CAL

Cal Actor Language

Language report

DRAFT

Johan Eker and Jörn W. Janneck

Ptolemy Technical Memo

EECS Department
University of California at Berkeley
May 6, 2002

Contents

Part I

Introduction

Chapter 1

Introduction

This report introduces 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. This is followed by a short outline of the actor model that we the programs 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. In the context of this work, 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 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 networks are of course key to the construction of complex systems, but we will not discuss this subject here, except for the following observations:

composition of actors

- 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

decoupling from communication model

[1] The notion of actor and firing is based on the one presented in [4], extended by a notion of state in [3].

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.

- 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.

decoupling from actor scheduling

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. [5, 6].

communication + scheduling = model of computation

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 meaningful 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 code language. This led to the design of CALCORE (cf also part **??**), and much of the implementation of CAL is based on this concept (cf. [2]).

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. sections 3 and C 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 environment. 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

language portability

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.

CAL's notion of 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.

platform profiles

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.

targeting existing virtual platforms

1.4 Outline

The remainder of this document is structured as follows. In part II, we will describe the CAL language, i.e. its syntax and whatever semantics is associated with it. Part **??** describes CALCORE, the core language that CAL is built on. Since it is a subset of CAL, its description is already included in part II, but this part will define the subset, and outline the transformations needed to turn a program in the full language into CAL-CORE. The appendices provide additional information on the standard environment of CAL, which is its Ptolemy platform binding.

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 CONSIDERATIONS.
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 II

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. Appendix A lists the keywords of CAL and their use.

Identifiers. Keywords, strings of digits etc. may be escaped by backslashes to form identifiers. any string of characters !, @, #, $, %, ^, &, *, /, +, -, =, <, >, ?, ∼, |.

Delimiters. (,), {, }, [,].

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

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.

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`. Semantical entities, such as types, are set *italic*.

2.3 Conventions

We use a form of BNF to describe the syntax rules. Literal elements are put on 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.

Chapter 3

Data types

The actor language itself is almost totally agnostic with respect to the type system used—the type system is considered part of the external environment that we try to keep CAL actor specifications orthogonal to. To achieve this, CAL provides several ways to express types, and a few basic types (some of which are parametric) 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 3.5.

minimal Caltrop type system

3.1 Type formats

There are three basic ways to express types in CAL, and addtionally four special syntaxes for common cases. The basic forms of types look like this:

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

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 $T[T_1, ..., T_n]$ is intended to stand for a *parametric type* T taking the types T_i as parameters. Such parametric type is also called a *type constructor*. The 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 identifier starting each type is called its *type name*—the name of the type `Integer` is `Integer`, the name of `List[Integer]` is `List`, and the name of `Matrix[element: Real, width = 4, height = 5]` is `Matrix`. These type names are important for type checking and the related operations on types—cf. section 3.5.

Type →
| '[' [Types] ']'
| '[' ID '→' Type { ',' ID '→' Type }] | '[' [Types] '- ->' Type ']'
| '[' [Types] '- ->' ']'

The form $[T_1, ..., T_n]$ last two forms are shorthand notations for often-used built-in types—cf. the next section.

It is important to emphasize again that most of these types remain *uninterpreted* by CAL in the sense that their meaning and their relation to each other is specified by the environment. The only exceptions to this rule are built-in types.

3.2 Built-in types

Built-in types are the types of objects created as the result of special language constructions, usually expressions. The following are built-in types in CAL:

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

$$List[T] < Seq[T], List[T] < Collection[T]$$

- `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.

- `[T1, T2, ..., Tn]`—product type of the types $T1$, ..., Tn, i.e. the type of tuples of these types.
- `Function[T1, T2]`—function closures (the result of lambda-expressions, cf. section 5.8.1) taking arguments in $T1$ and returning values in $T2$. For brevity, this type can also be written as `[T1 --> T2]`, or, if $T1$ is a product type $(T11, ..., T1n)$, as `[T11, ..., T1n --> T2]`.
- `Procedure[T]`—block closures (the result of proc-expressions, cf. section 5.8.2) taking arguments in T. For brevity, this type can also be written as `[T-->]` or, if T is a product type $[T1, ..., Tn]$, as `[T1, ..., Tn -->]`.
- `Actor[T1, T2]`—action closures (the result of **actor**-expressions, cf. section 5.8.3). The types $T1$ and $T2$ are tagged tuple types, identifying the input and output ports and their types.
- `InputPort[tokenType : T, multi = E]`—the type of an input port whose token type is T and which is a multiport if the boolean expression E is true, a single port otherwise.
- `OutputPort[tokenType : T, multi = E]`—the type of an output port whose token type is T and which is a multiport if the boolean expression E is true, a single port otherwise.

Note that CAL does explicitly not define primitive numeric types, strings etc. The choice of these types, 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 entirely up to the environment. See section 5.1 for more details, including a design rationale.

3.3 Structured types and locations

composite objects

locations

indices of any type

multiple indices

Most 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. CAL provides a general mechanism for identifying and retrieving such contained objects of structured objects, called *indexing*. The basic idea is that each contained object is at some *location* relative to the object that contains it. This location is defined by some other object, or tuple of objects, called its *index* or *indices*—e.g., in the case of lists, the locations are defined by the natural numbers between 0 and $n - 1$, if the list is of length n.

Indices are not restricted to be numeric, in fact they could be any kind of object. Which indices are permissible, or if indexing is supported at all, depends on each particular structured type. For instance, 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.

Indexing is not restricted to just one index—a data type, such as arrays, may support

any number of indices, which is equivalent to a single index tuple.

The following CAL types support indexing:

- `Seq[T]`—index type is the natural numbers.
- `Map[K, V]`—index type is `K`.
- `TaggedTuple[...]`—index type is `String`, the tags are the locations.
-

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.

3.4 Mutable 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 3.4.1 and 8.2.2.

Syntactically, mutation takes the form of an *indexed assignment*, such as this:

```
a[i] := x * y
```

Of course, the type of `a` must be a structured type that supports mutation, and `i` must be a valid location for that type (and object). After this assignment executed, the location `i` of the object contained in `a` is guaranteed to contain `x * y`.

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 `orthogonal` or `free`—all predefined types in CAL that are required to support mutation are of this kind. These are:

orthogonal/free types

- `List[T]`
- `Map[K, V]`

3.4.1 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.[1] 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

cloning

lazy cloning

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

chosen, mutations via a mutable variable must never have an effect on the value of other variables.

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.

IMPLEMENTATION CONSIDERATIONS.
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 `List`s 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 be 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.

3.5 Type checking framework

This section presents the operations that any given type system needs to provide in order to embed CAL into it. These operations, together with the basic minimal type system presented in section 3.2, constitute the CAL type framework. This framework

defines a *class of type systems*, viz. all those that contain the basic types and provide the following operations. Any embedding of CAL into a programming language, or any standalone implementation of it, will define a particular instance in this class.

In the following, we will assume variables such T and T_i etc. represent definite types, while C, C_i etc. represent *type constructors*, i.e. the type name of parameterized types. The set of all types is called $\mathcal{T}$.

Type ordering. The set $\mathcal{T}$ is partially ordered, i.e. there is a relation $\prec$ such that the following holds for any $T, T_1, T_2, T_3 \in \mathcal{T}$:

1. Reflexivity: $T \prec T$
2. Anti-symmetry: If $T_1 \prec T_2$ and $T_2 \prec T_1$ then $T_1 = T_2$.
3. Transitivity: If $T_1 \prec T_2$ and $T_2 \prec T_3$ then $T_1 \prec T_3$.

Chapter 4

Structure of an actor

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

Actor → **actor** ID
['[' TypePars ']'] '(' ActorPars ')' IOSig
{ ActorStateVarDecl | ActionRule | InitializationRule }
[SelectorDef]
{ StateVarDecl | ActionRule | InitializationRule }
end

TypePar → ID ['<' Type]

ActorPar → Type ID ['=' Expression]

IOSig → [PortDecls] '- ->' [PortDecls]

PortDecl → [**multi**] Type ID

ActorStateVarDecl → SimpleStateVarDecl ';'
| FuncProcVarDecl [';']

The header of an actor contains optional type parameters and actor parameters, and its I/O signature. This is followed by the body of the actor, containing a sequence of state declarations (section 5.2), actions (section 7), initialization rules (section 6.2), and at most one selector definition (section 7.3).

type parameters

type bounds

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 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 CONSIDERATIONS.
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.

The I/O 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.

single ports vs multiports

The names of the ports are visible as variables inside the actor definition. If the token type of the port is T, the type of the port name is `Port[T]` for single ports and `MultiPort[T]` for multiports, respectively. So an input port is really a mapping from channel identifiers to sequences of tokens.[1] This makes all the usual facilities for maps available for dealing with ports: `dom p` computes the set of channel identifiers defined for the port p, `p[a]` is the channel of port p identified by a.

[1] Note that the nature of channel identifiers is unspecified, and depends on the environment (cf. section C.2.2).

Chapter 5

Expressions

Expressions in CAL are side-effect-free and strictly typed. In the following we will define an expression by its typing rules and by a description of how its value is determined.

The following gives an overview of the kinds of expressions and expression syntaxes provided by CAL.

Expression → PrimaryExpression { Operator PrimaryExpression }

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

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

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

5.1 Literals

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`.

The exact types of the other literals is determined by the environment (cf. section C.2.1), but obviously 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 small as possible, to allow for a wide range of applications. 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.

5.2 Variables

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*.

CAL distinguishes between different kinds of bindings, depending on whether they can be assigned to, and whether the object they refer to may be mutated (cf. sections 3.4 and 8.2.2). The kind of binding is determined by the way a variable is declared. There are the following kinds of variable declarations:

- explicit variable declarations,
- actor parameters,
- input patterns,
- parameters of a procedural or functional closure,
- variables assigned to by a guarded assignment.

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

Variables declared in the head of a clause of a guarded assignment are assignable but non-mutable.

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

5.2.1 Explicit variable declarations

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.

Variables declared in any of the first four ways are called *state 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 *non-state 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.

5.2.2 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 particular, this includes the 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:

```
Integer n = 1 + k,
Integer k = 6,
Integer 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 be malformed:

```
Integer n = 1 + k,
Integer k = m - 36,
Integer 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.

This would lead to 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:

```
[Integer --> Integer] f = lambda (Integer n) --> Integer :
  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.

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 set D_v of a variable v among a set of variables V that are defined simultaneously in the same scope.

Definition 1 (D_v—the dependency set of a variable v). We assume that we have computed the sets F_v of all free variables in the definition of v, and I_v, which is the *immediate dependency set* of v, and which is defined as the subset of F_v that contains all free variables outside of closures. Further assume a set V of variables defined simultaneously, i.e. in the same scope (and therefore in the scope of each other's initialization expression). Then for any variable v, D_v is defined to be the smallest set such that

- $I_v \cap V \subseteq D_v$
- $\forall x \in D_v : F_x \cap V \in D_v$

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:

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

$$v \notin D_v$$

Note that, as in the example above, a variable may occur free in its own initialization expression, but still not be in its own dependency set, as this only includes those variables whose *value* must be known in order to compute the value of this variable[1]

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:

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

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

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).

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

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

$$v_1 \prec v_2 \iff v_1 \in D_{v_2}$$

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. Obviously, this relation is a non-reflexive (follows from well-formedness) partial order, since variables may not mutually depend on each other.

Example 1. Consider the following variable definitions:

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

Note that `f` and `g` are mutually recursive.

[1] 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 following lists the immediate dependencies and the free variable dependencies of each variable above,[2] along with their intersection with the set $\{a, f, g, b\}$, which is the set V in this case:

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

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

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

Now we compute

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

Then

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

Finally, we reach a fixpoint at

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

Trivially, we compute $D_f = D_g = D_b = \emptyset$ (all definitions that consist only of a closure have empty dependency sets, and so do definitions that only refer to variables which are not 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 2. Now consider the following slightly changed variable definitions, with an additional dependency added to `b`:

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

Again, the following table lists the dependency sets:

[2] Actually, when CAL is translated into CALCORE, a number of additional variables references are introduced, which refer to external functions that replace operators, control structures etc. Because these are always global, they do not affect the intersections with V, which are the relevant sets for our purposes here. So for simplicity, we will simply ignore them.

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

Now, computing D_a proceeds as follows:

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

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

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

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

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

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

5.2.3 Old variable references

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 3. `...`

```
Integer sum := 0;

action [a] ==> [old sum / sum] :
    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 (cf. also section **??**):

```
...

Integer sum := 0;

action [a] ==> [oldSum / sum]
    with oldSum = sum :
    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 3.4.1 for the implications of the original variable being mutable).

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 in at the beginning of the firing that created them.

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.

5.3 Function application

An expression of the form

$$E(E_1, ..., E_n)$$

is the application of a function to n parameters, possibly none. If the types of the E_i are T_i, then the value of the E expression must be a function of type

$$[T'_1, ..., T'_n - - > T]$$

where each $T'_i > T_i$. The static type of the application expression is the return type of the function, i.e. T.

Functions come in two forms: they are either the result of evaluating a lambda-expression (cf. section 5.8.1), or are provided as part of the environment (cf. chapter C). There is no difference in the way they are used inside expressions, but of course they differ in the way they are evaluated, and also in the way their types are determined.

5.4 Indexing

An indexing expression selects a subobject from a composite object (cf. section 3.3 for more details). Syntactically, indexing expressions are similar to function applications, except that the objects indexed are usually not functions. The general format is

$$E[E_1, ..., E_n]$$

where E must be of a type that supports an indexer, and the E_i 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.

5.5 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, which is defined by the environment for externally defined operators, and have fixed predefined

values for built-in operators (which are used to work on instances of built-in types, cf. appendix 5.5.2).

Operators are just syntactical elements—during execution, they are of course simply unary or binary functions, so there are no specific rules involved in type inference or evaluation of operators.

5.5.1 Operator syntax

TBD

5.5.2 Predefined operators

TBD

5.6 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.

5.7 Defining local variables

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** LocalDefs ':' Expression **end**

LocalDef → Type ID '=' Expression

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.

5.8 Closures

Closures are objects that encapsulate some program code along with the variable context (its *environment*) that was valid when it was created. CAL distinguishes three different kinds of closures, depending on what kind of code they encapsulate:

- *function closures* contain a parametric expression,
- *procedural closures* contain a parametric list of statements,
- *action closures* (also called an *actor*) contain a set of action clauses and state variable definitions.

The three 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 5.3), whereas the *execution* of a procedural closure to (a tuple of) arguments is a statement (cf. section 8.4), and so is *firing* an action closure on a tagged tuple of input token streams (cf. section 8.5).

5.8.1 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.

LambdaExpression → [**const**] **lambda** '(' [FormalPars] ')' '- ->' Type →
Expression **end**

FormalPar → Type ID

Function closures are 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 the mutable variables in their context.

The **const** keyword identifies those functional closures for which this is not the case, i.e. which do not refer to variables whose values may change. 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.

If the types of the formal parameters are T_1 to T_n, respectively, and the return type is T, then the type of the lambda expression is

$$Fun[[T_1, ..., T_n], T]$$

or more concisely

$$[T_1, ..., T_n - - > T]$$

The only built-in operation defined on a function closure is its *application* to a tuple of arguments, cf. section 5.3.

5.8.2 Proc-expressions and procedural closures

Procedural (or 'block') 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 block closures, this piece of code is a list of statements, i.e. executing a block closure is likely to have side effects (as opposed to the application of a function closure).

Syntactically, a block closure looks as follows:

ProcExpression → **proc** '(' [FormalPars] ')' [**var** LocalVarDecls ':'] { Statement } **end**

If the types of the formal parameters are T_1 to T_n, respectively, then the type of the lambda expression is

$$Proc[[T_1, ..., T_n]]$$

or more concisely

$$[T_1, ..., T_n - - >]$$

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 8.4.

5.8.3 Actor expressions and action closures

ActorExpression → **actor** [TypePars] '(' ActorPars ')' IOSig [TypeConstraints]
(StateDecl | Definitions | Action | InitializationRule)
end

TBD

5.9 Comprehensions

Comprehensions are expressions which iteratively construct one of the built-in composite objects: sets, lists, or maps. The syntax and semantics is very similar in all three cases, for sets it looks as follows:

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

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

The generators introduce new variables, and successively instantiate them with the elements of the 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.

The expressions following the **for**-clause are *filters*, i.e. they must be of type $Boolean$ and can refer to the generator variables to their left.

Once all generators have produced a variable assignment, the element expressions (those appearing to the left of the colon) are evaluated and the resulting values are added. After that, a new assignment is computed by first taking a new value for the rightmost generator variable, until that collection is exhausted, whereupon the next generator on the left is advanced etc. until the leftmost generator is exhausted and the process terminates.

The static type of a set comprehension is `Set[T]`, where T is the most specific common supertype (least upper bound) of the types of the element expressions. It is an error when this does not exist.

Example 4 (Set comprehensions). The expression $\{\}$, denotes the empty set, while
`{1, 2, 3}`
is the set of the first three natural numbers. The set
`{2 * a :  for Integer a in {1, 2, 3}}`
contains the values 2, 4, and 6, while the set
`{a :  for Integer a in {1, 2, 3}, a > 1}`
describes (somewhat redundantly) the set containing 2 and 3. Finally, the set
`{a * b :  for Integer a in {1, 2, 3}, for Integer b in {4, 5, 6}, b > 2 * a}`
contains the elements 4, 5, 6, 10, and 12.

Writing the above as
`{a * b :  for Integer a in {1, 2, 3}, b > 2 * a, for Integer 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`.

List comprehensions work in a similar fashion, except that the order is of course relevant to lists. Syntactically, they look like this:

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

Because lists are order-sensitive, the list
`[ a :  for Integer a in [1, 2, 3] ]`
is different from the list `[ a :  for Integer a in [3, 2, 1] ]`
If the collection computed in a generator is *not* itself a list but a set, as in
`[ a :  for Integer a in {1, 2, 3} ]`
then the order of the elements in the resulting list will be indeterminate.

Element expressions are of course added in left-to-right order to the list at each iteration.

The typing rules for lists are analogous to those for sets: if the least upper type bound for the element expressions is T, the type of the list comprehension is `List[T]`.

Map comprehensions again work similarly, but they construct map objects, i.e. finite mappings from *keys* of one type to *values* of another. Their syntax looks like this:

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

Mapping → Expression '->' Expression

Instead of element expressions as in the previous two comprehensions, we have *mappings*, describing the key/value pairs to be added to the map. Maps are order-insensitive—if two key evaluations result in the same value, the it is indeterminate which key/value assignment will end up in the map.

The type of a map comprehension is `Map[K, V]`, where K is the least upper type bound of the key expressions, and V is the least upper type bound of the value expressions. Both must exist, otherwise the type is undefined.

Chapter 6

Actor state variables and initialization

The state of an actor is exactly that piece of information which is stored between *firings*, and which the code associated with an action can therefore use to influence the actions taken by future firings.

6.1 State variable declarations

An actor may contain any number of state variable declarations at its top level (cf. section 5.2 for details on variable declarations). These variables are called *actor state variables*. One such declaration has the following syntax:

StateVarDecl → SimpleStateVarDecl
    | FunProcVarDecl

SimpleStateVarDecl → Type ID [(':=' | '=') Expression]
    | [**mutable**] Type ID [(':=' | '=') Expression]

FunProcVarDecl →

State variable names must be unique actor-wide, and may not coincide with the names of actor parameters or ports, because they are defined in the same scope. As a consequence, the expressions used to define their initial values can refer to actor parameters, ports, and any state variable, as long as this does not include any circular dependencies (cf. section 5.2.2).

6.2 Initialization rules

TBD

Chapter 7

Actions

An action is an atomic (from the outside) piece of computation performed by an actor, usually in response to some input. The definition of an action needs to describe three things:

- the *consumption* of input tokens,
- the *production* of output tokens,
- the *change of state* of the actor.

Usually, an actor definition contains a number of action definitions. Whenever it needs to make another computational step, the actor needs to choose one of them. It does so based on the availability of input tokens, and possibly based on further conditions on their values, and its own state.

The general structure of an action definition is as follows:

ActionRule → [ID ':'] **action** ActionHead [':' StatementsOpt] **end**

ActionHead → [InputPortPatterns] '==>' [OutputPortExpressions]
    [**guard** Expressions]
    [**var** StateVarDecls]

InputPortPattern → [ID '::']
    ChannelPattern [ChannelSelector] [RepetitionExpression]

ChannelPattern → SequencePattern
    | '[' [TokenPatterns ['|' SequencePattern]] ']'

TokenPattern, SequencePattern → ID

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

RepetitionExpression → **repeat** Expression

OutputPortExpression → [ID '::']
Expression [ChannelSelector] [RepetitionExpression]

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 port expressions.

Patterns and expressions are associated with ports either by position or by name. These two kinds of association cannot be mixed. So if the actor I/O signature is
`T Input1, T Input2 --> T Output`
an input pattern may look like this:
`[a], [b]`
(binding `a` to the first token coming in on `Input1`, and binding `b` to the first one from `Input2`). It may also look like this:
`Input2: [c]`
but never like this:
`[d] Input2:[e]`

The following sections elaborate on the structure of the patterns and 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.

7.1 Input patterns

Firing an action may consume some tokens from the input ports of the actor, and produce tokens on its output ports. On the input side, *input patterns* are used to describe the token consumption of a given action.

In addition to providing information about the number of tokens consumed by that action, such a pattern introduces a number of variables which are used to hold the token values read from the ports and channels.

Matching an input pattern is part of matching an action—an action matches if its input patterns match and its activation condition (its *guard*) evaluates to true. See section 7.4 for details.

Input patterns are important to the expressiveness of CAL. They allow the concise and intuitive description of input conditions, while at the same time facilitating a high degree of straightforward static analysis of properties such as:

- number of tokens consumed by an action,
- whether that number is constant, depending on parameters, or depending on the state,
- which channels are to be read from (in case of a multiport),
- whether these are constant, depending on parameters, or varying with the state.

The general format of input port patterns is a channel pattern, followed by an optional channel selector, followed by an optional repetition count:

[ID ':'] ChannelPattern [Expression] [**repeat** Expression]

As mentioned above, the channel pattern may be preceded by the name of the port it is supposed to be used for. If an action contains such a *tagged* pattern, all input patterns it uses must be tagged, but it does not have to provide a pattern for each port. If its input patterns are untagged, it must provide exactly as many patterns as the actor has input ports, and the patterns are used for the ports in the order in which they occur and the ports are declared in the actor head.

A channel pattern introduces a number of variables, indeed the entire input pattern may be thought of as an elaborate variable declaration. The exact type of the variables introduced will depend on the presence or absence of the channel selector or the repetition count.

There are three basic formats for channel patterns:

- $[t_1, ..., t_n]$—This introduces n variables, which represent the first n tokens available on that channel. The action code will have access to precisely these n tokens from the corresponding port, and it is guaranteed to have accessed them after the action is finished. This kind of pattern is also referred to as a *bounded (channel) pattern*.

- $[t_1, ..., t_n \mid s]$—This introduces n+1 variables, the t_i representing the first n tokens available on the input channel, while s represents the remainder of the sequence, again potentially infinite. Similarly, only the execution of the action code will determine which tokens in that remainder are accessed, but this pattern guarantees that the first n tokens are used by the action.

- s—This introduces one variable which represents the entire (potentially infinite) sequence of tokens on the respective input channel. The execution of the action code will determine which tokens are accessed, when and in which order. This is really a short form of $[\mid s]$.

In summary, these patterns introduce two kinds of variables:

- *token variables* (the t_i in the examples above), which hold the values of individual tokens and

- *sequence variables* (the s in the examples above), which are bound to sequences of tokens.

It is important to emphasize again that a variable introduced by such a pattern may end up representing more than one token or sequence of tokens, depending on the presence of a channel selector (and its type) and/or repetition counts. We will now discuss the different possible configurations, specifying for each kind of port pattern which type of object the token and sequence variables in it are bound to. In this discussion, we assume that the token type of the port is declared to be T.

7.1.1 Single port patterns without repetition count

The simplest class of port pattern is simply a channel pattern as discussed above. This can only occur for a single port, which has precisely one channel.

Consequently, the token variables simply hold the value of one token, and are therefore of type `T`. Similarly, the sequence variables are of type `Seq[T]`.

Example 5. Assume the input sequence [1, 2, 3, 4]. The pattern `[a, b]` matches, and binds `a` to 1, `b` to 2.

The pattern `[a, b | c]` also matches, and binds `a` to 1, `b` to 2, and `c` to [3, 4].

The pattern `[a, b, c, d | e]` also matches, binding `a`, `b`, `c`, and `d` to 1, 2, 3, and 4, respectively, and `e` to the empty list [].

The pattern `[a, b, c, d, e]` does not match.

7.1.2 Single port patterns with repetition count

A single port pattern with repetition has the following basic form:

[ID ':'] ChannelPattern **repeat** Expression

The expression must evaluate to a non-negative natural number, say n. Then the token variables in the channel pattern are of type `List[T]`, and are guaranteed to be bound to lists of length n. A pattern with a repetition count must not contain any sequence variables, i.e. only bounded patterns may be followed by a repetition count.

Conceptually, a pattern is matched n times against the incoming tokens, and each token variable contains the list of tokens the corresponding position of the pattern was matched against, in order of their occurrence in the stream. So e.g., the third variable in a pattern containing five variables (and assuming a repeat count of four) is bound to a list containing the token at position 3, 8, 13, and 18 in the input sequence.

Example 6. Assume the input sequence [1, 2, 3, 4, 5, 6, 7]. The pattern `[a, b] repeat 1` matches and binds `a` to [1] and `b` to [2].[1]

The pattern `[a, b] repeat 3` matches and binds `a` to [1, 3, 5], `b` to [2, 4, 6].

The pattern `[a, b] repeat 0` also matches (in fact, it matches any input sequence) and binds both `a` and `b` to the empty list.

The pattern `[a, b] repeat 5` does not match.

7.1.3 Channel selectors in multiport input patterns

Multiports have any number (including zero) of different *channels*, i.e. individual 'subports' which are independently associated with their own input sequences. Actors may want to read from all of these sequences, any of them, or from specific sequences. The patterns one specifies for multiports require the specification of which channels the pattern is to be applied to. This is called a *channel selector*, and there are two forms of

[1] In this example we only use constant repetition counts for simplicity, but it should be very clear that these can be *any* expression, and in fact repetition counts that depend on actor parameters, actor state, or even other input tokens are the really interesting ones in practice.

it: *single channel selectors* and *multichannel selectors*, which select either exactly one channel, or any number (including zero and one) of channels.

An input pattern with a channel selector has one of the following forms:

ChannelPattern **at** Expression

ChannelPattern **at*** Expression

ChannelPattern [**at***] **any**

ChannelPattern [**at***] **all**

In the first two cases, an expression determines which channels are to be selected. In the first case, the expression is preceded by the keyword **at**: this is a single channel selector, and the expression must evaluate to a `ChannelID`, which then determines the selected channel. When the expression is instead preceded by the keyword **at***, it must evaluate to a `Collection[ChannelID]`. In that case the selector is a multichannel selector, and the channel identifiers in the collection determine the selected channels.

The keywords **any** and **all** are always multichannel selectors (they may optionally be preceded by the keyword **at***, which has no effect). **any** selects all those channels that match against the pattern, possibly none at all. **all** selects all channels that are defined for the port.

Whatever form is chosen, when a pattern is matched against the channels of a port, it is matched *homogeneously*, i.e. the number of tokens read from each channel selected by the channel selector is identical.

7.1.4 Multiport patterns without repetition count

In the case of a single channel selector, the pattern is matched against precisely one channel, so the types of token and sequence variables are the same as for simple port patterns without repetition counts, i.e. `T` and `Seq[T]`, respectively. If the channel identified by the value of the selector expression does not exist, the pattern does not match.

If the selector expression is of type `Collection[ChannelID]`, or one of the two keywords, the pattern may be matched against any number of channels, including zero. In that case, token variables are of type `Map[ChannelID, T]`, and sequence variables of type `Map[ChannelID, Seq[T]]`.

The domains of all the maps bound to variables by a pattern are guaranteed to be equal, and to be either:

- equal to the set-interpretation of the collection that the selector expression evaluated to,[2]
- equal to the domain of the port (i.e. the set of all channels) in case the selector expression was the keyword **all**, or

[2]Since the selector expression may evaluate to a list, or any other collection, this might differ from the actual value of the selector expression.

	no repetition count	with repetition count
single port/channel	`T`	`List[T]`
multichannel	`Map[CID, T]`	`Map[CID, List[T]]`

Table 7.1: Token variable types depending on pattern type and repetition count.

	no repetition count	with repetition count
single port/channel	`Seq[T]`	N/A
multichannel	`Map[CID, Seq[T]]`	N/A

Table 7.2: Sequence variable types depending on pattern type and repetition count.

- equal to the set of matching channels in case the selector expression was the keyword **any**.

Example 7. Consider for example an actor that has the following input port signature:
`multi T input, ChannelID select`
Now consider an action in this actor that starts with following input patterns:
`[a]` {`sel`}`, [sel]`
These patterns match if

- the *select* port has a token and
- the *input* channel determined by that token has a token.

The effect is to bind the variable sel to the channel selection token, and a to the token read from the multiport (cf. section 7.4 for more discussion on the way variables are bound and patterns are matched).

7.1.5 Single channel input patterns

7.1.6 Multichannel input patterns

7.2 Output expressions

In an action header, the input patterns are followed by *output expressions*, which are used to compute the tokens to be sent to the output ports. The genral format of an output expression is similar to that of an input pattern:

[ID '::'] Expression [Expression] [**repeat** Expression]

The key difference is that instead of the actual channel pattern, an output expression contains an expression (the first expression above)—this expression is called the *token expression*. Its type and interpretation depends on the presence or absence of the selector expression (the second expression) as well as its type, and/or the presence or absence of the repeat expression (the one following the **repeat** keyword), as shown

	no repetition count	with repetition count
single port/channel	List[T]	List[Seq[T]]
multichannel	List[Map[CID, T]]	List[Map[CID, Seq[T]]]

Table 7.3: Output expression type depending on channel pattern type and repetition count.

in Fig. 7.3. As in the case of input patterns, the repeat expression must evaluate to an integer, and the channel selector expression must be either of type `ChannelID` or `Collection[ChannelID]`, or one of the keywords **any** or **all**.

7.2.1 Single channel output expressions

Output expressions for single channel are those without a channel selector. Hence they come only in two forms, with or without a repeat expression. The one without a repeat expression thus consists only of the token expression itself, which must simply evaluate to a list of tokens, and these tokens are sent to the corresponding output port.

A single port output expression with a repetition count must have a token expression that evaluates to a list of sequences of tokens. If the repeat expression evaluates to the repetition count n, the resulting token sequence S' is computed as follows:

Assume S is the list of sequences resulting from evaluating the token expression. Let m be the length of S. Then

$$\begin{aligned} S' = [&S[0][0], ..., S[m-1][0], ..., \\ &S[0][i], ..., S[m-1][i], ..., \\ &S[0][n-1], ..., S[m-1][n-1]] \end{aligned}$$

Here, $S[i][j]$ is the j-th element in the i-th sequence in S. This assumes that each of these sequences has at least n elements—it is an error if a sequence has less than those n elements.

Example 8. The output expression
`[[1, 2, 3],[4, 5, 6, 7],[8, 9, 10]] repeat 3`
produces the following sequence of tokens sent to the corresponding output port:
`[1, 4, 8, 2, 5, 9, 3, 6, 10]`

Example 9. The following output expression
`S`
is equivalent to
`[S] repeat 3` if `S` contains a list with three elements. The second version may be preferable, however, because it makes it easier to analyze precisely how many tokens the action will output.

Example 10. The output expression
`S`
is always equivalent to

`[S] repeat #S`
Neither form has advantages with respect to its static analyzability, as it is clear in both cases that the number of tokens produced depends on the length of (the value of) `S`.

7.2.2 Multichannel output expressions

Output expressions for multiports

7.3 Selector expression and selector tags

TBD

7.4 Action matching

A significant part of the expressiveness of CAL comes from the way actions are chosen. In general, an actor can consist of any number of action definitions. When fired, it has to select one of them (or none, if none applies) for acting on the inputs and computing a new state and outputs. An actor can only select an action that *matches* the current input in the current state. Such an action is said to be *firable*. This section describes the matching process that determines whether an action is firable.

The parts of an action definition that are considered during matching are the following:

- The input patterns.
- The local variable declarations in the **var**-clause.
- The boolean expressions in the **guard**-clause.
- The selector tag.

The variables declared by the input patterns and those in the **var**-clause are declared in the same scope, and their definitions are evaluated *in that scope*.[3] This implies that they can depend on each other complex ways—the only constraint is that they may not depend on each other in a circular fashion. A variable *depends* on another variable as follows.

- Variables in input patterns depend on all variables occurring free in any channel selector expression.
- Variables in **var**-clauses depend on all variables occurring free in their defining expressions (if they have one).

[3] Cf. section 8.1 for details on **var**-clauses.

For example, the following patterns and **var**-clauses are valid:

```
[a] {sel}, [sel] --> ...
[a], [b] --> ...  with sum := a + b
[a] any, [b] any, [c] s --> ...  with s = dom a * dom b
```

By contrast, the following patterns and **var**-clauses lead to circular dependencies, and are therefore invalid:

```
[a] f(a) --> ...
[a] dom b, [b] dom a --> ...
[a] s --> ...  with s = union({a[x] :  for x in dom a})
```

The order established by the dependencies constrains the sequence in which things are evaluated and, more importantly, in which tokens are read from the input ports. For example, in the following case

```
[a] any, [b] any, [c] s --> ...  with s = dom a * dom b
```

the tokens from the first two multiports can be read in any order, but both groups must be read and the tokens bound to the respective variables before the third port can be read. Of course, in an intermediate step, the expression defining s is evaluated and the result bound to the variable s.

The **guard**-clause contains a set of Boolean expressions that may impose additional conditions on the values of the variables bound by this process. For example, the following constraint states that the channels on which a token is available must be the same for both input ports:

```
[a] any, [b] any --> ...  where dom a = dom b
```

This is a valid formulation of the second illegal clause above.

An action matches an input in a given state if and only if

- The sequential reading of tokens as constrained by the variable dependencies finds enough tokens on each selected channel to bind the token variables.
- The expressions in the **guard**-clause all evaluate to **true**.

In case more than one action is firable at some point, disambiguation is left to the environment (cf. section C.5).

Chapter 8

Statements

The execution of an action (as well as initialization) happens as a sequence of statements, each of which may change the state of the actor. CAL provides the following kinds of statements:

Statement → AssignmentStmt
| ExecStmt
| BlockStmt
| IfStmt
| GuardedAssignmentStmt
| WhileStmt
| IterationStmt

Before discussing the individual kinds of statements in subsequent sections, the nest section presents a few details on the definition of local variables.

8.1 Local variables

Several statement constructs (and in fact action definitions themselves) may contain **var**-clauses, which serve to define new variable symbols that are local to the scope in which the **var**-clause occurs.

In addition to the variables that can be introduced in a **state**-block (viz. simple assignable and structured mutable variables), a **var**-clause may also contain *definitions* which are constant in the scope, i.e. they cannot be assigned to and neither are they mutable.

8.2 Assignment

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

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

Index → '[' [Expressions] ']'

An assignment without an index is a *simple assignment*, while one with an index is called an *indexed assignment*, or a *mutation* (cf. sections 3.3 and 3.4 for more information on indices and mutability of objects).

8.2.1 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.

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 a subtype of the declared type of the variable). The effect of the assignment is of course that the variable value is changed to the value of the expression. The original value is lost.

8.2.2 Indexed assignment

Conceptually, a mutable variable represents a number of assignable *locations*. In order to assign to these locations, and also in order to access them, the variable must be *indexed*.

In CAL, an index into a variable is an object (often a tuple) enclosed in square brackets.[1] An assignment into a mutable variable thus may look like this:

```
v[k, x*y] := a
```

A state variable that is declared to be mutable must be of a type that supports mutation. In order to support mutation, a type must have an *(indexed) mutator*, i.e. a procedure that changes specific locations in the object, which is invoked when such a location is assigned to. A type that supports mutation is called a *mutable type* (cf. section 3.4 for a discussion of the characteristics of mutable types).

In general, locations in a mutable type need not be independent, i.e. assigning to a location may affect the values of other locations.

8.3 Control flow constructs

CAL provides a number of constructions to write iterative programs and control the flow from statement to statement. We distinguish between *branching* constructs, which

[1] A tuple may be written without its outer parentheses, i.e. instead of `[(1, 2)]` one may simply write `[1, 2]`.

choose between a number of possible statement sequences based on some condition, and *iteration* constructs which repeatedly execute a set of statements.

8.3.1 Branching constructs

The most basic branching construct is the conditional statement:

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 is, of course, of type `Boolean`.

The following construct, a *guarded assignment statement* is used to branch based on the actual data types of a number of objects. The syntax is as follows:

GuardedAssignmentStmt → **assign** Expressions **to**
{ SimpleDeclPatterns ':' [Statements] **end** }
[**default** ':' [Statements] **end**]
end

SimpleDeclPattern → '_' | Type ID

Following the expression list is a list of cases, which in turn consist of a list of declaration patterns (the number of patterns must be the same as the number of expressions) followed by a sequence of statements. The statements are executed if the patterns *match* the values computed by the expressions, where values and patterns are matched according to their position.

A declaration pattern is either the underline character or a variable declaration consisting of a type and a variable symbol. An underline character matches any object, and does not introduce a new variable. A declaration matches only objects of the given type (or any subtypes of it). If it matches, it binds the corresponding object to the variable symbol, which will be visible inside the sequence of statements belonging to this case.

8.3.2 Iteration

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.

WhileStmt → **while** Expression [**var** LocalVarDecls] '→' [Statements] **end**

The **foreach**-construct allows to iterate over a collections, successively binding variables to the elements of the expression and executes a sequence of statements for

each such binding.

IterationStmt → **foreach** Type ID **in** Expression { ’,’ ID **in** Expression }
[**var** LocalVarDecls] ’:’ [Statements] **end**

Variables are bound from left to right, and the expressions defining the collections the variables are iterated over may use the values bound to the variables to their left.

Example 11. The following code fragment

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

8.4 Executing block closures

The only predefined operation on block closures (cf. section 5.8.2) is their execution. An execution statement is written as follows:

ExecStmt → **exec** Expression ’(’ [Expressions] ’)’;’

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

8.4.1 Statement blocks

A special case of the execution of a block closure are simple statement blocks, which are essentially syntactic sugar for one kind of execution statement. They look like this:

BlockStmt → **begin** [**var** LocalVarDecls ’:’] { Statement } **end**

The form
`begin with <decls>:  <stmts> end`
is equivalent to
`exec proc () with <decls>:  <stmts> end () ;`

8.5 Firing an action closure

TBD

Chapter 9

Exceptions

TBD

Part III

Appendices

Appendix A

Keywords

Keyword	Use
`at`	
`at*`	
`else`	
`end`	
`if`	
`while`	
`foreach`	
`actor`	
`endactor`	
`endif`	
`action`	

Appendix B

Standard library

Appendix C

The environment

C.1 Compile-time vs runtime interfaces

C.2 Types

C.2.1 Handling literals

C.2.2 Channel identifiers

C.2.3 Mutable types

Cloning becomes necessary if the value of a mutable variable is bound inside a functional closure, or if it is assigned to a non-mutable variable. In this case, in order to ensure that changes to the mutable variable do not have unwanted side effects, the object may have to be cloned.[1]

[1] Since cloning may be a costly operation, implementations are encouraged to be 'smart' about when and what to clone. They may realize a copy-on-write strategy that shares objects until the become mutated, and triggers cloning by those mutations. They may also only partially clone an object on those locations where it was modified.

C.3 Functions and procedures

C.3.1 Defining a function

C.3.2 Defining a procedure

C.3.3 Operators

C.4 Predefined types and operators

C.5 Action matching and disambiguation

C.6 Example: Split-phase execution in Ptolemy II

Bibliography

[1] The Ptolemy Project. Department EECS, University of California at Berkeley ($http://ptolemy.eecs.berkeley.edu$). 6

[2] Chris Chang, Johan Eker, Jörn W. Janneck, and Lars Wernli. Caltrop—developer's handbook. Technical memorandum, Electronics Research Lab, Department of Electrical Enginee and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, 2002. 8

[3] 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. 6

[4] Edward A. Lee. A denotational semantics for dataflow with firing. Technical Memorandum UCB/ERL M97/3, Electronics Research Lab, Department of Electrical Enginee and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, January 1997. 6

[5] Edward A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002. to appear. 7

[6] 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. 7