Comments are Java-style, i.e. single-line comments starting with "//" and multi-line comments delimited by "/*" and "*/".

Numeric literals. CAL provides two kinds of numeric literals: those representing an integral number and those representing a decimal fraction. Their syntax is as follows:[1]

Integer → DecimalLiteral | HexadecimalLiteral | OctalLiteral

Real → DecimalDigit { DecimalDigit } '.' { DecimalDigit } [Exponent]
  | '.' DecimalDigit { DecimalDigit } [Exponent]
  | DecimalDigit { DecimalDigit } Exponent

DecimalLiteral → NonZeroDecimalDigit { DecimalDigit }

HexadecimalLiteral→ '0' ('x'|'X') HexadecimalDigit { HexadecimalDigit }

OctalLiteral → '0' { OctalDigit }

Exponent → ('e'|'E') ['+'|'-'] DecimalDigit { DecimalDigit }

NonZeroDecimalDigit → '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

DecimalDigit → '0'| NonZeroDecimalDigit

OctalDigit → '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'

HexadecimalDigit → DecimalDigit
  | 'a'|'b'|'c'|'d'|'e'|'f'
  | 'A'|'B'|'C'|'D'|'E'|'F'

RATIONALE.

The reason for allowing identifiers to essentially be any sequence of characters (by providing an 'escaped' identifier syntax) is that CAL is intended to interoperate with as many other languages as possible, and therefore cannot assume any particular identifier syntax. We expect that most applications will be using C/C++/Java-style host environments, and thus the lexical conventions of CAL are very similar to those found in these languages. But we did not want to exclude other environments just by a too restrictive choice of the identifier space.

---

[1] In contrast to all other grammar rules in this report, the following rules do not allow whitespace between tokens.

2.2 Typographic conventions

In syntax rules, keywords are shown in **boldface**, while all other literal symbols are enclosed in single quotes.

In examples, CAL code is represented `monospaced` if it is to be read literally. Placeholders inside CAL code, such as for expressions, variables, statements and so forth, are set in italics, as e.g., E, v, and S.

2.3 Syntax rules

We use a form of BNF to describe the syntax rules. Literal elements are put in quotes (in the case of symbols and delimiters), or set in boldface (in the case of keywords). An optional occurrence of a sequence of symbols A is written as $[A]$, while any number of consecutive occurrences (including none) are written as $\{A\}$. The alternative occurrence of either A or B is expressed as $A \mid B$.

We often use plural forms of non-terminal symbols without introducing them explicitly. These are supposed to stand for a comma-separated sequence of at least on instance of the non-terminal. E.g., if A is the non-terminal, we might use As in some production, and we implicitly assume the following definition:

As → A { ',' A }

In the examples we will give in this report, we will assume the 'usual' interpretation of expression literals and mathematical operators, even though strictly speaking these are not part of the language and depend on the environment. A specific implementation of CAL may not have these these operators, or interpret them or the literals in a different manner.

2.4 Notational idioms

Like most programming languages, CAL involves a fair number of syntactical constructs that need to be learned and understood by its users in order to use the language productively. The effort involved in gaining familiarity with the language can be a considerable impediment to its adoption, so it makes sense to employ general guidelines for designing the syntax of constructs, which allow users to make guesses about the syntax if they are unsure about the details of a specific language construction. We call these guidelines, which define the *style* of a language, its *notational idioms*.

The following is a list of notational idioms guiding the design of CAL's language syntax. Not all of them may hold true all the time, but we tried to conform to them whenever reasonable.

Keyword constructs. Many constructs in CAL are delimited by *keywords* rather than more symbolic delimiters—such constructs are called *keyword constructs*. Examples of these would be lambda-expressions (see section 6.9.1). Other constructs are delimited by symbols (e.g. comprehensions, see section 6.10), or are at least partially lacking delimiters (such as assignments, which begin with a variable name, see section 7.1).

Alternative end markers. Every keyword construct ends with either the keyword **end** or the keyword that consists of appending the opening keyword to **end**. For instance, a lambda-expression can either be written

```
lambda (x) :  x * x end
```

or alternatively

```
lambda (x) :  x * x endlambda
```

Expression head/body separator. Composite expressions often consist of the opening keyword, a *head*, a *separator*, the *body*, and the closing end marker keyword. In such expressions, the separator is usually the ':'-character, as in the following examples:

```
let x = f(k) :  g(x, x) end
```

or

```
lambda (x) :  x + x end
```

Statement head/body separator. Many statements have a similar structure as the one for expressions. For statements, the keywords **do** or **begin** are used as a separator:

```
while n > 0 do k := f(k); n := n - 1; end
```

or

```
procedure p (x) begin
    println("Result:  " + x.toString());
end
```

Chapter 3

Structure of actor descriptions

Each actor description defines a named *kind* of actor. Actors may refer to entities defined in the implementation context, which provides a hierarchical namespace for these entities, see section 3.1 for details. Actor descriptions may use import declarations to use parts of this namespace or the objects defined in it as part of their global environment.

Actors are the largest lexical units of specification and translation. The basic structure of an actor is this:

Actor → [Imports] **actor** ID
    ['[' TypePars ']'] '(' ActorPars ')' IOSig [TimeClause] ':'
    { VarDecl | Action | InitializationAction | PriorityBlock }
    [ActionSchedule]
    { VarDecl | Action | InitializationAction | PriorityBlock }
    (**end**|**endactor**)

TypePar → ID ['<' Type]

ActorPar → [Type] ID ['=' Expression]

IOSig → [PortDecls] '==>' [PortDecls]

PortDecl → [**multi**] [Type] ID

TimeClause → **time** Type

The header of an actor contains optional type parameters and actor parameters, and its port signature. This is followed by the body of the actor, containing a sequence of state variable declarations (section 5.1), actions (chapter 8), initialization actions (section 8.5), priority blocks (section 9.3), and at most one action schedule (section 9.2).

Type parameters are variable symbols that are bound to types when the actor is instantiated. They can be used like any other type inside the actor

type parameters

type bounds

definition. Each type parameter may be optionally *bounded*, i.e. they may be associated with some type. In this case, the actual type that this parameter is instantiated to is required to be a subtype of the bound (including the bound itself).

actor parameters

By contrast, actor parameters are *values*, i.e. concrete objects of a certain type (although, of course, this type may be determined by a type parameter). They are bound to identifiers which are visible throughout the actor definition. Conceptually, these are non-assignable and immutable, i.e. they may not be assigned to by an actor.

IMPLEMENTATION NOTE.
A specific implementation such as the one in Ptolemy might change these parameters, for example in response to user interaction during design. For this to make sense in CAL, the implementation has to ensure the consistency of the actor state with the new parameter values, which it usually does by reinitializing the actor whenever one of its parameters is assigned a new value.

single ports vs. multiports

The port signature of an actor specifies the input ports and output ports, including their names, whether the port is a multiport or a single port, and the type of the tokens communicated via the port. While single ports represent exactly one sequence of input or output tokens, multiports are comprised of any number of those sequences (called *channels*), including zero.

3.1 Namespaces and imports

global environment

An actor description may contain free variables, i.e. references to variables not defined inside the actor. Often, these are functions or procedures, but also types, which are predefined as part of the respective implementation context. The collection of all globally visible variable bindings is called the *global environment* of an actor.

hierarchical context namespace

imports assemble global environment

However, implementation contexts may be very rich, providing a large number of functions, procedures, and types for actor writers to use. In such cases it would be inappropriate to define all of these as global environment—it would lead to a very large number of variable names, only a very small part of which would actually be used by each actor definition. For this reason, implementation contexts may use a hierarchical namespace for naming these entities, where each entity is denoted by a sequence of identifiers separated by dots (a so-called *qualified identifier*). Actor specifications may use them as part of their global environment by *importing* them into it. Effectively, one can think of import declarations as constructing the global environment of an actor description, starting with the default global environment, and adding bindings to it.

The qualified identifiers that denote each entity in the hierarchical namespace have two parts: the (possibly empty) sequence of identifiers up to and excluding the last, and the last identifier. The first part is called the *subnamespace* or *package*, while the second is called the *local name*. For example, in the qualified identifiers `X.Y.Z`, `XYZ`, and `java.util.HashMap`, the subnamespaces are `X.Y`, λ, and `java.util`, respectively,[1] while the corresponding local names are `Z`, `XYZ`, and `HashMap`.

qualified id = subnamespace + local name

An import declaration can either make a single entity available as the value of a global variable, or the group of all entities inside the same subnamespace.

single vs group imports

Import → SingleImport | GroupImport ';'

SingleImport → **import** QualID ['=' ID]

GroupImport → **import all** QualID

QualID → ID { . ID }

For a single import declaration, the qualified identifier denotes the entity to be imported into the global environment. If the optional identifier following it after an '=' is omitted, the entity denoted by the qualified identifier is imported under its local name. For instance, the import declaration
`import A.B.C;`
imports the entity denoted by `A.B.C` under the name `C` into the global environment. If an identifier is specified, it will be the name of the specified entity:
`import A.B.C = D;`
imports the same entity under the name `D`.

Group import declarations import entire groups of entities. In this case, the qualified identifer specifies the subnamespace, and all entities in that subnamespace are imported under their local names. For example,
`import all A.B;`
imports `A.B.C` as `C` and `A.B.E` as `E`, if these are the two entities in that subnamespace.

3.2 Time

CAL supports an abstract notion of *time* as a way to relate the "duration" of various actor firings to each other, and thus potentially control or constrain the concurrency in a model that consists of several actors. It is essential to the division of responsibility between an actor and its environment that the actor itself does not interpret this time information, as it impacts the coordination of more than one actor, rather than the execution of an isolated actor. For this reason, time is purely declarative in CAL, and the language itself does not attach any meaning to it whatsoever, other than it being a property of actor transitions.

time semantics external to actor

---

[1] λ denotes the empty sequence of identifiers.

time system:
tags, delays, addition, zero delay

In section 10.2 the *time system* of an actor is introduced as an algebraic structure that has

- a set of *time tags*
- a set of *time delays*
- a (partial) *order* on the tag set
- an *addition operation* that adds delays to tags to yield larger tags
- a *zero delay* that is the neutral delay with respect to that addition.

delay type

The actor specification, however, only contains the delays. For this reason, the optional **time**-tag in the actor header is only followed by one type, the type of the delay values in action descriptions (see section 8.3). This type must have the following properties:

- There must be a partial order that is compatible with the addition operation used to add the time delay to tags. More precisely, for any tag t, and any two delays d_1, d_2, the following must hold:

$$d_1 < d_2 \Longrightarrow (t + d_1) < (t + d_2)$$

- There must be a zero delay, say z, which defines the set of valid delays of the specified type as follows. If d is of the delay type, it is valid iff $z \leq d$.

time-clause: static type declaration

The time-clause in the actor head functions as a type declaration for the delays. Its presence does *not* imply that any or all actions have non-zero delays, and neither does its absence imply that all actions have zero delays—unless a platform requires type information to be added, in which case an actor that contained delay-clauses in actions but no time-clause would not typecheck and hence not be well-formed.

See section 8.3 on how to specify time delays in actions.

Chapter 4

Data types

CAL is *optionally typed*, i.e. it allows programmers to give each newly introduced identifier a type (see section 5.1 for more details on declaring variables), but it does not require it, in which case the identifier is said to be untyped. In general, the type system is considered part of the external environment that we try to keep CAL actor specifications orthogonal to. In this chapter we will therefore discuss primarily the syntax provided for writing types, leaving the concrete interpretation to the description of CAL implementations on individual platforms.

optionally typed

However, CAL does assume a few basic types (some of which are parametric), viz. those that are used as part of some language constructions. There are also some framework rules on how to type checking/inference is to be conducted, which will be discussed in section 4.6.

minimal type system

4.1 Objects, variables, and types

In general, there are really two kinds of types—the types with which variables are declared (*variable types*),[1] and the types of runtime objects (*object types*). In most languages, these are either the same sets, or there is a significant overlap. However, even in common languages, these sets are not identical: in Java, e.g., abstract classes and interfaces are only variable types, never the types of objects.

---

[1] We use *variable* here to mean any name inside the language, including parameters etc.

RATIONALE.
Most programming languages either require explicit typing, or they do not have constructs for expressing types as part of the program source. Some languages perform *type inference*, i.e. they allow users to omit type declarations as long as they can infer enough information from the context to guarantee the safeness of the program.

Making types in CAL optional reflects our choice to allow for a wide range of embeddings and applications of the language. CAL could thus be used as a scripting language that is interpreted and completely untyped, or as a language that is used to generate C code or hardware from, and which requires complete typing.

Tools are free to reject insufficiently typed programs, or to ignore type annotations if they are not helpful. Of course, tools should make a best effort to infer types whenever they need to know a type that has not been explicitly specified, so as to make actors as reusable as possible across platforms. They should also try to check types whenever feasible, in order to locate errors and to detect malformed actor definitions.

subtype relation

substitutability

objects of type t

Each variable or parameter in CAL may be declared with a variable type. If it is, then this type remains the same for the variable or parameter in the entire scope of the corresponding declaration. Variable types may be related to each other by a *subtype relation*, $\prec$, which is a partial order on the set of all variable types. When for two variable types t, t' we have $t \prec t'$, then we say that t is a *subtype* of t', and t' is a *supertype* of t. Furthermore, a t[2] may be used anywhere a t' can be used—as a consequence, variables of subtypes are *substitutable* for those of supertypes.

It is important that each object has precisely one object type. As a consequence, object types induce an exhaustive partition on the objects, i.e. for any object type t we can uniquely determine the "objects of type t".

IMPLEMENTATION NOTE.
Stating that each object has an object type does *not* imply that this type can be determined at run time, i.e. that there is something like run-time type information associated with each object. In many cases, particularly when efficiency is critical, the type of an object is a compile-time construct whose main use is for establishing the notion of assignability, i.e. for checking whether the result of an expression may legally be stored in a variable. In these scenarios, type information is removed from the runtime representation of data objects.

---

[2] More precisely, "a t" is an object whose object type is assignable to t, see below.

For each implementation context we assume that there is a set T_V of variable types and T_O of object types. They are related to each other by an *assignability* relation $\leftarrow \subset T_V \times T_O$ which has the following interpretation: for any variable type t_V and object type t_O, $t_V \leftarrow t_O$ iff an object of type t_O is a legal value for a variable of type t_V.

assignability

The assignability relation may or may not be related to subtyping, but at a minimum it must be compatible with subtyping in the following sense. For any two variable types t_V and t'_V, and any object type t_O:

subtyping and assignability

$$t_V \prec t'_V \wedge t'_V \leftarrow t_O \Longrightarrow t_V \leftarrow t_O$$

In other words, if an object type is assignable to a variable type, it is also assignable to any of its supertypes.

4.2 Type formats

Even though the CAL language itself does not specify the meaning of most types, it provides notation for expressing types, so actor writers may put this information into their actor descriptions, and an implementation context may use it. There are three basic ways to express types in CAL, and two more constructs for expressing the types of procedural and functional closures (cf. sections 6.9.1 and 6.9.2).

```
Type → ID
     | ID '[' TypePars ']'
     | ID '(' [TypeAttr { ',' TypeAttr }] ')'
     | '[' [Types] '-->' Type ']'
     | '[' [Types] '-->' ']'
TypeAttr → ID ':' Type
     | ID '=' Expression
```

A type that is just an identifier either refers to a type parameter (if it occurs in the type parameters list of the actor), or it denotes the name of some other non-parametric type. Examples may be `String`, `Integer`.

The form `T[T1, ..., Tn]` is intended to stand for a *parametric type T* taking the types `Ti` as parameters. Such a parametric type is also called a *type constructor*. Many built-in types are of this kind, e.g. `List[Integer]` is a list if elements of type `Integer`, or `Map[String, Real]` is a finite map from keys of type `String` to values of type `Real`.

The next form can be thought of as a more general version of the previous one, where the type constructor has named parameters that may be bound to either types or values. For instance, the type `Matrix[element: Real, width = 4, height = 5]` might be the type of all matrices of real numbers of a certain size.

The type of a lambda closure is written as `[T1, ..., Tn --> T]`, where the `Ti` are the argument types, and `T` is the return type. Similarly, the type of a procedural closure is written as `[T1, ..., Tn -->]`, with the `Ti` again being the argument types.

4.3 Required types

Required types are the types of objects created as the result of special language constructions, usually expressions. The following is a partial list of the required types in CAL, cf. the appendix C for more details on basic data types, their properties, and the operations on them.

- `Null`—the type containing only the value **null**.
- `Boolean`—the truth values `true` and `false`.

pseudotype `ChannelID`

- `ChannelID`—the data type comprising the identifiers of channels. Most likely this will be a *synonym* for some other simple type, like the one of integer or natural numbers.
- `Collection[T]`—a finite collection of elements of type `T`.
- `Seq[T]`—a sequence (finite or infinite) of elements of type `T`.
- `List[T]`—finite lists of elements of type `T`. Lists are subtypes of the corresponding sequences and also subtypes or the corresponding collections, i.e.
  `List[T] < Seq[T]` and `List[T] < Collection[T]`
- `Set[T]`—finite sets of elements of type `T`. Sets are subtypes of the corresponding collections, i.e.
  `Set[T] < Collection[T]`
- `Map[K, V]`—maps from keys of type `K` to values of type `V`.

In addition to these, the types of functional and procedural closures are, of course, also built-in datatypes.

The choice of many fundamental types, such as those representing numeric entities, as well as the operations on them is left to the environment. The only built-in support for these types are the literals which are a result of the lexical scanning of the actor text. Interpretation as well as type assignment to these literals is left up to the environment. See section 6.1 for more details, including a design rationale.

4.4 Structured objects and their types

Many data objects contain other data objects—consider e.g. lists, which contain their elements. An object that contains other objects is called *composite* or *structured*, and so is its type.

composite/structured objects

CAL provides two mechanisms for identifying a subobject inside a structured object (in other words, specifying its *location* inside the composite): *fields* and *indices*. The mechanism used to identify locations inside a composite, as well as the fields and indices that are valid for specifying a location (see below), depend on the composite object and its type.

locations = fields $\cup$ indices

The location of a subobject can be used to either retrieve it from the composite (see sections 6.4 and 6.5) or, in the case of mutable objects and types, to replace the subobject with another object (see sections 4.5, 7.1.2, and 7.1.3).

Fields. Fields are names for subobjects inside a composite. Any given object type provides a finite number of those field names, and using any other name to identify a location inside the composite is an error. Since the names are statically provided as part of the program text (as opposed to being computed at runtime) it is possible to check statically whether a field name is permissible for a given object. If A denotes the composite object, and `f` is the field name, then

fields: compile-time locations

$$A.\mathtt{f}$$

A`.f`

selects the subobject in that field. Another consequence of the fact that fields are provided explicitly is that different fields may be associated with different subobject types, i.e. they may be *heterogeneous*.

heterogeneous field locations

Indices. Indexing into a composite object is a different mechanism for identifying a location inside it. Indices are a number of objects that can be computed at runtime, and which can be though of as the *coordinates* of the location inside of the composite. Which and how many indices are *valid* for a given object is determined by the object. If A denotes the composite object, and $E_1, \ldots, E_n$ are n valid indices, then

indices: runtime locations

$$A\,[E_1, \ldots, E_n]$$

selects the subobject at the specified location. Since indices are computed at runtime, all indexed locations inside a composite object must be assumed to be of the same static type, i.e. they are *homogeneous*.

homogeneous indexed locations

Example 1. A typical application of fields would be a type `Complex` with fields `real` and `imaginary`.

Lists in CAL are indexed by the natural numbers, starting at 0. For instance, the list `[11, 7, 19]` can be indexed with the indices 0, 1, and 2, so that e.g.

```
[11, 7, 19][1]
```

would yield the value 7. See section C.2.4 for details.

While natural numbers are very common indices, indexed composites are in general not restricted to numeric indices, but can allow any kind of object for the purpose. See section C.2.5 for details on how maps are indexed into.

indices of any type

Example 2. A `Map[K, V]` from a key type `K` to a value type `V` accepts as indices objects of type `K`, and indexing is simply the application of the map. Say the map

```
map {"abc"->15, "def"->7}
```

(of type `Map[String, Integer]`) can be indexed by a string as follows:

```
map {"abc"->15, "def"->7}["abc"]
```

resulting in the value `15`.

The following CAL types are required to support indexing:

- `Seq[T]`—index type is the natural numbers, starting from 0.
- `Map[K, V]`—index type is `K`.

Abstractly, indexing is the invocation of a special function, the *indexer* of a given datatype. The type of the resulting expression may depend on the number and types of the indices, and of course also on the indexed object.

4.5 Mutable objects and their types

Some structured types allow the *modification* or *mutation* of an object, i.e. changing the object at some location inside it. Such a type is called *mutable*, and so is an object of that type.

Mutating objects without any restrictions would be a technique that would render a program essentially unanalyzable in the general case. For this reason, CAL imposes a number of constraints on the ability to use this feature, which will be discussed in the context of mutable variables in sections 7.1.2 and 7.1.3.

orthogonal locations

free types

However, CAL makes no guarantee that the various locations of an object are independent, i.e. that all other locations remain unaffected by an assignment. A structured mutable type whose locations *are* independent is called *free*, its locations are called *orthogonal*—all predefined types in CAL are of this kind if they support mutation.

Example 3. A common non-orthogonal type might be `Complex`, the type of complex numbers. Let us assume its objects are mutable, and each has four locations, represented as fields: `real`, `imag`, `modulus`, and `argument`. If `c` is a mutable variable containing a complex number, then

```
c.real := 4;
c.imag := 3;
```

would at the same time result in the `modulus` component to be set to 5, and the `argument` to be set to arctan 4/3. Now the assignment

```
c.argument := 2;
```

can be interpreted as leaving the `modulus` unchanged, and modifying the `real` and `imag` fields accordingly, viz. to $5\cos 2$ and $5\sin 2$, respectively.

4.6 Type framework

Section 4.1 introduced two relations between types: the subtyping relation $\prec$ between variable types, and the assignability relation $\leftarrow$ between variable types and object types. It also required that the two relations be consistent with each other in the sense that assignability to a variable type would imply assignability to any of its supertypes.

This section introduces some simple properties that any type system provided by an implementation context must have. These properties, together with the basic minimal type system presented in section 4.3, constitute the CAL type framework.

Subtype ordering. The subtyping relation $\prec$ on the variable types is a partial order, i.e. for any $t_1, t_2, t_3 \in T_V$ the following holds:

1. Reflexivity: $t_1 \prec t_1$
2. Anti-symmetry: $t_1 \prec t_2 \wedge t_2 \prec t_1 \Longrightarrow t_1 = t_2$
3. Transitivity: $t_1 \prec t_2 \wedge t_2 \prec t_3 \Longrightarrow t_1 \prec t_3$.

Finite cover. For any variable type t, there is a unique finite, and possibly empty, set $Supt$ of variable types which has the following properties:

finite set of direct supertypes

1. $t \notin Supt$
2. $\forall t' \in Supt : t \prec t'$
3. $\forall t', t'' \in Supt : t' \not\prec t''$
4. $\forall t' : t \prec t' \Rightarrow \exists t'' \in Supt : t'' \prec t'$

The second condition ensures that the elements of $Supt$ are all superclasses of t. The third condition requires the set to be minimal. The third condition requires it to be exhaustive: all proper supertypes of t are supertypes of at least one element in $Supt$.

Least upper bound. For any two variable types $t_1, t_2 \in T_V$ there is a unique finite, and possibly empty, set of variable types $t_1 \sqcup t_2$ which has the following properties:

finite set of supertypes for any pair of types

1. $\forall t \in t_1 \sqcup t_2 : t_1 \prec t \wedge t_2 \prec t$
2. $\forall t', t'' \in t_1 \sqcup t_2 : t' \not\prec t''$
3. $\forall t : (t_1 \prec t \wedge t_2 \prec t) \Rightarrow \exists t' \in t_1 \sqcup t_2 : t' \prec t$

Again, the first condition ensures that all types in the set are indeed upper bounds of the two types in question. The second condition requires them to be minimal. The third condition requires the set to be exhaustive: all supertypes of t_1 and t_2 that are not in that set are supertypes of a type that is.

Chapter 5

Variables

bindings

Variables are placeholders for other values. They are said to be *bound* to the value that they stand for. The association between a variable and its value is called a *binding*.

assignable & mutable bindings

CAL distinguishes between different kinds of bindings, depending on whether they can be assigned to (*assignable variable binding*), and whether the object they refer to may be mutated (*mutable variable binding*—cf. sections 4.5 and 7.1.3).

This chapter first explains how variables are *declared* inside CAL source code. It then proceeds to discuss the scoping rules of the language, which govern the visibility of variables and also constrain the kinds of declarations that are legal in CAL.

5.1 Variable declarations

declarations

Each variable (with the exception of those predefined by the platform) needs to be explicitly introduced before it can be used—it needs to be *declared* or *imported* (section 3.1). A declaration determines the kind of binding associated with the variable it declares, and potentially also its (variable) type. There are the following kinds of variable declarations:

- explicit variable declarations (section 5.1.1,
- actor parameters (chapter 3),
- input patterns (section 8.1),
- parameters of a procedural or functional closure (section 6.9).

Variables declared as actor parameters, in input patterns, or as parameters of a procedural or functional closure are neither assignable nor mutable.

The properties of a variable introduced by an explicit variable declaration depend on the form of that declaration.

5.1.1 Explicit variable declarations

Syntactically, an explicit variable declaration[1] looks as follows:

VarDecl → **[mutable]** [Type] ID [('=' | ':=') Expression]
        | FunDecl | ProcDecl

We will discuss function and procedure declarations (*FunDecl* and *ProcDecl*) in section 6.9.3.

An explicit variable declaration can take one of the following forms, where `T` is a type, `v` an identifier that is the variable name, and `E` an expression of type `T`:

- `T v`—declares an assignable, non-mutable variable of type `T` with the default value for that type as its initial value. It is an error for the type not to have a default value.
- `T v := E`—declares an assignable, non-mutable variable of type `T` with the value of `E` as its initial value.
- `mutable T v := E`—declares an assignable and mutable variable of type `T` with the value of `E` as its initial value.
- `mutable T v = E`—declares an non-assignable and mutable variable of type `T` with the value of `E` as its initial value.
- `T v = E`—declares a non-assignable, non-mutable variable of type `T` with the value of `E` as its initial value.

stateful vs stateless

Variables declared in any of the first four ways are called *stateful variables,* because they or the object they are containing may be changed by the execution of a statement. Variables declared in the last way are referred to as *stateless variables.*

Explicit variable declarations may occur in the following places:

- actor state variables
- the var block of a surrounding lexical context
- variables introduced by a let-block

While actor state variables and variables introduced in a var-block can be state variables as well as non-state variables, a let-block may only introduce non-state variables.

---

[1] These declarations are called "explicit" to distinguish them from more "implicit" variable declarations that occur, e.g., in generators or input patterns.

5.2 Variable scoping

The scope of a variable is the lexical construct that introduces it—all expressions and assignments using its name inside this construct will refer to that variable binding, unless they occur inside some other construct that introduces a variable of the same name, in which case the inner variable shadows the outer one.

lexical scoping

In particular, this includes the *initialization expressions* that are used to compute the initial values of the variables themselves. Consider e.g. the following group of variable declarations inside the same construct, i.e. with the same scope:

initialization expression

```
n = 1 + k,
k = 6,
m = k * n
```

This set of declarations (of, in this case, non-mutable, non-assignable variables, although this does not have a bearing on the rules for initialization expression dependency) would lead to `k` being set to 6, `n` to 7, and `m` to 42. Initialization expressions may not depend on each other in a circular manner—e.g., the following list of variable declarations would not be well-formed:

well-formed declaration

```
n = 1 + k,
k = m - 36,
m = k * n
```

More precisely, a variable may not be in its own *dependency set*. Intuitively, this set contains all variables that need to be known in order to compute the initialization expression. These are usually the *free* variables of the expression itself, plus any free variables used to compute them and so on—e.g., in the last example, `k` depended on `m`, because `m` is free in `m - 36`, and since `m` in turn depends on `k` and `n`, and `n` on `k`, the dependency set of `k` is {`m`, `k`, `n`}, which *does* contain `k` itself and is therefore an error.

dependency set

This would suggest defining the dependency set as the transitive closure of the free variable dependency relation—which would be a much too strong criterion. Consider e.g. the following declaration:

```
f = lambda (n) :
  if n = 0 then 1 else n * f(n - 1) end
end
```

Here, `f` occurs free in the initialization expression of `f`, which is clearly a circular dependency. Nevertheless, the above definition simply describes a recursive function, and should thus be admissible.

recursion

The reason why `f` may occur free in its own definition without causing a problem is that it occurs inside a closure—the *value* of `f` need not be known in order to construct the closure, as long as it becomes known before we *use* it—i.e. before we actually apply the closure to some argument.

We will now define the dependency sets I_v and D_v of a variable v among a set of variables V that are defined simultaneously in the same scope.

Definition 1 (I_v, D_v—the dependency sets of a variable v). Consider a set V of variables v which are defined simultaneously, i.e. the intial value of each of these variables defined by an expression E_v which is in the scope of all the variables in V. Let F_v be the set of free variables of E_v. As we are only interested in the free variables in V, we will usually use the intersection $F_v \cap V$.

The *dependency set* D_v is defined as the smallest set such that the following holds:

$$\begin{array}{ll} \text{(a)} & F_v \cap V \subseteq D_v \\ \text{(b)} & \bigcup_{x \in D_v} D_x \subseteq D_v \end{array}$$

immediate dependency set

The *immediate dependency set* I_v of each variable v is defined as follows

$$I_v = \begin{cases} \emptyset & \text{for } E_v \text{ a closure} \\ D_v & \text{otherwise} \end{cases}$$

Intuitively, D_v contains those variables in V on which the object bound to v directly or indirectly depends. I_v is the set of variables whose values need to be known when the object computed by E_v is created—for most expressions, it is the same as D_v, but for closures (procedural or functional) this set is empty, because there is no need to evaluate the body of the closure in order to construct the closure object.[2]

Now we capture the notion of well-formedness of a set of simultaneously defined variables V as a condition on the dependency sets as follows:

well-formed declaration set

Definition 2 (Well-formed declaration set). A set of simultaneously declared variables (a *declaration set*) V is *well-formed* iff for all $v \in V$

$$v \notin I_v$$

Note that, as in the example above, a variable may occur free in its own initialization expression, but still not be in its own immediate dependency set, as this only includes those variables whose *value* must be known in order to compute the value of the declared variable.

This notion of well-formedness is useful because of the following property:

Corollary 1 (No mutual dependencies in well-formed variable sets.). Given a well-formed variable set V, for any two variables $v_1, v_2 \in V$, we have the following property:

$$\neg(v_1 \in I_{v_2} \wedge v_2 \in I_{v_1})$$

That is, no two variables ever mutually immediately depend on each other.

---

[2] Here we use the fact that closures can be constructed without the values of their free variables, which is clearly an artifact of the way we envision closures to be realized, but it is a useful one.

The proof of this property is trivial, by contradiction and induction over the definition of the dependency set (showing that mutual dependency would entail self-dependency, and thus contradict well-formedness).

RATIONALE.
Strictly speaking, this definition of well-formedness is a conservative approximation to what might be considered well-formed variable declarations. It is chosen so that (a) it allows commonly occurring circular dependencies to be expressed and (b) it can be implemented without undue efforts. But it is important to realize that it does declare some declaration sets as erroneous which could easily be interpreted. For instance, consider the following case:

```
f = lambda (x) :
  if x <= 1 then 1 else x * g(x - 1) end
end,
g = f
```

According to the definitions above, both `g` ends up in its own immediate dependency set, and thus these declarations are not well-formed.

This allows us to construct the following relation over a set of variables:

dependency order relation

Definition 3 (Dependency relation). Given a set of variables V defined in the same scope, we define a relation $\prec$ on V as follows:

$$v_1 \prec v_2 \Longleftrightarrow v_1 \in I_{v_2}$$

In other words, a variable is 'smaller' than another according to this relation iff it occurs in its dependency set, i.e. iff it has to be defined before the other can be defined. The well-formedness of the declaration set implies that this relation is a non-reflexive partial order, since variables may not mutually depend on each other.

This order allows us to execute variable declarations in such a way that immediate dependencies are always evaluated before the dependent variable is initialized.

Example 4. Consider the following variable definitions:

```
a = f(x),
f = lambda (v) :
    if p(v) then v else g(h(v)) end
end,
g = lambda (w) : b * f(w) end,
b = k
```

Note that f and g are mutually recursive.

The following lists the immediate dependencies and the free variable dependencies of each variable above,[3] along with their intersection with the set $\{a, f, g, b\}$, which is the set V in this case:

v	F_v	$F_v \cap V$	D_v	I_v
a	$\{f,x\}$	$\{f\}$	$\{f,g,b\}$	$\{f,g,b\}$
f	$\{p,g,h\}$	$\{g\}$	$\{f,g,b\}$	$\emptyset$
g	$\{b,f\}$	$\{b,f\}$	$\{f,g,b\}$	$\emptyset$
b	$\{k\}$	$\emptyset$	$\emptyset$	$\emptyset$

Now let us compute the dependency set D_a of the variable a. We start with the set

$$F_a \cap V = \{f\}$$

Now we compute

$$(F_a \cup F_f) = \{f, g\}$$

Then

$$(F_a \cup F_f \cup F_g) \cap V = \{f, g, b\}$$

Finally, we reach a fixpoint at

$$D_a = (F_a \cup F_f \cup F_g \cup F_b) \cap V = \{f, g, b\}$$

Similarly, we compute D_f, D_g, and D_b. The immediate dependency sets of f and g are empty, because their initialization expressions are closures. The immediate dependency sets of a and b are the same as their dependency sets, which in the case of b is also empty, because it does not depend on any variable in V. As a result of this analysis, we see that the variables f, g, and b may be defined in any order, but all must be defined before a, as it depends on all of them.

Example 5. Now consider the following slightly changed variable definitions, with an additional dependency added to b:

```
a = f(x),
f = lambda (v) :
    if p(v) then v else g(h(v)) end
end,
```

---

[3] We are disregarding here the implicit variable references that will be introduced when the operators are resolved to function calls—strictly speaking, they would become part of F_v, but as they are always referring to global variables, and would thus disappear from $F_v \cap V$ anyway, we do not bother with them in the example.

```
g = lambda (w) :
    b * f(w)
end,
b = a .* k
```

Again, the following table lists the dependency sets:

v	F_v	$F_v \cap V$	D_v	I_v
a	$\{f, x\}$	$\{f\}$	$\{f, g, b, a\}$	$\{f, g, b, a\}$
f	$\{p, g, h\}$	$\{g\}$	$\{f, g, b, a\}$	$\emptyset$
g	$\{b, f\}$	$\{b, f\}$	$\{f, g, b, a\}$	$\emptyset$
b	$\{a, k\}$	$\{a\}$	$\{f, g, b, a\}$	$\{f, g, b, a\}$

Now, computing D_a proceeds as follows:

$$F_a \cap V = \{f\}$$

$$(F_a \cup F_f) \cap V = \{f, g\}$$

$$(F_a \cup F_f \cup F_g) \cap V = \{f, g, b\}$$

$$(F_a \cup F_f \cup F_g \cup F_b) \cap V = \{f, g, b, a\}$$

$$D_a = (F_a \cup F_f \cup F_g \cup F_b \cup F_a) \cap V = \{f, g, b, a\}$$

Obviously, in this case $a \in I_a$, thus the set of variable definitions is not well-formed.

Chapter 6

Expressions

no side effects

Expressions evaluate to a value and are side-effect-free, i.e. they do not change the state of the actor or assign or modify any other variable. Thus, the meaning of an expression can be described by the value it is evaluating to.

expressions and types

If the computation of an expression terminates, it results in a value, and that value has an object type. The precise value depends on the environment surrounding the expression when it is evaluated. In general, the objects computed by an expression in different environments may belong to different object types. However, Using the properties of the type system outlined in section 4.6, we can compute a set of least upper variable type bounds for each expression, knowing the declared types of each of its free identifiers and the types of the literals occurring in it. The object types of all objects ever computed by the expression must be assignable to *all* of the variable types in that set.

expression syntax

The following is an overview of the kinds of expressions and expression syntaxes provided in CAL.

Expression → PrimaryExpression { Operator PrimaryExpression }

PrimaryExpression → [Operator] SingleExpression
    { '(' [Expressions] ')' | '[' Expressions ']' | '.' ID }

SingleExpression → [**old**] ID
    | ExpressionLiteral
    | '(' Expressions ')'
    | IfExpression
    | LambdaExpression
    | ProcExpression
    | LetExpression
    | ListComprehension | SetComprehension | MapComprehension
    | TypeAssertionExpr

We will now discuss the individual kinds of expressions in more detail.

6.1 Literals

literals describe constants

Expression literals are constants of various types in the language. They look as follows:

ExpressionLiteral → IntegerLiteral | DecimalFractionLiteral
| StringLiteral
| **true** | **false** | **null**

The type of **true** and **false** is `Boolean`, the type of **null** is `Null`.

literal types determined by context

The exact types of the other literals are determined by the environment, but the intuition behind them is that they represent the integers, the real numbers, and character strings, respectively.

> RATIONALE.
> We are not committing to particular types for the numeric literals because we want to keep the requirements on the type system as weak as possible, to allow for a wide range of implementations. For example, some implementations may have only integer numbers up to a specific word size (say, 32 or 64 bits), while others have variable-sized integers. Such an implementation may want to assign a specific type to an integer literal depending on its numeric size—e.g., it may make it an object of type `Integer32` if it fits into 32 bits, and of type `Integer` if it is larger than that.

6.2 Variable references

The expression used to refer to the value bound to a variable at any given point during the execution is simply the name of the variable itself, i.e. an identifier.

6.2.1 Old variable references

assignable & mutable bindings

The code inside an action may refer to the value of a variable at the beginning of the action by prefixing the variable name with the keyword **old**, as in the following example:

Example 6. . . .

```
sum := 0;

action [a] ==> [old sum / sum]
do
  sum := sum + a;
end

...
```

The output expression refers to both the value of sum at the beginning of the firing as well as its value at the end of the firing. This code is equivalent to the following:

old-references: state *before* a firing

```
...

sum := 0;

action [a] ==> [oldSum / sum]
var
  oldSum = sum
do
  sum := sum + a;
end

...
```

In other words, using **old** values of variables in an action introduces an implicit non-assignable, non-mutable variable (cf. section 7.1.4 for the implications of the original variable being mutable).

implicit variable

Closures created inside an action may also refer to **old** variables, and the meaning of this follows from the transformation above: They will always refer to the value of the variable at the beginning of the firing that created them.

old-references and closures

The **old** keyword may not be used outside of an action, or in front of a variable that is not an assignable or mutable actor state variable.

6.3 Function application

Function applications have the following syntactic format:

FunctionApplication → Expression '(' Expressions ')'

The first expression must evaluate to a function of the appropriate type and arity. The expression inside the parentheses are the function *arguments*. The value of the function application expression is the value computed by the function when applied to the values of the argument expressions.

language implementation: the **art of cheating** without getting caught

IMPLEMENTATION NOTE.
Functions may, in principle, be non-strict, i.e. evaluation of argument expessions may be deferred until their values are actually needed. The bounds for implementations are defined only by an expression's proper behavior with respect to the modification of state. For example, consider the following statements:

```
g := lambda (x) : lambda (y) : x + y end end;
a := 11;
f := g (a * a);
a := 7;
b := f(1);
```

Any correct implementation must ensure that `b` has the value 122 at the end of this statement sequence. If it naively defers the evaluation of the argument expression `a * a` until `f` is *called*, `b` will erroneously be assigned the value 50. This, of course, does not mean that the argument expression must be evaluated when `g` is called, but it must behave *as if* it were evaluated in the state valid at the time of the call whenever its value is needed. In this example, it could move the old value of a into the lambda-closure and use it, rather than the current value of a, when it finally evaluates the argument expression.

In other words, if an implementation chooses to employ some form of lazy evaluation, the effect is only to give meaning to programs that would otherwise not be valid, while not affecting the meaning of programs that do evaluate under an eager implementation. Consider this example:

```
let
  f = lambda (x, y) : x end,
  g = lambda () : g() end
:
  f(1, g())
end
```

In lazy implementations, this expression will yield 1, while in eager implementations it will be an error.

6.4 Field selection

A field selector expression extracts a sobobject from a composite object (see 4.4 for details on composite objects). The syntax is as follows:

FieldSelectorExpr → Expression '.' ID

The result of this expression is the subobject of the value of the expression that is contained in the field specified by the identifier.

6.5 Indexing

indices define *location*

An indexing expression selects a subobject from a composite object (cf. section 4.4 for more details). Syntactically, indexing expressions are similar to function applications, although they use square brackets for enclosing the arguments. The general format is

IndexerExpr → Expression '[' Expressions ']'

where the first expression must be of a type that supports an indexer, and the expressions between the brackets must be indices specifying a valid location for the given object. The type of an indexing expression is determined by the indexer, which is different for each structured data type, and may differ according to the number of indices and their types.

6.6 Operators

unary & binary operators

There are two kinds of operators in CAL: unary prefix operators and binary infix operators. A binary operator is characterized by its associativity and its precedence. In CAL, all binary operators associate to the left, while their precedence is defined by the platform. Unary operators always take precedence over binary operators.

Example 7.
`a + b + c` is always `(a + b) + c`.
`#a + b` is always `(#a) + b`.
`a + b * c` is `a + (b * c)` if `*` has a higher precedence than `+`, which is usually the case (see also the appendix C.1).

syntactical sugar

Operators are just syntactical elements—they represent ordinary unary or binary functions, so the only special rules for operators are syntactical. In general, the set of operators is defined by the implementation context, although a small number of operators are predefined. These operators are represented by keywords, as opposed to strings of symbols, which represent all the other operators (cf. section 2.1).

6.7 Conditional expressions

The simple conditional expression has the following form:

IfExpression → **if** Expression **then** Expression **else** Expression **end**

The first subexpression must be of type `Boolean`, and the value of the entire expression is the value of the second subterm if the first evaluated to `true`, and the value of the third subterm otherwise.

The type of the conditional expression is the most specific supertype (least upper bound) of both, the second and the third subexpression. It is undefined (i.e. an error) if this does not exist.

6.8 Introducing a local scope

In expressions, local variables are introduced using a let-construct. This is often useful to factor out large subexpressions that occur several times.

LetExpression → **let** VarDecls ':' Expression (**end**|**endlet**)

The list of local definitions defines new identifiers and binds them to values. The variables (which are non-mutable and non-assignable) are visible inside the body expression. Its type is the type of the entire construct.

6.9 Closures

closure = code + environment

Closures are objects that encapsulate some program code along with the variable context (its *environment*) that was valid when it was created. CAL distinguishes two kinds of closures, which differ in the *kind* of code they encapsulate:

functions & procedures

- *function closures* (or simply *functions*) contain a parametric expression,
- *procedure closures* (or just *procedures*) contain a parametric list of statements.

The two kinds of closures are used in different contexts, and in different ways—the *application* of a functional closure to (a tuple of) arguments is an expression (cf. section 6.3), whereas the *call* of a procedural closure on (a tuple of) arguments is a statement (cf. section 7.2).

6.9.1 Lambda-expressions and function closures

Function closures are the result of evaluating a **lambda**-expression. They represent functions that are defined by some expression which is parameterized and may also refer to variables defined in the surrounding context.

function = expression + environment

LambdaExpression → [**const**] **lambda** '(' [FormalPars] ')' ['-->' Type]
    [**var** VarDecls] ':' Expression (**end**|**endlambda**)

FormalPar → [Type] ID

Applying function closures is side-effect free, i.e. their application (to arguments) does not change the state. However, in general they may *refer* to stateful variables, and thus may themselves depend on the assignment of variables in their context, and thus be affected by side effects of other constructs.

function application side-effect-free

The **const** keyword identifies those function closures for which this is not the case, i.e. which do not refer to variables whose values may change—such a function is also called an *invariant function (closure)*. It does not change the behavior of the closure, i.e. removing it will not affect the value computed by the closure. It is intended to serve as a declaration that expresses the programmers intention, and that may be checked by a compiler. It is an error for a **const lambda**-closure to refer to assignable or mutable variables.

invariant function closures

If the types of the formal parameters are `T1` to `Tn`, respectively, and the return type is `T`, then the type of the lambda expression is

```
[T1, ..., Tn --> T]
```

The type of an invariant function closure is a subtype of the corresponding function closure type, and is written as

```
const [T1, ..., Tn --> T]
```

The only built-in operation defined on a function closure is its *application* to a tuple of arguments, cf. section 6.3.

applying functions ⟶ 6.3

6.9.2 Proc-expressions and procedure closures

Procedure closures are somewhat similar to function closures, in that they encapsulate a piece of code together with the context in which it was defined. However, in the case of procedure closures, this piece of code is a list of statements, i.e. executing a procedure closure is likely to have side effects (as opposed to the application of a function closure).

procedure = statements + environment

Syntactically, a procedure closure looks as follows:

ProcExpression → **proc** '(' [FormalPars] ')' [**var** VarDecls]
(**do**| **begin**) { Statement } (**end**|**endproc**)

If the types of the formal parameters are `T1` to `Tn`, respectively, then the type of the proc expression is

```
[T1, ..., Tn -->]
```

calling procedures ⟶ 7.2

Since block closures can produce side effects, their execution cannot be part of the evaluation of an expression. Executing a block closure is a fundamental kind of statement, which is discussed in section 7.2.

6.9.3 Function and procedure declarations

One very common use for closures is the definition of functions or procedures with a particular fixed name inside some scope, often the actor itself. This can be done using the standard variable declaration syntax, as follows:

```
timestwo = lambda(x) : 2 * x end
```

However, since this use is so frequent, CAL provides special syntax that looks a little more familiar and makes the definition of functions and procedures a little easier. The above could also be written like this:

```
function timestwo (x) : 2 * x end
```

The general format for these constructs is as follows:

FuncDecl → **function** ID '(' [FormalPars] ')' [**var** VarDecls ':']
Expression **end**

ProcDecl → **procedure** ID '(' [FormalPars] ')' [**var** VarDecls (**begin**| **do**)]
{ Statement } **end**

The variable introduced by these declarations is non-assignable and non-mutable.

6.10 Comprehensions

constructing composite objects

Comprehensions are expressions which construct one of the built-in composite objects: sets, lists, or maps. There are two variants of comprehensions, those with and those without *generators*. We will first focus on the simpler ones without generators, and then turn to the more general comprehensions with generators. The reason for this order of presentation is that the meaning of comprehensions with generators will be defined by reducing them to simple collection expressions.

Note. The `Collection` type is a supertype of both `Set` and `List`, but *not* `Map`. In spite of this, we will use the term *collection* for maps as well in this section, because the way they are constructed is very much the same as for sets and lists. To avoid confusion, we will refer to sets and lists as *proper collections* or `Collections` if we want to distinguish them from maps.

6.10.1 Simple collection expressions

simple comprehension: enumerate elements

Simple collection expressions just enumerate the elements of the set or list and the mappings of the map, respectively. They are written as follows:

SimpleSetComprehension → '{' [Expressions] '}'
SimpleListComprehension → '[' [Expressions ['|' Expression]] ']'
SimpleMapComprehension → **map** '{' [Mappings] '}'

unordered set vs ordered list

The elements of a set are not ordered, and each element occurs inside the set at most once. If two or more element expressions evaluate to the same value, they result in only one element inside the set.[1] In contrast, the elements in a list are ordered, and the same element may occur in it more than once. Also, the list syntax allows a *tail* to be specified, which must be a list of the same type that is appended to the list of the explicitly specified elements.

Example 8. If `n` is the number `10`, then the simple set expression
`{n, n*n, n-5, n/2}`
evaluates to the set `{10, 100, 5}`.
If `s` represents the list `[1, 2, 3]`, then the simple list expression
`[4, 5, 6 | s]`
is the list `[4, 5, 6, 1, 2, 3]`.

nondeterminism in map construction

Simple map expressions explicitly specify the mappings from *keys* to *values*. Similar to the case of sets, if two key expressions result in the same value, only one key/value mapping will be generated from them. If the corresponding value expressions are not the same, one of the values will be chosen.

Example 9. Let again `n` be `10`. The map
`map {n -> 1, n*n -> 2, n-5 -> 3, n/2 -> 4}`
evaluates to either `map {10 -> 1, 100 -> 2, 5 -> 3}`
or to `map {10 -> 1, 100 -> 2, 5 -> 4}`.

---

[1] For this reason, it is in general only possible to infer an upper bound on the number of elements from a simple set expression, not the precise number of elements.

6.10.2 Comprehensions with generators

constructing large collections

Simple comprehension expressions only allow the construction of sets, lists, or maps of a size that is directly correlated with the size of the expression. In order to facilitate the construction of large or variable-sized collections, CAL provides *generators* to be used inside an expression constructing a collection. The syntax of the resulting general form of comprehensions looks as follows:

SetComprehension → '{' [Expressions [':' Generators]] '}'
ListComprehension → '[' [Expressions [':' Generators] ['|' Expression]] ']'
MapComprehension → **map** '{' [Mappings [':' Generators]] '}'
Mapping → Expression '->' Expression
Generator → **for** [Type] ID [',' IDs] **in** Expression { ',' Expression }

generators & filters

The generators, which begin with the `for` keyword, introduce new variables, and successively instantiate them with the elements of the proper collection after the `in` keyword. The expression computing that collection may refer to the generator variables defined to the left of the generator it belongs to. If that expression is of type *Collection*[*T*], the corresponding generator variable is of type *T* unless declared otherwise.

The optional expressions following the collection expression in a generator are called *filters*—they must be of type `Boolean`, and only variable bindings for which these expressions evaluate to **true** are used to construct the collection.

Example 10. The expression {} denotes the empty set, while

```
{1, 2, 3}
```

is the set of the first three natural numbers. The set

```
{2 * a :  for a in {1, 2, 3}}
```

contains the values 2, 4, and 6, while the set

```
{a :  for a in {1, 2, 3}, a > 1}
```

describes (somewhat redundantly) the set containing 2 and 3. Finally, the set

```
{a * b :  for a in {1, 2, 3},
      for b in {4, 5, 6}, b > 2 * a}
```

contains the elements 4, 5, 6, 10, and 12.

generator variable scoping

Writing the above as

```
{a * b :  for a in {1, 2, 3},
      b > 2 * a, for b in {4, 5, 6}}
```

is **illegal** (unless b is a defined variable in the context of this expression, in which case it is merely very confusing!), because the filter expression

```
b > 2 * a
```

occurs before the generator that introduces `b`.

If the generator collection is a set rather than a list, the order in which elements are extracted from it will be unspecified. This may affect the result in case of a list comprehension.

element order in generators

Example 11. Because lists are order-sensitive, the list

```
[ a :  for a in [1, 2, 3] ]
```

is different from the list

```
[ a :  for a in [3, 2, 1] ]
```

If the collection computed in a generator is *not* itself a list but a set, as in

```
[ a :  for a in {1, 2, 3} ]
```

then the order of the elements in the resulting list will be indeterminate.

Generator semantics. In order to precisely describe the evaluation of a comprehension that contains generators and filters, we need to introduce a few symbols first. We call the original expression E. Removing all generators from E results in the simple collection expression E'. Now E has $N \geq 0$ generators. The i-th generator has $M_i \geq 1$ variables, with names $v_{i,1}, ..., v_{i,M_i}$. The collection expression of the i-th generator will be called C_i. Following the i-th generator are $K_i \geq 0$ filter expressions, which we call $F_{i,1}, ..., F_{i,K_i}$. Let Z denote the corresponding empty collection expression, i.e. {} for sets, `[]` for lists, and `map {}` for maps. The tail expression of lists will be treated separately: we replace $[S|T]$ with $[S] + T$ first, and then apply the following algorithm to the *tail-free* list comprehension $[S]$.

for $v_{i,1}, ..., v_{i,M_i}$ in C_i, $F_{i,1}, ..., F_{i,K_i}$

In the following, we define the meaning of generators in comprehensions by replacing them with a number of previously described constructs, such as function closures, function application, and conditional expressions. The key to this replacement is a function `$mapadd`, which we will describe below.

syntactical transformation of generators

$$E \equiv G(1)$$

$$G(i) \equiv \begin{cases} E' & i > N \\ \underline{\text{let}}\ a_i \underline{\equiv} C_i \underline{:} G_V(i, 1)\ \underline{\text{end}} & \text{otherwise} \end{cases}$$

$$G_V(i, j) \equiv \begin{cases} G_F(i, 1) & j > M_i \\ \underline{\text{\$mapadd}(}a_i\underline{,\ \text{lambda}\ (}v_{i,j}\underline{)\ :}\ G_V(i, j+1)\ \underline{\text{end})} & \text{otherwise} \end{cases}$$

$$G_F(i, k) \equiv \begin{cases} G(i+1) & k > K_i \\ \underline{\text{if}}\ F_{i,k}\ \underline{\text{then}}\ G_F(i, k+1)\ \underline{\text{else}}\ Z\ \underline{\text{end}} & \text{otherwise} \end{cases}$$

The a_i are distinct and fresh variable symbols, i.e. they are mutually different, and they also differ from any variable symbol occurring in the comprehension.

The `$mapadd` function takes two arguments, a collection and a unary function. It iterates over the elements of its first argument, and applies the function

`$mapadd`:
`Collection[T]`
`[T-->Set/List/Map[T]]`

to each element. The resulting values are *added*—for sets, this means it produces the union of all results, for lists the concatenation, and for maps the map-union.

RATIONALE.
Of course, whether there exists a function `$mapadd` at all, and whether it has this name will be implementation-dependent. The reason for choosing a name with a $-sign is because users are discouraged from using the $-sign for their identifiers, so that generated identifiers and internal names can be formed without creating conflicts with names chosen by the user.

Example 12. The expression

```
$mapadd([1, 2, 3], lambda(x) :  {x, x + 2} end)
```

results in `{1, 3} + {2, 4} + {3, 5}`, which is

```
{1, 2, 3, 4, 5}
```

The expression

```
$mapadd([1, 2, 3], lambda(x) :  [x, x + 2] end)
```

results in `[1, 3] + [2, 4] + [3, 5]`, which is

```
[1, 3, 2, 4, 3, 5]
```

The expression

```
$mapadd([1, 2, 3],
     lambda(x):  map{x->x*x, x+2->x} end)
```

results in `map{1->1, 3->1} + map{2->4, 4->2} + {3->9, 5->3}`, which is either

```
map{1->1, 2->4, 3->1, 4->2, 5->3}
```

or

```
map{1->1, 2->4, 3->9, 4->2, 5->3}
```

efficiency of
comprehension
implementations

IMPLEMENTATION NOTE.
The fact that we define the meaning of generators inside comprehensions by replacing them with other constructs is not to suggest that this is a good implementation strategy. Even though it simplifies the language implementation, it is most likely very inefficient, introducing a lot of overhead in the form of closure creation, function application, and many intermediate collection objects that get added up to compute the final result.

6.11 Type assertion

A type assertion expression is a way to attach type information to an arbitrary expression inside the program code. Other than type assertions, the only expressions that the user explicitly provides type information for are variables, viz. when they are declared. The types of all other expressions are inferred from those. Depending on the type system, this inference may be more or less precise, and it may or may not be sufficient to guarantee a safe execution of the program code. In such cases, it might be useful to explicitly add a type assertion into the code, either to detect that it will not hold at compile time, or to be able to check it at runtime. The syntax looks as follows:

attaching type information to any expression

TypeAssertionExpr → '(' Expression '::' Type ')'

At compile time, if types are checked and inferred, there are three possible consequences of this construction depending on the type that the inference yielded for the embedded expression.

compile-time type checking

1. The type can be shown to be a subtype of the asserted type. In this case, the assertion is always true.
2. The type can be shown to not intersect with the asserted type. In this case, objects computed by the expression will always fail to be of the asserted type, and thus this expression is will always result in an error, which can be reported at compile time.
3. If neither can be shown, the expression may or may not produce objects of the asserted type. This results in three subcases:
    (a) In a conservative system, an error will be signaled at compile time.
    (b) The translator inserts a check that tests for compliance with the asserted type and causes an error if an object computed by the expression fails this check.
    (c) The translator does nothing based on the assumption that the assertion implies that the expression will always produce the proper values.

runtime type checks

Chapter 7

Statements

The execution of an action (as well as actor initialization) happens as the execution of a (possibly empty) sequence of *statements*. The only observable effect of a statement is a change of the variable assignments in its environment.[1] Consequently, the meaning of a statement is defined by how the variables in its scope change due to its execution. CAL provides the following kinds of statements:

statements have"side-effects"

Statement → AssignmentStmt
 | CallStmt
 | BlockStmt
 | IfStmt
 | WhileStmt
 | ForeachStmt
 | ChooseStmt

7.1 Assignment

Assigning a new value to a variable is the fundamental form of changing the state of an actor. The syntax is as follows:

AssignmentStmt → ID [Index | FieldRef] ':=' Expression ';'

Index → '[' [Expressions] ']'

FieldRef → '.' ID

---

[1] Other effects of statements, and indeed expressions, may include input/output operations, but we will disregard them in this context as they are not directly observable from within the language.

An assignment without an index or a field reference is a *simple assignment*, while one with a field reference is a *field assignment*, and one with an index is called an *indexed assignment*. Field assignments and indexed assignments are also collectively referred to as *mutations* (cf. sections 4.4 and 4.5 for more information on structured objects and mutability).

7.1.1 Simple assignment

In a simple assignment, the left-hand side is a variable name. A variable by that name must be visible in this scope, and it must be assignable.

assignability ⟶ 4.1

The expression on the right-hand side must evaluate to an object of a value compatible with the variable (i.e. its type must be assignable to the declared type of the variable, if any—see section 4.1). The effect of the assignment is of course that the variable value is changed to the value of the expression. The original value is thereby overwritten.

7.1.2 Field assignment

assigning to field locations

If a variable is of a type that has fields (see section 4.4)), and if it is mutable (see section 4.5), assignments may also selectively assign to one of the fields rather than only to the variable itself. The syntax is as follows:

FieldAssignmentStmt → ID '.' ID ':=' Expression ';'

Here, the first identifier is the variable name, while the second is the *field name*, which must be valid for the variable and the object that the variable refers to.

7.1.3 Assignment with indices

assigning to indexed locations

If a variable is of a type that is indexed, and if it is mutable, assignments may also selectively assign to one of its indexed locations, rather than only to the variable itself. The syntax is as follows:

IndexedAssignmentStmt → ID '[' Expressions ']' ':=' Expression ';'

indices

In CAL, an indexed location inside an object is specified by a sequence of objects called *indices*, which are written after the identifier representing the variable, and which enclosed in square brackets.

7.1.4 Assigning to and from mutable variables

In order to be able to reason about the actor state, and to facilitate program transformations, CAL is designed to avoid *aliasing* of stateful structures. In other words, if a structure can be mutated, no two variables may point to it at the same time.

no aliasing

Therefore, when assigning a data structure to a mutable variable, that data structure must be cloned.[2] Of course, this cloning operation can occur on demand, or lazy, whenever the data structure, or a part of it, is mutated. Whichever implementation is chosen, mutations via a mutable variable must never have an effect on the value of other variables.

cloning

lazy cloning

Similarly, when assigning *from* a mutable variable, the structure assigned is conceptually copied, so that subsequent mutations of it are not visible via the new variable.

---

[2] In fact, it must be deeply cloned, up to the point where mutations can occur.

IMPLEMENTATION NOTE.
This may pose difficult implementation issues. Consider the following example:

```
mutable List[Integer] a = ...;
List[Integer] b := f(a[7, 1111]);
```

Let us assume that the indexer with two arguments on `Lists` computes the sublist from the first index to the second, inclusive, i.e. `a[7, 1111]` computes a list of length 1105 elements. A naive implementation would do just that, i.e. actually create the sublist. However, if the sublist is only an intermediate value in the computation of `f`, this would be very wasteful, e.g. in this case:

```
function f(List[Integer] v)-->List[Integer] :
    [#v]
end
```

Here, `f` returns a list of length 1 whose only element is the length of its parameter list.
Alternatively, a sublist could be represented by a special sublist-object that is backed by the original list, thus avoiding the explicit construction of the new structure. However, consider an `f` defined like this:

```
function f(List[Integer] v)-->List[Integer] :
    v
end
```

Now, `f` returns its parameter directly, with the consequence that whenever the original list, the one that is the value of the mutable variable `a`, is changed, so will the value of `b`, because its implementation is backed by the original list.
Obviously, an implementation that tries to achieve good performance therefore needs to do some bookkeeping of which parts of a data structure could be mutated, and when these get assigned to some variable, either clone them immediately, or mark them for cloning in case they should ever be mutated. In either case, the behavior must be as if the mutable data structure was cloned right away.

7.2 Procedure call

The only predefined operation on procedures (cf. section 6.9.2) is *calling* them, i.e. invoking them with a number of arguments. Calling a procedure is written as follows:

CallStmt → Expression '(' Expressions ')'';'

The first expression must evaluate to a procedure, the other expressions must be of the appropriate argument types. The result of this statement is the *execution* of the procedure, with its formal parameters bound positionwise to the corresponding arguments.

7.3 Statement blocks (begin ... end)

Statement blocks are essentially syntactic sugar for a special case of the call statement, used to introduce a local scope and local variables. Their syntax looks like this:[3]

local scope

BlockStmt → **begin** [**var** VarDecls **do**] { Statement } **end**

The form

`begin var decls do stmts end`

is equivalent to the following procedure call:

`proc () var decls do stmts end () ;`

special case of procedure call

7.4 If-Statement

The if-statement is the most simple control-flow construct:

IfStmt → **if** Expression **then** { Statement } [**else** { Statement }] **end**

As is to be expected, the statements following the **then** are executed only if the expression evaluates to **true**, otherwise the statements following the **else** are executed, if present. The expression must be of type `Boolean`.

7.5 While-Statement

Iteration constructs are used to repeatedly execute a sequence of statements. A **while**-construct repeats execution of the statements as long as a condition specified by a `Boolean` expression is true.

---

[3] Note that this is the one exception from the general rule that each keyword construct can end with an end marker that consists of the string `end` and the opening keyword. The keyword `endbegin` would have looked too awful.

WhileStmt → **while** Expression [**var** VarDecls] **do** [Statements] (**end**|**endwhile**)

It is an error for the while-statement to not terminate.

7.6 Foreach-Statement

iteration over collections

The **foreach**-construct allows to iterate over collections, successively binding variables to the elements of the collections and executing a sequence of statements for each such binding.

ForeachStmt → ForeachGenerator { ',' ForeachGenerator }
[**var** VarDecls] **do** [Statements] (**end**|**endforeach**)

ForeachGenerator → **foreach** [Type] ID { ',' ID } **in** Expression
[',' Expressions]

relation to comprehensions
⟶ 6.10.2

The basic structure and execution mechanics of the foreach-statement is not unlike that of the comprehensions with generators discussed in section 6.10.2. However, where in the case of comprehensions a collection was constructed piecewise through a number of steps specified by the generators, a foreach-statement executes a sequence of statements for each complete binding of its generator variables.

Example 13. The following code fragment

```
s := 0;
foreach a in {1, 2}, b in {1, 2}:
  s := s + a*b;
end
```

results in s containing the number 9.

foreach $v_{1,1}, ..., v_{1,M_1}$ in C_1, $F_{1,1}, ..., F_{1,K_1}$, ...,
foreach $v_{n,1}, ..., v_{n,M_n}$ in C_n, $F_{n,1}, ..., F_{n,K_n}$
var D do B end

Foreach-statement semantics. In order to precisely describe the execution of a foreach-statement, we need to introduce a few symbols first. We call the original foreach-statement S. The (optional) declarations we write as D, and the body of the foreach-statement as B. Now S has $N \geq 1$ generators. The i-th generator has $M_i \geq 1$ variables, with names $v_{i,1}, ..., v_{i,M_i}$. The collection expression of the i-th generator will be called C_i. Following the i-th generator are $K_i \geq 0$ filter expressions, which we call $F_{i,1}, ..., F_{i,K_i}$.

syntactical transformation of foreach-statement

We can now define the meaning of the foreach-statement by replacing it with a number of previously described constructs, such as procedure closures, procedure call, and if-statement. The key to this replacement is a function `$iterate`, which we will describe below.

$$
\begin{aligned}
S &\equiv G(1) \\
G(i) &\equiv \begin{cases} \underline{\text{begin var}}\ D\ \underline{\text{do}}\ B\ \underline{\text{end}} & i > N \\ \underline{\text{begin var}}\ a_i \underline{\equiv} C_i\ \underline{\text{do}}\ G_V(i,1)\ \underline{\text{end}} & \text{otherwise} \end{cases} \\
G_V(i,j) &\equiv \begin{cases} G_F(i,1) & j > M_i \\ \underline{\text{\$iterate(}}a_i\underline{\text{, proc (}}v_{i,j}\underline{\text{) do}}\ G_V(i,j+1)\ \underline{\text{end)}} & \text{otherwise} \end{cases} \\
G_F(i,k) &\equiv \begin{cases} G(i+1) & k > K_i \\ \underline{\text{if}}\ F_{i,k}\ \underline{\text{then}}\ G_F(i,k+1)\ \underline{\text{end}} & \text{otherwise} \end{cases}
\end{aligned}
$$

The a_i are distinct and fresh variable symbols, i.e. they are mutually different, and they also differ from any variable symbol occurring inside the foreach-statement.

The `$iterate` procedure takes two arguments, a collection and a unary procedure. It iterates over the elements of its first argument, and calls the procedure on each element. `$iterate` is defined in appendix C.2.1.

`$iterate:`
`Collection[T]`
`[T-->]`
Def. ⟶ C.2.1

See also page 52 for an explanation of the iterator name choice, and page 52 for a note concerning the efficiency of generator implementations via syntactic substitution.

7.7 Choose-Statement

The choose-statement permits the explicit specification of (potentially nondeterministic) choice among a set of alternatives.

ChooseStmt ⟶ ChooseGenerator { ',' ChooseGenerator }
[**var** VarDecls] **do** [Statements]
[**else**[[**var** LocalVarDecls] **do**] [Statements]] (**end**|**endchoose**)

ChooseGenerator ⟶ **choose** [Type] ID [',' IDs] **in** Expression [',' Expressions]

The binding of values to variables happens very much like in the case of comprehension generators or the foreach-statement—see below for a precise definition of the semantics.

In contrast to the foreach-statement, the choose-statement executes its body at most once, viz. for the first combination of values that satisfy all the filters (which are part of the generators just as in the case of the foreach-statement or collection comprehensions). If no such combination can be found, and if an else-branch is present, the statements following the else-keyword are executed instead.

Choose-statement semantics. In order to precisely describe the execution of a choose-statement, we need to introduce a few symbols first. We call the orig-

choose $v_{1,1}, \ldots, v_{1,M_1}$ in C_1, $F_{1,1}, \ldots, F_{1,K_1}, \ldots$,
choose $v_{n,1}, \ldots, v_{n,M_n}$ in C_n, $F_{n,1}, \ldots, F_{n,K_n}$
var D do B else E end

inal choose-statement S. The (optional) declarations we write as D, and the body of the foreach-statement as B, the else branch is E. Now S has $N \geq 1$ generators. The i-th generator has $M_i \geq 1$ variables, with names $v_{i,1}, \ldots, v_{i,M_i}$. The collection expression of the i-th generator will be called C_i. Following the i-th generator are $K_i \geq 0$ filter expressions, which we call $F_{i,1}, \ldots, F_{i,K_i}$.

We can now define the meaning of the choose-statement by replacing it with a number of previously described constructs, such as procedure closures, procedure call, and if-statement. The key to this replacement is a function `$try`, which we will describe below.

syntactical transformation of choose-statement

$$S \equiv \underline{\text{begin var}}\ a_*\ \underline{:= \text{false do}}\ G(1)\ \underline{\text{if not}}\ a_*\ \underline{\text{then}}\ E\ \underline{\text{end end}}$$

$$G(i) \equiv \begin{cases} \underline{\text{begin var}}\ D\ \underline{\text{do}}\ a_*\ \underline{:= \text{true;}}\ B\ \underline{\text{end}} & i > N \\ \underline{\text{begin var}}\ a_i\ \underline{\equiv}\ C_i\ \underline{\text{do}}\ G_V(i,1)\ \underline{\text{end}} & \text{otherwise} \end{cases}$$

$$G_V(i,j) \equiv \begin{cases} G_F(i,1) & j > M_i \\ \underline{\$\text{try}(}a_i\underline{,\ \text{proc}\ (}v_{i,j}\underline{)\ \text{do}}\ G_V(i,j+1)\ \underline{\text{end, lambda}() :}\ a_*\ \underline{\text{end})} & \text{otherwise} \end{cases}$$

$$G_F(i,k) \equiv \begin{cases} G(i+1) & k > K_i \\ \underline{\text{if}}\ F_{i,k}\ \underline{\text{then}}\ G_F(i,k+1)\ \underline{\text{end}} & \text{otherwise} \end{cases}$$

The a_i and a_* are distinct and fresh variable symbols, i.e. they are mutually different, and they also differ from any variable symbol occurring inside the choose-statement.

`$try`:
`Collection[T]`
`[T-->]`
`[-->Boolean]`
Def. ⟶ C.2.1

The `$try` procedure takes three arguments, a collection, a unary procedure, and a nullary function. It iterates over the elements of its first argument, and for each element it does the following: It applies the function (to no arguments, as it is nullary). If the value of that application is false, it proceeds to call the procedure (its second argument) on the element, otherwise it simply returns (disregarding any subsequent elements of the collection).

See also page 52 for an explanation of the iterator name choice, and page 52 for a note concerning the efficiency of generator implementations via syntactic substitution. `$try` is defined in appendix C.2.1.

RATIONALE.
Having a construct that explicitly allows the use of non-determinism (as opposed to expressions, such as map comprehensions, that may evaluate nondeterministically, but where this behavior is most likely unwanted and potentially erroneous) allows actor authors to express internal choice points which are explicitly not under the control of the actor context (i.e. the model of computation).
The responsibility for many other choices, such as which action to fire, can be assumed by the model of computation. Using a choose-statement inside an action provides an explicit signal that this action may be nondeterministic, and allows the model of computation to either reject it, or deal with it accordingly.

backtracking

Example 14. The choose-statement can be used to implement simple backtracking algorithms, as in the following example:

```
s := null;
choose
  a in [1, 2],
choose
  b in if a = 1 then {} else {3, 4} end
do
  s := [a, b];
else
  s := [];
end
```

Since for `a = 1` the collection for `b` is the empty set, the body will never be executed for this value of `a`, because no choice could be made for `b`. However, if `a = 2`, two choices can be made for `b`, and it is unspecified which one is made. After executing this statement, `s` will be either `[2, 3]` or `[2, 4]`.

It will never be `[]`, because the else branch is only executed if no choice can be made.

Chapter 8

Actions

action ≡ family of transitions

An *action* in CAL represents a (often large or even infinite) number of *transitions* of the actor transition system described in section 10.3. A CAL actor description can contain any number of actions, including none. The definition of an action includes the following information:

- its *input tokens,*
- its *output tokens,*
- the *state change* of the actor,
- additional *firing conditions,*
- the *time delay.*

choice between actions

In any given state, an actor may take any number of transitions (including zero), and these transitions may be represented by any number of actions in the actor description. The choice between them is ultimately made by the context of an actor (though the actor can constrain the possible choices, see chapter 9)—see section 8.4 for details.

The syntax of an action definition is as follows:

Action → [ActionTag ':'] **action** ActionHead [**do** Statements] (**end**|**endaction**)

ActionTag → ID { '.' ID }

ActionHead → InputPatterns '==>' OutputExpressions
    [**guard** Expressions] [**var** VarDecls] [**delay** Expression]

action tags

Actions are optionally preceded by *action tags* which come in the form of qualified identifiers (i.e. sequences of identifiers separated by dots), see also section 9.1. These tags need not be unique, i.e. the same tag may be used for more than one action. Action tags are used to refer to actions, or sets of actions, in action schedules and action priority orders—see chapter 9 for details.

The head of an action contains a description of the kind of inputs this action applies to, as well as the output it produces. The body of the action is a sequence of statements that can change the state, or compute values for local variables that can be used inside the output expressions.

Input patterns and output expressions are associated with ports either by position or by name. These two kinds of association cannot be mixed. So if the actor's port signature is

```
Input1, Input2 ==> ...
```

an input pattern may look like this:

```
[a], [b, c]
```

(binding a to the first token coming in on `Input1`, and binding b and c to the first two tokens from `Input2`). It may also look like this:

```
Input2:  [c]
```

but never like this:

```
[d] Input2:[e]
```

This mechanism is the same for input patterns and output expressions.

The following sections elaborate on the structure of the input patterns and output expressions describing the input and output behavior of an action, as well as the way the action is selected from the set of all actions of an actor.

actions declarative

In discussing the meaning of actions and their parts it is important to keep in mind that the interpretation of actions is left to the model of computation, and is not a property of the actor itself. It is therefore best to think of an action as a declarative description of how input tokens, output tokens, and state transitions are related to each other. See also section 8.4.

8.1 Input patterns, and variable declarations

input pattern:
activation condition &
variable declaration

Input patterns, together with variable declarations and guards, perform two main functions: (1) They define the input tokens required by an action to fire, i.e. they give the basic conditions for the action to be *firable* which may depend on the value and number of input tokens and on the actor state, and (2) they declare a number of variables which can be used in the remainder of the action to refer to the input tokens themselves. This is their syntax:

InputPattern → [ID ':'] '[' IDs ']' [RepeatClause] [ChannelSelector]

ChannelSelector → **at** Expression
| **at*** Expression
| [**at***] **any**
| [**at***] **all**

RepeatClause → **repeat** Expression

multiport pattern
—channel selector

Input patterns differ depending on whether they are written for a single port or a multiport—the former represents one *channel* of incoming tokens,

	no repeat-clause	with repeat-clause
single channel	`T`	`List[T]`
multichannel	`Map[CID, T]`	`Map[CID, List[T]]`

Table 8.1: Token variable types depending on input pattern kind and presence of repeat-clause. (`T` is the token type of the corresponding input port.)

while the latter represents an arbitrary number (including zero) of channels. A multiport input pattern has to have a *channel selector* associated with it, which is a construction that serves to specify the channels that a pattern is applied to (i.e. that the tokens bound by the pattern are read from).

The static type of the variables declared in an input pattern depends on the token type declared on the input port, but also on the kind of pattern. Fig. 8.1 shows this dependency. It distinguishes between patterns according to whether they contain a repeat-clause, and whether they take tokens from one channel (*single-channel patterns*), or any number of channels (*multichannel patterns*). Channels are specified using *channel identifiers*, which are objects identifying a specific channel inside a multiport. Usually, channel identifiers are represented by some other type in some platform-dependent manner. For instance, non-negative integer numbers can be used to identify channels. Fig. 8.1 refers to the type of channel identifiers as *CID*.

8.1.1 Single-port input patterns

A single-port input pattern binds a number of tokens from the input channel associated with the specified port to variables specified as part of the pattern. Because there is exactly one input channel associated with a single port, these patterns are always single-channel patterns.

single-port ⟹ single-channel

SinglePortInputPattern → [ID ':'] '[' IDs ']' [RepeatExpression]

A pattern without a repeat-expression is just a number of variable names inside square brackets. The pattern binds each of the variable names to one token, reading as many tokens as there are variable names. The number of variable names is also referred to as the *pattern length*. The static type of the variables is the same as the token type of the corresponding port (Fig. 8.1).

Example 15 (Single-port input pattern). Assume the sequence of tokens on the input channel is the natural numbers starting at 1, i.e.

`1, 2, 3, 4, 5, ...`

The input pattern `[a, b, c]` results in the following bindings:

`a = 1, b = 2, c = 3`

If the pattern contains a repeat-clause, that expression must evaluate to a non-negative integer, say N. If the pattern length is L, the number of tokens read by this input pattern and bound to the L pattern variables is NL. Since in general there may be more tokens than variables (N times more, exactly), variables are bound to *lists* of tokens, each list being of length N. In the pattern, the list bound to the k-th variable contains the tokens numbered k, $L+k$, $2L+k$, ..., $(N-1)L+k$. The static type of these variables is `List[T]`, where `T` is the token type of the port (Fig. 8.1).

Example 16 (Single-port input pattern with repeat-clause). Assume again the natural numbers as input sequence. If the input pattern is

```
[a, b, c] repeat 2
```

it will produce the following bindings:

```
a = [1, 4], b = [2, 5], c = [3, 6]
```

8.1.2 Multiport input patterns

channel selector: one or many

Multiport input patterns contain an expression that functions as a *channel selector*, i.e. it specifies which channels to read tokens from. There are two kinds of channel selectors—those that identify precisely one channel, and those that identify any number of channels.

Single-channel channel selectors. The at-construct is used to select one channel from a multiport. The expression following the at-keyword must evaluate to a valid channel identifier for the given port. The reading of tokens and binding to the pattern variables happens in the same way as described for single-port input patterns above.

Example 17 (at channel selector). Assume four input channels `'a'`, `'b'`, `'c'`, `'d'` with the following input tokens:

```
'a':  11, 12, 13, 14, 15, 16, 17, 18, 19
'b':  21, 22, 23, 24, 25, 26, 27, 28, 29
'c':  31, 32, 33, 34, 35, 36, 37, 38, 39
'd':  41, 42, 43, 44, 45, 46, 47, 48, 49
```

Then the input pattern

```
[a, b, c] repeat 2 at 'c'
```

would yield the bindings

```
a = [31, 34], b = [32, 35], c = [33, 36]
```

different ways to determine channel set

Multichannel channel selectors. CAL provides three constructs for selecting tokens from more than one channel of a multiport at the same time. They are distinguished by the keyword that introduces them: **at***, **any**, **all**. They differ in the way they determine the set of channels to read from as follows:

1. Following the at*-keyword is an expression that computes a (possibly empty) collection of channel identifiers.

2. The **any**-keyword results in those channels being read from that have a sufficient number of tokens available to be bound to the variables of the pattern—which could be none at all, in which case the set of channels would be empty.

3. The **all**-keyword selects all channels of the multiport.

All multichannel input patterns have in common that they are applied *homogeneously*, i.e. the same number of tokens is read from all selected channels (and no tokens are read from any of the other channels). The variables are bound to *maps*—the keys of these maps are channel identifiers, while the values are, depending on the absence or presence of a repeat-clause, either tokens or lists of tokens. Fig. 8.1 gives the static types of the pattern variables for multichannel patterns.

Example 18 (Multichannel input patterns). Assume four input channels `'a'`, `'b'`, `'c'`, `'d'` with the following input tokens:

```
'a':  11, 12, 13, 14, 15, 16, 17, 18, 19
'b':  21, 22, 23, 24, 25
'c':  31, 32, 33, 34, 35, 36, 37, 38, 39
'd':  41, 42, 43
```

The input pattern

```
[a, b, c] at* {'a', 'c'}
```

yields the bindings

```
a = map{'a'->11, 'c'->31}
b = map{'a'->12, 'c'->32}
c = map{'a'->13, 'c'->33}
```

The input pattern

```
[a, b] all
```

results in the bindings

```
a = map{'a'->11, 'b'->21, 'c'->31, 'd'->41}
b = map{'a'->12, 'b'->22, 'c'->32, 'd'->42}
```

In this case, the keyword **any** instead of **all** would have produced the same result.

The input pattern

```
[a, b] repeat 2 any
```

results in the bindings

```
a = map{'a'->[11, 13], 'b'->[21, 23], 'c'->[31, 33]}
b = map{'a'->[12, 14], 'b'->[22, 24], 'c'->[32, 34]}
```

In this case, **all** instead of **any** would not have worked, because the channel `'d'` does not provide the required number of tokens.

8.1.3 Scoping of action variables

The scope of the variables inside the input patterns, as well as the explicitly declared variables in the var-clause of an action is the entire action—as a consequence, these variables can depend on each other. The general scoping rules from section 5.2 need to be adapted in order to properly handle this situation.

pattern variables ↔ var-variables
F_v for pattern variables ⟶ 5.2, Def. 1

In particular, input pattern variables do not have any initialization expression that would make them depend explicitly on any other variable. However, their values clearly depend on the expressions in the repeat-clause and the channel selector (if present). For this reason, for any input pattern variable v, we define the set of free variables of its initialization expression F_v to be the union of the free variables of the corresponding expressions in the repeat-clause and the channel selector.

The permissible dependencies then follow from the rules in section 5.2.

Example 19 (Action variable scope). The following action skeleton contains dependencies between input pattern variables and explicitly declared variables

```
[n] at c, [k], [a] repeat m * n ==> ...
var
    m = k * k, c = f(m)
do ... end
```

These declarations are well-formed, because the variables can be evaluated in the order k, m, c, n, a.

By contrast, the following action heads create **circular dependencies** and are therefore **errors**:

```
[n] at f(a), [a] repeat n ==> ... do ... end

[a] repeat a[0] + 1 ==> ... do ... end

[a] repeat n ==> ...
var
    n = f(b), b = sum(a)
do ... end

[a] at c, [b] repeat a ==> ...
var
    c = g(b)
do ... end
```

8.2 Output expressions

output expressions dual of input patterns

Output expressions are conceptually the dual notion to input patterns—they are syntactically similar, but rather than containing a list of variable names which get bound to input tokens they contain a list of expressions that computes the output tokens, the so-called *token expressions*.

	no repeat-clause	with repeat-clause
single channel	`T`	`Seq[T]`
multichannel	`Map[CID, T]`	`Map[CID, Seq[T]]`

Table 8.2: Token expression types depending on output expression kind and presence of a repeat-clause. (`T` is the token type of the corresponding output port.)

OutputExpression → [ID ':'] '[' Expressions ']' [RepeatClause] [ChannelSelector]

ChannelSelector → **at** Expression
| **at*** Expression
| [**at***] **any**
| [**at***] **all**

RepeatClause → **repeat** Expression

The repeat-clause and channel selector work not unlike in the case of input patterns, but with one crucial difference. For input patterns, these constructs control the *construction* of a data structure that was assembled from input tokens and then bound the pattern variables. In the case out output expressions, the values computed by the token expressions are themselves these data structures, and they are indexed into according to the repeat-clause and the channel selector, if these are present.

deconstructing output expression values

The table in Fig. 8.2 shows the kind of data structure a token expression has to evaluate to depending on the presence or absence of a repeat-clause, and depending on whether the output expression is a single-channel or a multichannel output expression.

Single-channel output expressions. In single-channel output expressions without repeat-clause, the token expressions represent the output tokens directly, and the number of output tokens produced is equal to the number of token expressions. If a single-channel output expression does have a repeat-clause, the token expressions must evaluate to sequences of tokens, and the number of tokens produced is the product of the number of token expressions and the value of the repeat-expression. In addition, the value of the repeat-expression is the minimum number of tokens each of the sequences must contain.

repeat-clause: lists must be long enough!

Example 20 (Single-channel output expressions.). The output expression

```
... ==> [1, 2, 3]
```

produces the output tokens `1, 2, 3`.

The output expression

```
... ==> [[1, 2, 3], [4, 5], [6, 7, 8]] repeat 2
```

produces the output tokens `1, 4, 6, 2, 5, 7`.

The output expression

```
    ...  ==> [[1, 2, 3], [4, 5]] repeat 1 at 'b'
```

produces the output tokens `1, 4` on channel `'b'`.

RATIONALE.
One of the key reasons for interpreting the repeat-construct in output expressions as described above is to maintain symmetry with the way input patterns are handled. For instance, consider the following action:

```
action [a, b, c] repeat k ==> [a, b, c] repeat k end
```

This action copies its input tokens identically to its output, $3k$ tokens at a time. It is, therefore, equivalent to the action

```
action [a] repeat 3 * k ==> [a] repeat 3 * k end
```

The same rationale applies to multichannel output expressions, see below. Here, too, the design strives for a symmetry between the input patterns and output expressions, as far as their fundamental directedness allows (the former declares variables, while the latter uses them to compute values).

channel selector constrains token map domains

Multichannel output expressions. Similarly, for multichannel output expressions without a repeat expression, the token expressions must evaluate to maps from channel identifiers to tokens. If a repeat-clause is present, the maps must map channel identifiers to sequences of tokens, the number of elements on those sequences being not smaller than the value of the repeat expression. The channel selector places some constraints on the domains of the maps as follows:

- For an **at*** channel selector, the intersection of the domains of the maps must contain the collection of channel identifiers the selector expression evaluates to.
- For an **all** channel selector, the intersection of all domains must contain the set of all channel identifiers currently valid for the corresponding port.
- For a **any** channel selector, no constraints apply to the domains. The channels that output will be produced on are those in the intersection of all domains, which may be empty.

homogeneous output production

These rules are designed to make it possible to produce output *homogeneously* on all selected output channels—in other words, all output channels of a multiport output the same number of tokens if they output any tokens at all.

Example 21 (Multichannel output expressions.). The output expression

```
... ==> [map {'a'->1, 'b'->2},
         map {'a'->3, 'b'->4,'c'->5}] at* {'a', 'b'}
```

produces the following output on the channels `'a'` and `'b'`:

```
a:  1, 3
b:  2, 4
```

Nothing is output on channel `'c'`.

The output expression

```
... ==> [map {'a'->[1, 2], 'b'->[3, 4]},
         map {'a'->[5, 6], 'b'->[7, 8]}]
         repeat 2 at* {'a', 'b'}
```

results in the following output:

```
a:  1, 5, 2, 6
b:  3, 7, 4, 8
```

8.3 Delays

uninterpreted delays

The expression following the **delay**-keyword specifies the delay of the action. The actor language itself does not attach any meaning to this information, other than that it is part of the label of a state transition of an actor. See section 10.2 for the precise meaning of time and delay, and section 3.2 for some more information on time inside the CAL language.

If the actor contains a time-clause, then the static type of each delay expression must be a subtype of the type specified in that time-clause.

8.4 On action selection: guards and other activation conditions

activation conditions

At any given point during the execution of an actor, an action may potentially fire on some input data. Whether it is *activated*, i.e. whether in a given situation it actually can fire, depends on whether its *activation conditions* have all been met. The minimal conditions are as follows:

1. According to the action schedule (see section 9.2) this action may fire next, according to Def. 4 on page 79.
2. No higher-priority action is activated (see section 9.3).
3. There are sufficient input tokens available to bind the input pattern variables to appropriate values.
4. Given such a binding, all guard expressions (which must be `Boolean` expressions) evaluate to **true**.

5. There is sufficient room for the output tokens produced by this firing.

Depending on the context in which the actor is embedded, additional conditions for activating actions may apply. A model of computation may reject an actor as non-well-formed if it cannot statically ascertain that it will meet the relevant activation conditions whenever it needs to.

additional conditions by model of computation

IMPLEMENTATION NOTE.
The last activation condition may not be easily satisfiable in the general case, as it may be unknown exactly how many output tokens a firing will produce until it has executed and the output tokens have been computed. The important point is that firing an actor has to appear to the outside as an atomic step—the physical time it takes is not directly relevant, and the execution may even stall due to a lack of space for the output tokens, but it is imperative that the firing looks as if it occurred in one atomic transition.

8.5 Initialization actions

special action:
no input, no invariants

Initialization actions are executed at the beginning of an actor's life cycle. They are very similar to regular actions, with two important differences:

1. Since the assumption is that at the beginning of an actor execution no input is available, initialization actions have no input patterns. They may produce output, however.
2. With the exception of initialization expressions in variable declarations, an initialization action contains the first code to be executed inside the actor. Any state invariants in the actor may not hold, and instead have to be established by the initialization action.

The syntax of initialization actions is as follows:

InitializationAction → [ActionTag ':']
  initialize InitializerHead [**do** Statements] (**end**|**endinitialize**)

InitializerHead → '==>' OutputExpressions
  [**guard** Expressions] [**var** VarDecls] [**delay** Expression]

activation conditions

The activation conditions for actions apply also to initialization actions—of course, since there is no input, the conditions concerning input tokens become vacuously true.

If an actor has more than one initialization action, and if more than one is activated at the beginning of an actor execution, one of them is chosen arbitrarily. The actor context can of course make than choice.

only one init action gets fired

Chapter 9

Action-level control structures

In CAL, an action expresses a relation between the state of an actor and input tokens, and the successor state of the actor and output tokens. In general, CAL actors may contain any number of actions, and in a given situation, any subset of those may be ready to be executed. For example, both actions of the following actor may be able to execute, if there is a token available on either input port:

ambiguous action choice

Example 22 (Nondeterministic Merge).

```
actor NDMerge () A, B ==> C :

  action A: [x] ==> [x] end

  action B: [x] ==> [x] end
end
```

It is important to emphasize that the policy used to choose between the two actions above is not part of the actor specification. In practice, the context of the actor will make some choice, based on whatever policy it implements. This flexibility may be desired, but sometimes the actor writer may want to have more control over the choice of the action—e.g., if the `Merge` actor is supposed to alternate between reading its input ports, one might use actor state to realize this behavior:

control over action selection

Example 23 (Basic FairMerge).

using state to control action selection

```
actor FairMerge () A, B ==> C :

  s := 1;

  action A: [x] ==> [x]
  guard s = 1
  do
```

```
    s := 2;
  end

  action B: [x] ==> [x]
  guard s = 2
  do
    s := 1;
  end
end
```

schedules easier to read and to analyze

This way of specifying action choice has two key drawbacks. First, it is very cumbersome to write and maintain, and it does not scale very well even for modest numbers of actions and states. Furthermore, this way of specifying action choice essentially obfuscates the "real" logic behind guards, state variables, and assignments, so that it becomes harder to extract the intent from the actor description, both for tools and for human readers.

These are the key motivations for using *action schedules*, i.e. structured descriptions of possible orders in which actions may fire. Before we can discuss action schedules in section 9.2, we need to take a closer look at how actions are referred to inside of them.

9.1 Action tags

Actions are optionally prefixed with *action tags* (see chapter 8), which are qualified identifiers:

ActionTag → QualID
QualID → ID { '.' ID }

multiple actions per tag

The same tag may be used for more than one action. In the following, we write the set of all actions tagged by a tag t as $\bar{t}$, and the tag of some action a as t_a. The empty tag is written as ϵ, and the set of all untagged actions is therefore $\bar{\epsilon}$.

prefix order among tags

Action tags are ordered by a *prefix ordering*: We say that $t \sqsubseteq t'$, i.e. t is a *prefix* of t', if t' starts with all the identifiers in t in the same order, followed by any number of additional identifiers, including none. For instance, $a.b.c \sqsubseteq a.b.c.x$ and $a.b \sqsubseteq a.b$, but $a.b \not\sqsubseteq a.c$. We call t' an *extension* of t.

tag: denoted action set

When used inside action schedules and priority orderings, a tag denotes the set of actions which are labeled with tags that are extensions of it. For any tag t this set is called $\hat{t}$ and is defined as follows:

$$\hat{t} =_{def} \{a \mid t \sqsubseteq t_a\}$$

9.2 Action schedules

Action schedules are structured descriptions of possible sequences in which the actions of an actor may be executed. These sequences are either specified by a finite state machine or a regular expression in the alphabet of action tags. In general, like any regular language, the set of possible sequences may be finite or infinite.

fsm & regexp schedules

Irrespective of whether a finite state machine is used or a regular expression, an action schedule effectively describes a (regular) language $\mathcal{L}$ in the alphabet of action tags. This language is used to constrain the legal sequences of action firings as follows.

schedule = regular tag language

Definition 4 (Legal action sequence). Given a tag language $\mathcal{L}$, assume a finite sequence of actions $(a_i)_{i=1..n}$, and a sequence $(b_j)_{j=1..m}$ with $m \leq n$ and a strict monotonic function $f : \{1..m\} \longrightarrow \{1..n\}$ such that the following holds for all $j \in \{1..m\}$ and $i \in \{1..n\}$:

(b_j) tagged subsequence of (a_i)

1. $b_j = a_{f(j)}$
2. $t_{b_j} \neq \epsilon$
3. $t_{a_i} = \epsilon, \quad \forall i \notin f^{-1}[\{1..m\}]$

In other words, the (b_j) are the subsequence in the (a_i) with non-empty tags. If (b_j) is empty, i.e. $m = 0$, then (a_i) is a *legal action sequence.*

If (b_j) is not empty, i.e. $m \geq 1$, then (a_i) is a *legal action sequence* if and only if there exists a sequence of tags $(t_j)_{j=1..m}$ such that the following holds:

1. for all $j \in \{1..m\}$, $b_j \in \hat{t}_j$
2. there exists a $w \in \mathcal{L}$ such that $(t_j) \sqsubseteq w$.

A consequence of this definition is that untagged actions may occur at any point in the schedule—conversely, schedules do not constrain untagged actions in any way.

no constraints on untagged actions

We will now describe the two ways of defining a tag language: finite state machines and regular expressions.

9.2.1 Finite state machine schedules

A finite state machine schedule defines a number of transitions between states (and an initial state) that are each labeled with one or more action tags.

ScheduleFSM → **schedule [fsm]** ID ':'
    { StateTransition ';' }
    (end|endschedule)

StateTransition → ID '(' ActionTags ')' '-->' ID
    { '|' '(' ActionTags ')' '-->' ID }

ActionTags → ActionTag
    | ActionTag ',' ActionTags

The state before the colon is the initial state, and all states are accepting (or final states). The tag language is the set of all sequences of tags that label transitions leading from the initial state to any other state of the finite state machine.

Several transitions starting from the same state may be written as separated by the '|' character.

The following illustrates the use of a finite state machine action schedule to express the `FairMerge` actor somewhat more concisely.

Example 24 (FairMerge, with FSM schedule).

```
actor FairMerge1 () A, B ==> C :

  InA: action A: [x] ==> [x] end

  InB: action B: [x] ==> [x] end

  schedule fsm WaitA :
    WaitA (InA) --> WaitB;
    WaitB (InB) --> WaitA;
  end
end
```

9.2.2 Regular expression schedules

In many cases, using an explicit finite state machine to define an action schedule is still rather verbose, and specifying the tag language through a regular expression is much more straightforward. The syntax is as follows:

ScheduleRegExp → **schedule regexp** RegExp (**end**|**endschedule**)

RegExp → ActionTag
    | '(' RegExp ')'
    | '[' RegExp ']'
    | RegExp '*'
    | RegExp RegExp
    | RegExp '|' RegExp

The simplest regular expression (regexp) is simply a tag, which denotes the language consisting of just this tag. Parentheses are used to explicitly group regexps. The square brackets enclose an optional regexp, describing the language that consists of all sequences described by the regexp, as well as the empty sequence. The Kleene operator '*' means repetition any number of times (including zero), juxtaposition of two regexps denotes concatenation of sequences of the two languages, and the '|' operator describes alternative choice. The binding strength is the order given above, i.e. the regexp

operator precedence

```
A B* | C* D
```

is interpreted as

```
(A (B*)) | ((C*) D)
```

The `FairMerge` actor can be expressed with regexps as follows:

Example 25 (FairMerge, with regular expression schedule).

```
actor FairMerge2 () A, B ==> C :

  InA: action A: [x] ==> [x] end

  InB: action B: [x] ==> [x] end

  schedule regexp
    (InA InB)*
  end
end
```

RATIONALE.
The reason for this conceptual redundancy is convenience—many schedules are best written as regular expressions. On the other hand, in many cases producing a regular expression can be cumbersome, and finite state machines are preferable. This is especially true for generated code, which will usually use the finite state machine formulation.
Of course, both syntaxes are of identical expressiveness, and can therefore be translated into one another.

Using regular expressions, control structures among actions become much more concise and easier to read and to specify. For example, assume we do not want to predetermine which port of the `FairMerge` actor is read from first. It is very easy to augment the `FairMerge2` description to realize this:

Example 26 (Symmetric FairMerge).

```
actor FairMergeSymmetric () A, B ==> C :

  InA: action A: [x] ==> [x] end

  InB: action B: [x] ==> [x] end

  schedule regexp
    (InA InB)* | (InB InA)*
  end
end
```

9.3 Priorities

priorities increase expressiveness

allow test for absence of tokens

Priorities are very different from action schedules in that they genuinely add to the expressiveness of CAL—it would not be possible in general to reduce them to existing constructs, in the way schedules can in principle be reduced to a state variable and guards/assignments. Among other things, priorities allow actors to effectively test for the *absence* of tokens. As a consequence, actors can express non-prefix monotonic processes,[1] which is powerful, but at the same time can be dangerous, because it means the results computed by an actor may depend on the way it was scheduled with respect to the other actors in the system.

induced partial order on actions

Priorities are defined as a partial order relation over action tags, which induces a partial order relation over the actions. An action can only fire if there is no other *enabled* action that is higher in this partial order. The order is specified as follows:

PriorityOrder → **priority**{ PriorityInequality ';' } **end**

PriorityInequality → ActionTag '>' ActionTag { '>' ActionTag }

The priority inequalities are specified over tags, i.e. they have the form $t_1 > t_2$. These inequalities induce a binary relation on the actions are follows:

$$\begin{aligned} a_1 > a_2 \Longleftrightarrow & \exists t_1, t_2 : t_1 > t_2 \wedge a_1 \in \hat{t}_1 \wedge a_2 \in \hat{t}_2 \\ & \vee \exists a_3 : a_1 > a_3 \wedge a_3 > a_2 \end{aligned}$$

validity of priority system

The priority inequalities are *valid* iff the induced relation > on the actions is an irreflexive partial order, i.e. it is antisymmetric and transitive. Transitivity follows from the definition, but antisymmetry and irreflexivity do not. In fact, they do not even follow if the relation on the tags is a partial order. Consider the following example:

---

[1] See [10] for details on prefix monotonicity and its implications for the description of dataflow systems.

```
    A.B > X > A
```

This is obviously a proper order on the *tags*. However, if we have two actions labeled `X` and `A.B`, then the induced relation is not antisymmetric, because according to `A.B > X`, the action `A.B` has higher priority than `X`, while the inequality `X > A` implies that any action whose label starts with `X` has a higher priority than any action whose label starts with `A`, including, of course, `A.B`. Therefore, this system of priority inequalities is inconsistent and thus invalid.

With priorities, we can express a *Merge* actor that prefers one input over the other like this:

Example 27 (BiasedMerge).

```
actor BiasedMerge () A, B ==> C :

  InA: action A: [x] ==> [x] end
  InB: action B: [x] ==> [x] end

  priority
    InA > InB;
  end
end
```

Perhaps more interestingly, we can express a merge actor that is fair, in the sense that it will consume equal amounts of tokens from both inputs as long as they are available, but will not halt due to lack of tokens on only one of its input ports. It is also nondeterministic, i.e. it does not specify the order in which it outputs the tokens.

Example 28 (FairMerge, with priorities).

```
actor FairMerge3 () A, B ==> C :

  Both: action [x], [y] ==> [x, y] end
  Both: action [x], [y] ==> [y, x] end
  One:  action A: [x] ==> [x] end
  One:  action B: [x] ==> [x] end

  priority
    Both > One;
  end
end
```

Part II

Semantics

Chapter 10

Actor model

language:
finite representation of infinite structure

This chapter presents the formal definition of our notion of *actor*. In general, actors may be highly complex and infinite computational structures, which can be non-deterministic, contain and manipulate internal state, and consume and produce units of data (*tokens*). The purpose of the CAL language (as that of any other programming language) can be seen as providing a finite representation of such structures, while exhibiting their inner structures and regularities in a way that allows tools to identify and check them, and to use them when composing actors or when generating implementations from actor descriptions.

10.1 Preliminaries

universe $\mathcal{U}$
finite sequences $\mathbb{S}$
infinite sequences $\mathbb{S}_\infty$

In the following, we assume a *universe* of all token values $\mathcal{U}$ that can be exchanged between actors. The communication along each connection between actors can be viewed as a sequential stream of tokens, and actors will remove tokens from this stream and add tokens to it. We define the set $\mathbb{S} =_{def} \mathcal{U}^*$ as the set of all finite sequences over $\mathcal{U}$. The set $\mathbb{S}_\infty =_{def} \mathbb{S} \cup \mathbb{S}^{\mathbb{N}}$ is the set of all finite and infinite sequences over $\mathcal{U}$. We write the empty sequence as λ. The length of a finite sequence s is denoted by $| s |$.

prefix order

The elements of $\mathbb{S}_\infty$ (and consequently $\mathbb{S}$) are partially ordered by their *prefix relation*: $s \sqsubseteq r$ iff s is a prefix of r, i.e. r starts with the tokens of s in the order they occur in s. For example, abc $\sqsubseteq$ abcd, and ab $\sqsubseteq$ ab, but abc $\not\sqsubseteq$ cabd. Note that for any $s \in \mathbb{S}_\infty$, $\lambda \sqsubseteq s$ and $s \sqsubseteq s$.

... extends to $\mathbb{S}^n$

Many actors have multiple input and output sequences, so most of the time we will work with tuples of sequences, i.e. with elements of $\mathbb{S}^n$ or $\mathbb{S}^n_\infty$ for some n. The prefix order extends naturally to tuples as follows:

$$(s_i)_{i=1..n} \sqsubseteq (r_i)_{i=1..n} \Leftrightarrow_{def} \forall i = 1..n : s_i \sqsubseteq r_i$$

Note that since $\mathbb{S}_\infty$ and $\mathbb{S}^n_\infty$ under the prefix order both have a least element, and every chain in them has a least upper bound, they are complete partial orders. This property is relevant when relating dataflow models with a notion of firing to Kahn process networks, as in [10, 7].

In the following we use *projections* from $\mathbb{S}^m$ to $\mathbb{S}^n$, with $m \geq n$. These are functions that extract from an m-tuple an n-tuple such that for each value its number of occurrences in the argument is not smaller than the number of occurrences in the result. We write projections using the letter π with appropriate subscripts, and use them to map the input and output of an actor onto a subset of its ports. A special kind of projection maps a tuple onto one port, say p, of the actor. We write this projection as π_p.

10.2 Time systems

We want to allow actors to have a notion of time, without committing to any particular system of time stamps (or "tags", cf. [12]). Instead, we introduce the notion of *time systems*, which are basically sets of time stamps with the properties and operations we need to manipulate them.

Definition 5 (Time system). A *time system* is a structure

$$(T, \Delta, z, +, \leq)$$

with the following properties:

1. T and Δ are sets of time *tags* and *delays*, respectively.
2. $\leq$ is a reflexive partial order on T
3. $+ : T \times \Delta \longrightarrow T$ is defined such that $\forall t \in T, \delta \in \Delta : t \leq t + \delta$
4. $z \in \Delta$ is the *zero delay* such that $\forall t \in T : t + z = t$.

temporal metric

Many time systems have a *temporal metric* associated with them that defines a quantitative measure of the "distance" between two time tags. If we use a time system together with such a metric we call it a *metric time system*.

Definition 6 (Temporal metric/distance). Given a time system $(T, \Delta, z, +, \leq)$, a partial function

$$d : T \times T \longrightarrow \mathbb{R}$$

is a *temporal metric* of this system iff it has the following properties for all $t_1, t_2, t_3 \in T$:

- $t_1 \leq t_2 \Rightarrow d(t_1, t_2) \geq 0$
- $d(t_1, t_2) = -d(t_2, t_1)$
- $t_1 \leq t_2 \leq t_3 \Rightarrow d(t_1, t_2) \leq d(t_1, t_3) \wedge d(t_2, t_3) \leq d(t_1, t_3)$
- $t_1 \leq t_2 \leq t_3 \Rightarrow d(t_1, t_2) + d(t_2, t_3) \geq d(t_1, t_3)$

For any two time tags t_1, t_2 the value $d(t_1, t_2)$, if defined, is called their *(directed) temporal distance*.

Note. d differs from what is usually called a "metric" in several important ways. First of all it is a partial function, and is required to only quantify distances between time tags that are comparable inside the time system. Also, it allows distinct time tags to have a zero temporal distance, and it reflects the "directedness" of the time system by allowing for negative distances.

temporal "metric"

trivial temporal metric

Since a temporal metric is not required to return a non-zero value for distinct time tags, there is always a trivial temporal metric for each time system, viz. the one that yields 0 for any two tags.

Example 29. Some common time systems are the following:

- A familiar time system would be

  $$(\mathbb{R}, \mathbb{R}_0^+, \leq, +)$$

  where the comparisons and operators are the common ones on the real numbers. Its temporal metric is subtraction.

- A somewhat more sophisticated time system would be one that employs a notion of *delta time*:

  $$(\mathbb{R} \times \mathbb{N}, \mathbb{R}_0^+ \times \mathbb{N}, \leq_{\mathrm{LEX}}, +^n)$$

  The time tags are tuples of a real time stamp and an additional natural number index. Here, $T = \mathbb{R} \times \mathbb{N}$, $\Delta = \mathbb{R}_0^+ \times \mathbb{N}$, $+$ adds two tuples positionwise, the order is the lexicographic order on the tuples, and the d is simply subtraction in the first (real-valued) component. For instance, $(-3.4, 2) + (5, 4) = (1.6, 6)$, and $(1.2, 50) \leq_{\mathrm{LEX}} (1.21, 5) \leq_{\mathrm{LEX}} (1.21, 6)$, and $(1, 6) \not\leq_{\mathrm{LEX}} (2, 5)$.

  A common metric that is used with this time system computes the difference of the first component, e.g. $d((1.2, 5), (1.21, 50)) = 0.01$.

- Another time system is *vector time*. Each time tag (and delay) is an n-ary vector of natural numbers:

  $$(\mathbb{N}^n, \mathbb{N}^n, \leq^n, +^n)$$

  Addition is the usual vector addition as in the previous example. The comparison compares two vectors positionwise, and two vectors are only comparable iff all their components are related to each other in the same way—e.g., $(1, 2, 3) \leq^3 (2, 3, 4)$ but $(1, 2, 3) \not\leq^3 (2, 3, 1)$ and also $(2, 3, 1) \not\leq^3 (1, 2, 3)$. In other words, unlike the previous two examples, this time system is not totally ordered.

  It is also a system that is often not associated with a temporal metric. Many possible candidate metrics exist, however, such as sum or product of positionwise differences, or for each component of the vector the difference in that position.

10.3 Actor transition systems

We can now define an actor as an entity that has input/output ports through which is interacts with its environment by receiving and sending tokens, that makes atomic steps, and that in each such step consumes and produces a finite (and possibly empty) sequence of tokens at each input or output port. It also has a state (which we will not further characterize), which may change during this step.

Definition 7 (Actor transition system, actor, ATS). Let $\mathcal{U}$ be the *universe* of all values, and $\mathbb{S} = \mathcal{U}^*$ be the set of all finite sequences in $\mathcal{U}$. Further, let $(T, \Delta, z, +, \leq)$ be a time system. For any non-empty set Σ of *states* an *m-to-n actor transition system* (or just *actor* or *ATS* for short) is a labeled transition system

$$\langle \sigma_0, \Delta, \tau, \succ \rangle$$

with $\sigma_0 \in \Sigma$ its *initial state*, and

$$\tau \subseteq \Sigma \times \mathbb{S}^m \times \Delta \times \mathbb{S}^n \times \Sigma$$

its *transition relation*. Any $(\sigma, s, \delta, s', \sigma') \in \tau$ is called a *transition*. σ is its *source state*, σ' its *destination state*, s its *input tuple*, s' its *output tuple*, and δ its *delay*. (s, δ, s') are the transition *label*.

Finally, $\succ$ is a non-reflexive, anti-symmetric and transitive partial order relation on τ, called its *priority* relation.

Notation 1. For any transition $(\sigma, s, \delta, s', \sigma') \in \tau$ we also write

$$\sigma \xrightarrow[\delta]{s \mapsto s'} \sigma'$$

or, if δ is understood or not relevant,

$$\sigma \xrightarrow{s \mapsto s'} \sigma'$$

The set of all m-to-n actors with firing is $\mathcal{A}^{m \longrightarrow n}$. The set of all actors is

$$\mathcal{A} =_{def} \bigcup_{m,n \in \mathbf{N}} \mathcal{A}^{m \longrightarrow n}$$

The core of this definition can be found in [10], where "firing rules" defined the input tuples and a firing function mapped those to output tuples. State was added in an extension proposed in [7]. Here, we add the priority relation, which makes actors much more expressive, by allowing them to ascertain and react to the "absence" of tokens. On the other hand, it can make them harder to analyze, and it may introduce unwanted non-determinism into a dataflow model.

priority relation allows test for *absence* of tokens

Intuitively, the priority relation determines that a transition cannot occur if some other transition is possible. We can see this in the definition of a valid *step* of an actor, which is a transition such that two conditions are satisfied: the required input tokens must be present, and there must not be another transition that has priority.

Definition 8 (Enabled transition, step). Given an m-to-n actor $\langle \sigma_0, \Sigma, \tau, \succ \rangle$, a state $\sigma \in \Sigma$ and an *input tuple* $v \in \mathbb{S}^m$, a transition $\sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'$ is *enabled* iff

$$s \sqsubseteq v$$

$$\neg\exists \sigma \xrightarrow{r \mapsto r'} \sigma'' \in \tau : r \sqsubseteq v \wedge \sigma \xrightarrow{r \mapsto r'} \sigma'' \succ \sigma \xrightarrow{s \mapsto s'} \sigma'$$

A *step* from state σ with input v is any enabled transition $\sigma \xrightarrow{s \mapsto s'} \sigma'$. The *residual input tuple* v' is defined by $v = s + v'$.

Note that the second condition for a transition to be enabled becomes vacuously true if $\succ = \emptyset$, leaving $s \sqsubseteq v$, the usual dataflow condition. We call an actor with an empty priority relation a *pure* actor.

pure actor

Part III

Appendices

Appendix A

CAL language syntax

The following is a summary of the CAL language syntax. See chapter 2 for some relevant conventions and the syntax of lexical tokens.

A.1 Actor

Actor → [Imports] **actor** ID
['[' TypePars ']'] '(' ActorPars ')' IOSig [TimeClause] ':'
{ VarDecl | Action | InitializationAction | PriorityOrder }
[ActionSchedule]
{ VarDecl | Action | InitializationAction | PriorityOrder }
(**end**|**endactor**)

TypePar → ID ['<' Type]

ActorPar → [Type] ID ['=' Expression]

IOSig → [PortDecls] '==>' [PortDecls]

PortDecl → [**multi**] [Type] ID

TimeClause → **time** Type

Import → SingleImport | GroupImport ';'

SingleImport → **import** QualID ['=' ID]

GroupImport → **import all** QualID

QualID → ID { . ID }

Type → ID
| ID '[' TypePars ']'
| ID '(' [TypeAttr { ',' TypeAttr }] ')'

| '[' [Types] '-->' Type ']'
| '[' [Types] '-->' ']'

TypeAttr → ID ':' Type
| ID '=' Expression

VarDecl → **[mutable]** [Type] ID [('=' | ':=') Expression]
| FunDecl | ProcDecl

A.2 Expressions

Expression → PrimaryExpression { Operator PrimaryExpression }

PrimaryExpression → [Operator] SingleExpression
{ '(' [Expressions] ')' | '[' Expressions ']' | '.' ID }

SingleExpression → **[old]** ID
| ExpressionLiteral
| '(' Expressions ')'
| IfExpression
| LambdaExpression
| ProcExpression
| LetExpression
| ListComprehension
| SetComprehension
| MapComprehension

ExpressionLiteral → IntegerLiteral | DecimalFractionLiteral
| StringLiteral
| **true** | **false** | **null**

IfExpression → **if** Expression **then** Expression **else** Expression **end**

LetExpression → **let** VarDecls ':' Expression **(end|endlet)**

LambdaExpression → **[const] lambda** '(' [FormalPars] ') ['-->' Type]
[**var** VarDecls] ':' Expression **(end|endlambda)**

FormalPar → [Type] ID

ProcExpression → **proc** '(' [FormalPars] ')' [**var** VarDecls]
(do| begin) { Statement } **(end|endproc)**

FuncDecl → **function** ID '(' [FormalPars] ')' [**var** VarDecls ':'] Expression **end**

ProcDecl → **procedure** ID '(' [FormalPars] ')' [**var** VarDecls (**begin**| **do**)]
{ Statement } **end**

SetComprehension → '{' [Expressions [':' Generators]] '}'

ListComprehension → '[' [Expressions [':' Generators] ['|' Expression]] ']'

MapComprehension → **map** '{' [Mappings [':' Generators]] '}'

Mapping → Expression '->' Expression

Generator → **for** [Type] ID [',' IDs] **in** Expression { ',' Expression }

TypeAssertionExpr → '(' Expression '::' Type ')'

A.3 Statements

Statement → AssignmentStmt
| CallStmt
| BlockStmt
| IfStmt
| WhileStmt
| ForeachStmt
| ChooseStmt

AssignmentStmt → ID [Index | FieldRef] ':=' Expression ';'

Index → '[' [Expressions] ']'

FieldRef → '.' ID

CallStmt → Expression '(' [Expressions] ')';'

BlockStmt → **begin** [**var** VarDecls **do**] { Statement } **end**

IfStmt → **if** Expression **then** { Statement } [**else** { Statement }] **end**

WhileStmt → **while** Expression [**var** VarDecls] **do** [Statements] (**end**|**endwhile**)

ForeachStmt → ForeachGenerator { ',' ForeachGenerator }
    [var VarDecls**] do** [Statements] **(end|endforeach)**

ForeachGenerator → **foreach** [Type] ID { ',' ID } **in** Expression
    [',' Expressions]

ChooseStmt → ChooseGenerator { ',' ChooseGenerator }
    [var VarDecls**] do** [Statements]
    [else [[var VarDecls**] do]** [Statements] **] (end|endchoose)**

ChooseGenerator → **choose** [Type] ID [',' IDs] **in** Expression [',' Expressions]

A.4 Actions

Action → [ActionTag ':'] **action** ActionHead [**do** Statements] **(end|endaction)**

ActionTag → ID { '.' ID }

ActionHead → InputPatterns '==>' OutputExpressions
    [guard Expressions] **[var** VarDecls] **[delay** Expression]

InputPattern → [ID ':'] '[' IDs ']' [RepeatClause] [ChannelSelector]

ChannelSelector → **at** Expression
    | at* Expression
    | [at*] any
    | [at*] all

RepeatClause → **repeat** Expression

OutputExpression → [ID ':'] '[' Expressions ']' [RepeatClause]
    [ChannelSelector]

InitializationAction → [ActionTag ':']
    initialize InitializerHead [**do** Statements] **(end|endinitialize)**

InitializerHead → '==>' OutputExpressions
    [guard Expressions] **[var** VarDecls] **[delay** Expression]

A.5 Action control

ActionSchedule → ScheduleFSM | ScheduleRegExp

ScheduleFSM → **schedule [fsm]** ID ':'
{ StateTransition ';' }
(end|endschedule)

StateTransition → ID '(' ActionTag { '
' ActionTag } ')' '-->' ID
{ '|' '(' ActionTags ')' '-->' ID }

ActionTags → ActionTag
| ActionTag ',' ActionTags

ScheduleRegExp → **schedule regexp** RegExp **(end|endschedule)**

RegExp → ActionTag
| '(' RegExp ')'
| '[' RegExp ']'
| RegExp '*'
| RegExp RegExp
| RegExp '|' RegExp

PriorityOrder → **priority** { PriorityInequality ';' } **end**

PriorityInequality → ActionTag '>' ActionTag { '>' ActionTag }

Appendix B

Keywords

The following table lists the currently active reserved words in CAL. The comment either names the context of its use or provides an example schema of its use. Note that these are *not* complete syntax rules. The references point to further information on the keyword.

Keyword	Comment	Reference
`action`	`action` *actionhead* `do` *statements* `end`	8
`actor`	`actor` *actorhead* `:` *decls, actions, etc.* `end`	3
`all`	input patterns	8.1
	output expressions	8.2
`and`	Boolean operator	C.1, C.2.9
`any`	input patterns	8.1
	output expressions	8.2
`at`	input patterns	8.1
	output expressions	8.2
`at*`	input patterns	8.1
	output expressions	8.2
`begin`	`procedure` *head* `begin` *statements* `end`	6.9.3
	`proc` *head* `begin` *statements* `end`	6.9.2
	`begin` *statements* `end`	7.3
`choose`	`choose` *generators* `do` *statements* `end`	7.7
`const`	`const lambda` *head* `:` *expression* `end`	6.9.1
`delay`	action head	8.3
`div`	arithmetic operator	C.1, C.2.6
`do`	actions	8
	structured statements	7.5, 7.6, 7.7
`dom`	domain operator on maps	C.1, C.2.5

continued on next page

(continued)

Keyword	Comment	Reference
`else`	`if` *expression* `then` *expression* `else` *expression* `end`	6.7
	`if` *expression* `then` *statements* `else` *statements* `end`	7.4
	`choose` *generators* `do` *statements* `else` *statements* `end`	7.7
`end`	generic end marker	
`endaction`	alternative end marker, see `action`	
`endactor`	alternative end marker, see `actor`	
`endchoose`	alternative end marker, see `choose`	
`endforeach`	alternative end marker, see `foreach`	
`endfunction`	alternative end marker, see `function`	
`endif`	alternative end marker, see `if`	
`endinitialize`	alternative end marker, see `initialize`	
`endlambda`	alternative end marker, see `lambda`	
`endlet`	alternative end marker, see `let`	
`endpriority`	alternative end marker, see `priority`	
`endproc`	alternative end marker, see `proc`	
`endprocedure`	alternative end marker, see `procedure`	
`endschedule`	alternative end marker, see `schedule`	
`endwhile`	alternative end marker, see `while`	
`false`	Boolean constant literal	C.2.9
`for`	`for` *vars* `in` *expression*	6.10.2
`foreach`	`foreach` *generators* `do` *statements* `end`	7.6
`fsm`	`schedule fsm` *tag* `:` *transitions* `end`	9.2.1
`function`	`function` *head* `:` *expression* `end`	6.9.3
`guard`	action header	8
`if`	`if` *expression* `then` *expression* `else` *expression* `end`	6.7
	`if` *expression* `then` *statements* `else` *statements* `end`	7.4
`import`	`import` *qualid* [= *var*]	3.1
`in`	membership operator, collections	C.1, C.2.1
	generators	6.10.2, 7.6, 7.7
`initialize`	`initialize` *head* `do` *statements* `end`	8.5
`lambda`	`lambda` *head* `:` *expression* `end`	6.9.1
`let`	`let` *vardecls* `:` *expression* `end`	6.8
`map`	`map {` *expressions* [`:` *generators*] `}`	6.10
`mod`	modulo operator	C.1, C.2.6

continued on next page

(continued)

Keyword	Comment	Reference
`multi`	multi-port declaration	3
`mutable`	`mutable` *vardecl*	5.1.1
`not`	Boolean operator	C.1, C.2.9
`null`	literal	2.1
`old`	`old` *var*	6.2.1
`or`	Boolean operator	C.1, C.2.9
`priority`	`priority` *inequalities* `end`	9.3
`proc`	`proc` *head* do *statements* `end`	6.9.2
`procedure`	`procedure` *head* do *statements* `end`	6.9.3
`regexp`	`schedule regexp` *regexp* `end`	9.2.2
`repeat`	input patterns	8.1
	output expressions	8.2
`rng`	range operator on maps	C.1, C.2.5
`schedule`	`schedule` *schedule* `end`	9.2
`then`	`if` *expression* `then` *expression* `else` *expression* `end`	6.7
	`if` *expression* `then` *statements* `else` *statements* `end`	7.4
`time`	actor head	3
`true`	Boolean constant literal	C.2.9
`var`	variable declarations marker	
`while`	`while` *expression* do *statements* `end`	7.5

The following keywords are reserved for future use.

`assign` `case` `default` `endinvariant` `endtask`
`endtype` `ensure` `invariant` `now` `out`
`protocol` `require` `task` `type`

Appendix C

Basic runtime infrastructure

This appendix describes the basic runtime infrastructure, i.e. the kinds of objects and operations on them that implementations must provide in order to implement CAL. A realistic implementation might support many more data types and have a much more extensive library of functions and procedures. Most of the facilities described in this appendix are used in CAL constructs, such as collections in generators, or maps and sets in input patterns and output expressions.

C.1 Operator symbols

The following table summarizes the predefined unary operator symbols in CAL.

Operator	Operand type	Meaning
`not`	`Boolean`	logical negation
`#`	`Collection[T]`	number of elements
	`Map[K, V]`	number of mappings
`dom`	`Map[K, V]`	domain of a map
`rng`	`Map[K, V]`	range of a map
`-`	`Number`	arithmetic negation

The next table lists the predefined binary operator symbols in the CAL language. They are sorted by increasing binding strength. Their binding strength is given by a precedence figure P, higher precedence binds stronger.

P	Operator	Operand 1	Operand 2	Meaning
1	`and`	`Boolean`	`Boolean`	logical conjunction
	`or`	`Boolean`	`Boolean`	logical disjunction
2	`=`	`Object`	`Object`	equality
	`!=`	`Object`	`Object`	inequality
	`<`	`Number`	`Number`	less than
		`Set[T]`	`Set[T]`	
		`String`	`String`	
		`Character`	`Character`	
	`<=`	*analogous to <*		less then or equal
	`>`	*analogous to <*		greater than
	`>=`	*analogous to <*		greater than or equal
3	`in`	`T`	`Collection[T]`	membership
4	`+`	`Number`	`Number`	addition
		`Set[T]`	`Set[T]`	union
		`List[T]`	`List[T]`	concatenation
		`Map[K, V]`	`Map[K, V]`	map union
	`-`	`Number`	`Number`	difference
		`Set[T]`	`Set[T]`	set difference
5	`div`	`Number`	`Number`	integral division
	`mod`	`Number`	`Number`	modulo
	`*`	`Number`	`Number`	multiplication
		`Set[T]`	`Set`	intersection
	`/`	`Number`	`Number`	division
6	`^`	`Number`	`Number`	exponentiation

C.2 Basic data types and their operations

This section shortly lists the basic data types used by CAL, describes their fundamental properties, and the operations they can be expected to support.

selector function

Many of the basic data types are *composite*, i.e. elements of these types may be objects that contain other objects. In the cases presented in this section, such a composite can be recursively decomposed into a subobject and a residual composite, until a canonical *empty* composite is reached. Functions operating on such composites can often easily be described by using *selector functions*. A selector function takes a composite and another function as arguments. It picks one of the subobjects and passes it and the residual composite into the argument function. In the cases below, we can describe our composite data types by some properties of the data objects and the behavior of the selector function.

C.2.1 `Collection[T]`—collections

An object of type `Collection[T]` is a finite collection of objects of type `T`. The most fundamental operation on it is *selecting* an *element*, which is really dividing the collection into an element and a residual collection of the same kind. The `selectf` selector function does this, and has the following signature:

`selectf`

```
selectf<A, B>: [[A, Collection[A] --> B],
                Collection[A],
                B --> B]
```

Different kinds of collections may differ in the way the selector function picks the element from the collection (e.g., in the case of lists, it is guaranteed to pick the first element of the list, see C.2.4).

size and membership

The two fundamental operations on a collection are computing its *size* and testing whether a given object is contained by it. These are represented by two operators, the prefix operator `#` and the infix operator `in`. We assume that these are substituted by function calls as follows:

`# C` ≡ `$size(C)`
`a in C` ≡ `$member(a, C)`

We can define the `$size` and `$member` functions using the `selectf` function in the following manner:

```
function $size(C) : // prefix #
    selectf(lambda (a, C1) :
              1 + $size(C1)
            end, C, 0)
end

function $member(x, C) :
    selectf(lambda (a, C1) :
                if a = x then true else
                    $member(x, C1)
                end
            end, C, false)
end
```

Finally, we can define iteration over a collection by writing a procedure that works somewhat analogous to the `selectf` function:

```
procedure selectp(p, C) begin
    selectf(lambda (a, C1) :
                proc () do p(a, C1); end
            end,
            C,
            proc () do end)();
    end
end
```

Consequently, the signature of `selectp` is as follows:

```
selectp<A>: [[A, Collection[A] -->], Collection[A]]
```

Generator functions and procedures

The selection function and procedure also allow us to construct the various functions and procedures used in defining generators in CAL. For instance, the $mapadd function used in comprehensions (see section 6.10.2), which is slightly different depending on whether it produces a set, a list, or a map:

```
function $mapaddset (C, f) :
    selectf(lambda (a, C1) : f(a) + $mapaddset(C1, f) end,
            C, {})
end

function $mapaddlist (C, f) :
    selectf(lambda (a, C1) : f(a) + $mapaddlist(C1, f) end,
            C, [])
end

function $mapaddmap (M, f) :
    selectf(lambda (a, M1) : f(a) + $mapaddmap(M1, f) end,
            M, map {})
end
```

The `$iterate` procedure used in the foreach-statement (see section 7.6) can be built on top of `selectp` as follows:

```
procedure $iterate (C, p) begin
    selectp(proc (a, C1) do p(a); $iterate(C1, p); end, C);
end
```

Finally, the `$try` procedure used in the choose-statement (see section 7.7) can be constructed as follows:

```
procedure $try (C, f, p) begin
    selectp(proc (a, C1) do
                if f() then
                    p(a);
                    $try(C1, f, p);
                end
            end, C);
end
```

C.2.2 `Seq[T]`—sequences

Sequences are arrangements of objects that are indexed by non-negative integers, starting at zero if not empty, and either ending at some maximal index *k*, or not at all. The first kind of sequence is called *finite sequence* or *list* (see section C.2.4), the second kind is called *infinite sequence*.

The selector function of sequences splits the non-empty sequence into its first element and the rest of the sequence:

```
selectf<A, B>: [[A, Seq[A] --> B],
                Seq[A],
                B --> B]
```

`hasElement`

The only operations on sequences are indexing (using a single integral index) and testing whether a given non-negative integer is a valid index. They can be expressed in terms of the selector as follows:

```
function $nth(n, S) : // indexer
    selectf(lambda (a, R) :
                if n = 0 then a else
                    $nth(n - 1, R)
                end, S, null)
end

function hasElement (n, S) :
    selectf(lambda (a, R) :
                if n = 0 then true else
                    hasElement(n - 1, R)
                end, S, false)
end
```

C.2.3 `Set[T] < Collection[T]`—sets

nondeterministic `selectf`

Sets are a special kind of collection (see section C.2.1) that guarantee that each object occurs in them at most once. The selection function `selectf` is *nondeterministic* on sets with two or more elements, i.e. it is unspecified which element will be chosen.

`subset`: partial order

Sets are partially ordered by the *subset* order, which can be defined as follows:

```
function subset(S1, S2) :
    selectf(lambda (a, S) :
                if a in S2 then
                    subset(S, S2)
                else
                    false
                end
```

```
            end, S1, true)
end
```

set equality

The comparison operators (<, <=, >, >=) are based on the subset order. In addition, two sets are considered *equal* if they are subsets of each other, i.e. they contain the same elements.

`addElement`

The fundamental way of constructing sets is to add one element to an existing set. This can be done using the function `addElement`, which has the following signature:

```
  addElement<T> : [T, Set[T] --> Set[T]]
```

The result of `addElement(a, S)` is the smallest set that contains both, a and all the elements in `S`.

Sets also have operators for union (+), intersection (`*`) and set difference (`-`), which can be defined as follows:

```
function $union (S1, S2) : // infix +
    selectf(lambda(a, S) :
                addElement(a, $union(S, S2))
            end, S1, {})
end

function $intersection(S1, S2) : // infix *
    selectf(lambda(a, S) :
                if a in S2 then
                    addElement(a, $intersection(S, S2))
                else
                    $intersection(S, S2)
                end
            end, S1, {})
end

function $setDifference(S1, S2) : // infix -
    selectf(lambda(a, S) :
                if a in S2 then
                    $setDifference(S, S2)
                else
                    addElement(a, $setDifference(S, S2))
                end
            end, S1, {})
end
```

In addition to these basic operations, there are set comprehensions for computing sets (section 6.10).

C.2.4 `List[T] < Collection[T], Seq[T]`—lists

Lists are finite, sequential arrangements of objects. They are collections and sequences at the same time. As a consequence, its selector function splits the (non-empty) list into its first element and the rest of the list.

For any list, the value computed by the `$size` function (section C.2.1) defines the valid indices (via `$nth` and `hasElement`, see section C.2.2) into the list, which are the non-negative integers smaller than that number.

cons constructor

Similar to sets, lists can be constructed elementwise using a cons function, that adds an element to the front of an existing list:

```
cons<T> : [T, List[T] --> List[T]]
```

Lists can be concatenated using the + operator, which can be defined as follows:

```
function $add (L1, L2) : // operator +
    selectf(lambda (a, L) :
                cons(a, $add(L, L2))
            end, L1, L2)
end
```

Similarly to sets, list can be created using list comprehensions (section 6.10).

C.2.5 `Map[K, V]`—maps

A map of type `Map[K, V]` maps finitely many *keys* of type `K` to *values* of type `V`. The set of keys of a map is called its *domain*, while the set of values is called its *range*.

Like collections, maps are accessed using a selector function. In the case of maps, this has the following signature:

```
selectf<K, V>: [[K, V, Map[K, V] --> A],
                Map[K, V],
                A --> A]
```

It separates a non-empty map into a key, the corresponding value, and the residual map, which it then applies its first argument to.

The domain and range of a map can then be defined like this:

```
function $domain(M) : // operator dom
    selectf(lambda (k, v, M1) :
                addElement(k, $domain(M1))
            end, M, {})
end

function $range(M) : // operator rng
    selectf(lambda (k, v, M1) :
                addElement(v, $range(M1))
            end, M, {})
end
```

The value corresponding to a key in a map is retrieved via an indexer, using the key as in index into the map. The indexing function is defined as follows:

```
function $get(x, M) // indexer
    selectf(lambda(k, v, M1) :
                if x = k then v else
                    $get(x, M1)
                end
            end, M, null)
end
```

C.2.6 `Number`—numbers

Most of the number system of CAL is left to platform definition. The language itself provides two kinds of numeric literals (integers are decimal fractions). There are a number of operators on numbers, cf. section C.1 for details.

The type `Number` contains the subtype `Integer` of integer numbers.

C.2.7 `Character`—characters

Characters are usually related to other data types, such as numbers (via some encoding) or strings (as components of strings). The details of this are left to the definition of platforms.

Characters are (possibly partially) ordered, and thus support the relational operators <, <=, >, >=.

C.2.8 `String < List[Character]`—strings

Strings are lists of characters, supported by a special syntax for their construction. It is left to an implementation whether they are identical to a corresponding list of characters, i.e. whether

`"abc" = ['a', 'b', 'c']`

Strings 'inherit' the order from characters, and expand it lexicographically:

```
function $stringLE (s1, s2) : // operator <=
    selectf(lambda (c1, r1) :
                selectf(lambda (c2, r2) :
                            if c1 = c2 then
                                $stringLE(r1, r2)
                            else
                                c1 < c2
                            end
                        end, s2, false)
```

```
              end, s1, true)
end
```

C.2.9 `Boolean`—truth values

The `Boolean` data type represents truth values. It is supported by two constant literals (**true** and **false**) as well as a number of operators, cf. section C.1.

C.2.10 `Null`—the null value

The `Null` data type has only one representative, the object denoted by the literal **null**. Apart from equality tests, no operations are defined on this type.

Acknowledgments

Many people have contributed to this work in various ways, and the authors would like to acknowledge their contributions and thank them for their input and support.

Special thanks go to Edward A. Lee, Christopher Hylands, H. John Reekie, Anders Blomdell, Lars Wernli, Chris Chang, Yang Zhao, Ernesto Wandeler, Ed Willink, Steven Neuendorffer, Kees Vissers, C. Xavier Brooks, and the Ptolemy Group at UC Berkeley.

The research reported here is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of California MICRO program, and the following companies: Agilent Technologies, Cadence Design Systems, Hitachi, and Philips.

Part of this research was supported by STINT—The Swedish Foundation for International Cooperation in Research and Higher Education.

Addresses & contact

Johan Eker	`johan.eker@ericsson.com`
Jörn W. Janneck	`jwj@acm.org`
The Ptolemy Project	`http://ptolemy.eecs.berkeley.edu`
CAL Development	`http://www.caltrop.org`

Index

- `(...)`
  - in action schedule, 79, 80
  - in expression, 41
- `*`
  - in action schedule, 80
- `-->`
  - in lambda-expression, 47
- `->`, 49, 50
- `.`
  - in action tag, 65
  - in assignments, 55
  - in expression, 41
  - in field selection expression, 45
- `:`
  - after action tag, 65
  - in action schedule, 79
  - in actor head, 21
  - in initialization action, 74
  - in input pattern, 66
  - in lambda-expression, 47
  - in output expression, 70
- `::`, 53
- `:=`
  - in assignments, 55
  - in variable declarations, 34
- `;`
  - in action schedule, 79
  - in assignments, 55
  - in priority clause, 82
  - in procedure call, 58
- `<`
  - in type parameter constraint, 21
- `=`
  - in actor parameters, 21
  - in variable declarations, 34
- `==>`
  - in action head, 65
  - in initialization action, 74
  - in port declaration, 21
- `>`
  - in priority clause, 82
- `[...]`, 49, 50
  - in action schedule, 80
  - in assignments, 55
  - in expression, 41
  - in indexing expression, 45
  - in input pattern, 66
  - in output expression, 70
- `#`
  - on collection, 105
- `$`
  - in reserved identifiers, 17
- `$iterate`
  - definition, 106
  - use in foreach-statement, 61
- `$mapadd`
  - definition, 106
  - use in generators, 51
- `$try`
  - definition, 106
  - use in choose-statement, 62
- `{...}`, 49, 50
- `|`
  - in action schedule, 79, 80

- action, 21, 65–75
  - activated, 73
  - activation conditions, 73
  - and transition, 65
  - body, 65

  choice, 65
  control structures, 77–83
  delay, 65, 73
  duration, 23
  firability, 66
  head, 65
  initialization action, 74–75
    activation conditions, 74
    multiple activated, 74
  legal sequence of, 79
  priority, 82–83
    order construction, 82
    partial order, 82
  schedule, 78–82
    finite state machine, 79–80
    regular expression, 80–82
  scope, 69–70
  selection, 73–74, 77–83
  tag, 65, 78
    denoted actions, 78
    extension, 78
    language, 79, 80
    prefix order, 78
  untagged
    and schedules, 79
  variable
    scope, 69–70
action, 65
action schedule, 21, 78
activation conditions
  of action, 73
  of initialization action, 74
actor, 86–90
  firing, 9
  initial state, 89
  parameter, 21, 22, 33
  port signature, 21, 22
  priority relation, 89
  pure, 90
  state, 9, 89
    change, 65
  structure of, 21
  transition, 89
    relation to action, 65
  transition relation, 89
  type parameter, 21
    bound, 21
actor, 21
actor model, 86–90
actor transition system, 89
actors
  composition of, 9
  scheduling of, 10
aliasing, 57
all, 66, 68, 70
any, 66, 68, 70
application expression, 43–44, 47
art of cheating, 44
assignability, *see* types, assignability
assignment, *see* variable, 55–58
  field, 55–56
  indexed, 55–56
  simple, 56
associativity
  of operators, 45
at, 66, 68, 70
at*, 66, 68, 70
ATS, *see* actor transition system

backtracking
  and choose-statement, 63
Backus-Naur-Form, 19
begin, 47
binary operator
  predefined, 103
binding, 33
  assignable, 33
  mutable, 33
BNF, 19
body
  of expression, 20
  of statement, 20
`Boolean` (type), 28, 42, 46, 50, 59, 62, 111

cast, *see* type assertion
channel, 22
channel identifier, 67
channel selector, 66, 68
  free variables of, 70
  in output expression, 71
  multichannel, 68, 72

    output expression
        constraints on map domains, 72
    single-channel, 68, 71
`ChannelID` (type), 28
`Character` (type), 110
choose, 61
choose-statement, 61–63
    backtracking, 63
    filter, 61
    semantics, 61
cloning, 57
    lazy, 57
closure, 46–48
    and old variables, 43
    function, 44, 46–47
        application, 47
        application of, 46
        const, 47
        invariant, 47
        type, 47
    parameter, 33
    procedure, 46–48
        call of, 46
    relation to variable dependency, 35
`Collection` (type), 28, 50, 105–106
    membership test, 105
    selector function, 105
    size, 105
    vs collection, 48
collection
    expression, 49
    proper, 48
    vs `Collection`, 48
comment, 18
    syntax, 18
complete partial order
    of sequences, 86
composite object, 29, 45
comprehension, 48–52
    efficiency, 52
    filter, 50
    generator, 48, 50
        element order, 50
        semantics, 51
        syntax, 50
    implementation, 52
    list, 49, 50
        tail-free, 51
    map, 49, 50
    set, 49, 50
    simple, 49
    with generators, 50–52
condition
    for firing, 65
conditional expression, 46
const, 47
control flow, 59, 60
control structures
    action-level, 77–83
corollary
    mutual dependencies, 36
cpo, *see* complete partial order

dataflow, 10
declaration
    of function, 34, 48
    of procedure, 34, 48
declaration set
    well-formed, 36
definition
    actor, 89
    actor transition system, 89
    ATS, 89
    dependency relation, 37
    dependency set, 36
    enabled transition, 90
    immediate dependency set, 36
    legal action sequence, 79
    step (of actor), 90
    temporal metric, 87
    time system, 87
    well-formed declaration set, 36
delay, 23–24, 65
    in time system, 23
        zero, 23
    of action, 73
    time, 87
delay, 65, 73, 74
delimiter, 17
dependency
    circular, 35
dependency relation, 37

  - is partial order, 37
- dependency set, 35–39
  - *definition*, 36
  - example, 37
  - immediate, 36
- **do**, 47, 59–61, 65, 74
- domain
  - of map, 109
    - definition, 109–110
- duration
  - of action firing, 23

- eager evaluation, 44
- **else**, 46, 59, 61
  - in foreach-statement, 61
- **end**, 21, 46–48, 59–61, 65, 74, 79, 80, 82
  - alternatives, 20
    - exception, 59
- end marker, 20
- **endaction**, 65
- **endactor**, 21
- **endchoose**, 61
- **endforeach**, 60
- **endfunction**, 48
- **endif**, 46, 59
- **endinitialize**, 74
- **endlambda**, 47
- **endlet**, 46
- **endpriority**, 82
- **endproc**, 47
- **endprocedure**, 48
- **endschedule**, 79, 80
- **endwhile**, 59
- environment, 46
  - global, 21, 22
- example
  - **old** reference, 42
  - action variable scope, 70
  - actor
    - Basic FairMerge, 77
    - BiasedMerge, 83
    - FairMerge with priorities, 83
    - FairMerge with regexp schedule, 81
    - Nondeterministic Merge, 77
    - Symmetric FairMerge, 82
  - backtracking using choose-statement, 63
  - collections with generators, 50
  - FairMerge, with FSM schedule, 80
  - fields and indices, 29
  - foreach-statement, 60
  - `$mapadd` function, 52
  - multichannel input patterns, 69
  - mutually recursive variable declarations, 37–38
  - non-numeric indices, 29
  - non-well-formed variable declarations, 38–39
  - operator associativity and precedence, 45
  - order in generator collections, 51
  - simple map expressions, 49
  - simple set and list expressions, 49
  - single-channel output expression, 71
  - single-port input pattern, 67
  - single-port input pattern with repeat-clause, 68
  - time systems, 88
- expression, 41–53
  - application, 46, 47
  - body, 20
  - closure, 46–48
    - **const** function, 47
    - function, 46–47
    - invariant function, 47
    - procedure, 46–48
  - comprehension, 48–52
    - filter, 50
    - generator, 50
    - generator semantics, 51
    - list, 49, 50
    - map, 49, 50
    - set, 49, 50
    - simple, 49
    - with generators, 50–52
  - conditional, 46
  - constant, 42
  - field selector, 45
  - function
    - application, 43–44, 47

  head, 20
  indexing, 45
  lambda, 44
  lambda, 47
    const, 47
  list comprehension
    tail-free, 51
  literal, 42
    decimal fraction, 42
    integer, 42
    string, 42
    syntax, 42
  local scope, 46
  operator, 45–46
  primary, 41
  proc, 47–48
  selector, 45
  side-effect-free, 41, 47
  syntax, 41
  type assertion, 53
  type of, 41
  variable, 33–39, 42–43
extension
  of action tag, 78

false, 28, 42, 62, 111
field, 29
  selector expression, 45
field reference
  in assignments, 55
filter, 50
firing
  duration, 23
firing condition, 65
firing rules (from [10]), 89
for, 49, 50
foreach, 60
  statement, 55
foreach-statement, 60–61
  semantics, 60
free variable, 35, 38
fsm, 79
function, 46–47
  application, 43–44, 46, 47
  const, 47
  declaration, 34, 48
  invariant, 47
  recursive, 35
  type, *see* types, closure, 47
function, 48

generator, 48, 50–52
  collection
    order, 50
  filter, 50
  in foreach-statement, 60
  semantics, 51
  syntax, 50
  variable, 50
    scoping, 50
generic type, *see* types, parametric
group import, 23
guard, 65, 74

`hasElement`
  definition, 107
head
  of expression, 20
  of statement, 20
homogeneous use
  of input pattern, 69
  of output pattern, 72
host environment, 18
host language, 18

identifier, 17
  escape syntax, 17
  escaped, rationale, *see* rationale, escaped identifier
  qualified, 22, 78
  reserved, 17
if, 46, 59
  statement, 55
if-statement, 59
implementation note
  actor parameter, 22
  cloning, 57
  generator implementations, 52
  lazy evaluation, 44
  object types, 26
  output token activation condition, 74

- **import**, 22–23
- importing
  - group import, 23
  - of binding, 22
  - single import, 23
- **in**, 49, 50, 60, 61
- `in` (operator)
  - on collection, 105
- index, 29
  - in assignments, 55
- indexing expression, 45
- initial state
  - of actor, 89
- initialization action, 74–75
  - activation conditions, 74
  - multiple activated, 74
- initialization expression, 34
  - of a variable, 35
- initialization rule, 21
- **initialize**, 74
- input pattern, 66–70
  - association with port, 66
  - channel selector, 66
  - homogeneous use, 69
  - implicit variable declaration, 66
  - multichannel, 67
  - multiport, 68–69
  - single-channel, 67
  - single-port, 67–68
  - variable
    - dependencies, 70
    - type, table, Fig. 8.1, 67
- input token, 65
- `Integer` (type), 29, 110

- key
  - of map, 109
- keyword construct, 19
- keywords, 17
- Kleene operator
  - in action schedule, 80

- labeled transition system, 89
- **lambda**, 47
- lambda closure
  - type, *see* types, closure
- lambda expression, 44, 47
  - **const**, 47
- language
  - core, 11
  - domain-specific, 11
  - usability, 11
- language design
  - goals, 10–12
- lazy evaluation, 44
- **let**, 34, 46
- lexical scoping, 35
- lexical token, 17
- lexicographical order, 110
- `List` (type), 28, 109
  - concatenation, 109
  - `cons`, 109
  - constructor, 109
  - `hasElement`, 109
  - selector function, 109
  - size, 109
  - valid indices, 109
- list
  - comprehension, 48–50
    - simple, 49
    - tail-free, 51
  - construction, 48
- literal, 42
  - decimal fraction, 42
  - integer, 42
  - numeric, 18
  - string, 42
- local name, 22
- location, 29–30, 45
  - field, 29
  - independence of, 30
  - index, 29
  - orthogonality, 30
- LTS, *see* labeled transition system

- `Map` (type), 28, 69, 72, 109–110
  - domain, 109
    - definition, 109–110
  - indexer, 110
  - key, 109
  - range, 109
    - definition, 109–110

  - selector function, 109
  - value, 109
- map
  - comprehension, 48–50
    - simple, 49
  - construction, 48
    - nondeterminism, 49
  - key, 49
  - value, 49
- **map**, 49, 50
- membership test
  - in collection, 105
- metric
  - of time system, *see* temporal metric
- metric time system, 87
- model, 9
  - communication, 10
- model of computation, 10
- **multi**, 21
- multichannel channel selector, 68, 72
- multichannel input pattern, 67
- multichannel output expression, 72
- multiport, *see* port, multiport, 67
  - and input pattern, 66
- multiport input pattern, 68–69
- **mutable**, 34
- mutation, 56

- $\mathbb{N}$ (natural numbers), 86
- namespace, 22–23
  - hierarchical, 21, 22
  - local name, 22
  - subnamespace, 22
- nondeterminism
  - choose-statement, 61–63
- notation
  - $\sigma \xrightarrow[\delta]{s \mapsto s'} \sigma'$, 89
- notational idiom, 19
- `Null` (type), 28, 42, 111
- **null**, 28, 42
- `Number` (type), 110
- numeric literal, *see* literal, numeric

- object
  - mutable, 30
  - structured, 29–30
- object modification, *see* types, mutable
- object mutation, *see* types, mutable
- object types, *see* types, object
- **old**, 41–43
  - and closures, 43
  - meaning, 43
  - restrictions, 43
  - translation, 43
  - use in action, 42
- operator, 17, 45–46
  - associativity, 45
  - basic, 103–104
    - binary, 103
    - unary, 103
  - binary, 45
  - binding strength, 103
  - infix, 45
  - precedence, 45, 103
  - predefined, 103–104
    - binary, 103
    - unary, 103
  - prefix, 45
  - symbols, 103
  - unary, 45
- operators
  - represented by functions, 45
- order
  - lexicographical, 110
  - of time tags, 23
- output expression, 70–73
  - association with port, 66
  - multichannel, 72
  - single-channel, 71
  - token expression
    - type, table (Fig. 8.2), 71
- output pattern
  - homogeneous use, 72
- output token, 65

- package, *see* subnamespace
- parameter
  - actor, 33
  - closure, 33
  - generic type, 21
    - bound, 21

  - of actor, 21, 22
- partial order
  - dependency relation is, 37
- pattern variable
  - dependencies, 70
  - scoping, 69–70
  - type
    - table, Fig. 8.1, 67
- platform
  - profile, 13
- platform independence, 12–13
- port
  - and input patterns, 66
  - and output expression, 66
  - input, 9
  - multiport, 22, 67
    - and input pattern, 66
  - output, 9
  - single, 22
  - single port, 67
    - and input pattern, 66
- port signature, *see* actor, port signature
- portability, *see* platform independence
  - language, 13
  - source code, 13
  - target code, 13
- precedence
  - of operators, 45
- prefix
  - of input, 9
- prefix order, 86
  - is complete partial order, 86
- prefix-monotonicity, 82
- primary expression, 41
- priority
  - of action, 82–83
    - order construction, 82
    - partial order, 82
  - order, 82
    - construction, 82
    - validity, 82
- **priority**, 82
- priority block, 21
- priority relation
  - of actor, 89
- **proc**, 47
- proc expression, 47–48
- procedural closure
  - type, *see* types, closure
- procedure, 46–48
  - call, 46, 58–59
  - declaration, 34, 48
- **procedure**, 48
- profile, *see* platform, profile
- Ptolemy, 9, 10
- pure actor, 90

- qualified identifier, *see* identifier, qualified, 78

- range
  - of map, 109
    - definition, 109–110
- rationale
  - choose-statement, 62
  - escaped identifier, 18
  - literals, 42
  - `$mapadd` function, 52
  - optional types, 25
  - output expression design, 72
  - redundant action schedule syntax, 81
  - variable declaration
    - well-formedness, 37
- recursion, *see* recursive closure
- recursive closure, 35
- recursive function, 35
- regexp, *see* regular expression
- **regexp**, 80
- regular expression
  - in action schedule, 80
- **repeat**, 66, 70
- repeat-clause, 71, 72
  - expression
    - free variables of, 70
  - in input pattern, 67
  - in output expression, 71

- $\mathbb{S}$ (finite sequences)
  - $= \mathcal{U}^*$, 86
- $\mathbb{S}_\infty$ (sequences)
  - $\mathbb{S} \cup \mathbb{S}^{\mathbb{N}}$, 86

    complete partial order, 86
    prefix order, 86
schedule
    of actions, 79–82
        finite state machine, 79–80
        regular expression, 80–82
schedule, 79, 80
scope
    of variable, *see* variable, scoping
scoping
    lexical, 35
`selectf`
    for collection, 105
    for list, 109
    for map, 109
    for sequence, 107
    for set, 107
selector
    expression, 45
selector function, 105
    of collection, 105
    of list, 109
    of map, 109
    of sequence, 107
    of set, 107
selector procedure
    of collection, 105
`selectp`
    definition, 105
`Seq` (type), 28, 107
    finite, *see* `List` (type)
    `hasElement`
        definition, 107
    index into, 107
    indexer
        definition, 107
    infinite, 107
    selector function, 107
sequence
    is complete partial order, 86
    prefix order, 86
`Set` (type), 28, 107–108
    `addElement`, 108
    comparison operators, 108
    construction, 108
    difference, 108
    equality, 108
    intersection, 108
    selector function, 107
    subset order, 107
    union, 108
set
    comprehension, 48–50
        simple, 49
    construction, 48
side-effect free, 47
single import, 23
single port, 67
    and input pattern, 66
single-channel channel selector, 68, 71
single-channel input pattern, 67
single-channel output expression, 71
single-port input pattern, 67–68
size
    of collection, 105
state
    of actor, 89
        change, 65
statement, 55–63
    assignment, 55–58
        field, 55–56
        indexed, 55–56
        simple, 56
    begin-end, 59
    block, 55, 59
    body, 20
    call, 46, 58–59
    choice, 55
    choose, 61–63
        backtracking, 63
        semantics, 61
    foreach, 55, 60–61
        generator, 60
        semantics, 60
    head, 20
    if, 55, 59
    list of, 47
    procedure call, 55, 58–59
    while, 55, 59–60
`String` (type), 29, 110–111
    order, 110
subnamespace, 22

subset order, 107
substitutability, *see* types, substitutable
subtype relation, 26, 31

tag
  in time system, 23
    order, 23
  of action, 78
    denoted actions, 78
    extension, 78
    language, 79, 80
    prefix order, 78
  time, 87
temporal distance, *see* temporal metric
temporal metric, 87
  definition, 87
  trivial, 88
  vs proper metric, 87
then, 46, 59
time, 23–24
  adding delays, 23
  addition, 87
  delay, 23, 87
    zero, 23
  order, 87
  tag, 23, 87
    order, 23
  zero delay, 87
time, 21, 24
time system, 23–24, 87–88
  definition, 87
  metric, 87
token, 9, 86
  input, 65
  lexical, *see* lexical token
  output, 65
  test for absence of, 82, 89
token expression
  table (Fig. 8.2), 71
  type of, 71
transition
  and action, 65
  of actor, 89
transition relation
  of actor, 89
transition system, *see* actor transition system
true, 28, 42, 46, 50, 59, 111
type assertion, 53
type bound, *see* actor, type parameter, bound
type cast, *see* type assertion
type constructor, 27
type framework, 31
  finite cover, 31
  least upper bound, 31
  subtype order, 31
type inference, 53
type parameter, 27
  value, 27
types, 25–31
  assignability, 26, 31
  closure, 27
    function, 47
  free, 30
  function, 47
  mutable, 30, 33, 56
  object, 25–27
  of token, 67
  optional, 25
  parametric, 27
  required, 28
  simple, 27
  structured, 29–30, 45, 56
  substitutable, 26
  subtypes, *see* subtype relation
  syntax, 27–28
  variable, 25–27

$\mathcal{U}$ (the universe of values), 86
unary operator
  predefined, 103
universe
  of token values, 86

value
  of map, 109
var, 34, 47, 48, 59–61, 65, 74
variable, 33–39, *see* assignment
  assignable, 56
  binding, 33

        assignable, 33
        mutable, 33
    bound, 33
    declaration, 33
        actor parameter, 33
        closure parameter, 33
        explicit, 33–34
        in function, 47
        in procedure, 47
        input pattern, 33
        not well-formed, 35
    declaration set
        well-formed, 36
    dependency, 35
        circular, 35
    dependency relation, 37
    dependency set, 35–39
        definition, 36
        example, 37
        immediate, 36
    free, 22, 35, 38
    generator, 50
    initialization expression, 34, 35
    mutable, 30
    name, 34
    old, 42–43
        and closures, 43
        meaning, 43
        restrictions, 43
        translation, 43
        use in action, 42
    reference, 42–43
    scoping, 35–39
        in actions, 69–70
        lexical, 35
    shadowing, 35
    stateful, 34, 47
    stateless, 34
    type, 34
variable type, *see* types, variable

while, 59
    statement, 55
while-statement, 59–60

zero delay, 87

    in time system, 23

Bibliography

[1] The Ptolemy Project. Department EECS, University of California at Berkeley (*http : //ptolemy.eecs.berkeley.edu*). 9

[2] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1999. 7

[3] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, 1986. 9

[4] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1993. 9

[5] J. B. Dennis. First version data flow procedure language. Technical Memo MAC TM 61, MIT Lab. Comp. Sci., May 1975. 9

[6] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323–363, June 1977. 9

[7] Jörn W. Janneck. *Syntax and Semantics of Graphs—An approach to the specification of visual notations for discrete event systems*. PhD thesis, ETH Zurich, Computer Engineering and Networks Laboratory, July 2000. 9, 87, 89

[8] Jörn W. Janneck. Actors and their composition. Technical Report UCB/ERL 02/37, University of California at Berkeley, 2002. 10

[9] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*. North-Holland Publishing Co., 1974. 9

[10] Edward A. Lee. A denotational semantics for dataflow with firing. Technical Report UCB/ERL M97/3, EECS, University of California at Berkeley, January 1997. 9, 82, 87, 89, 119

[11] Edward A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002. to appear. 10

[12] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A denotational framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998. 10, 87

[13] Charles S. Pierce. How to make our ideas clear. In P. P. Wiener, editor, *Values in a Universe of Chance*. 1878. 7