Chi 1.0 reference manual

Systems Engineering Group
Department of Mechanical Engineering
Eindhoven University of Technology
PO Box 513
5600 MB Eindhoven
The Netherlands
http://se.wtb.tue.nl/

SE Report: Nr. 2008-04

Chi 1.0 reference manual

A.T. Hofkamp and J.E. Rooda

ISSN: 1872-1567

SE Report: Nr. 2008-04
Eindhoven, July 2008
SE Reports are available via http://se.wtb.tue.nl/sereports

Abstract

The Chi 1.0 Language Reference Manual describes the syntax and the static semantics of all available Chi 1.0 language constructs in a compact manner. It is set up as a reference manual, intended for users that know the language but need a more precise and complete description of a construct.

Contents

Chapter 1
Introduction

The Chi language is a very rich language; it has a lot of different data types and statements. This makes it possible to express models in a very compact way. Also, Chi is a hybrid language, which means that you can write discrete-event models, continuous-time models, and combined discrete-event and continuous-time models, Finally, the language is largely based on mathematics. This makes it possible to attach a clear meaning to models.

The combination of the wide variety of models and huge expressiveness makes that a single implementation of the language in a tool is not feasible, such an implementation would become too big, too complex, or too slow. As a result, the implementation uses a divide and conquer strategy. Rather than having one big do-it-all implementation, several implementations for different subsets of the language exist in parallel. Depending on the kind of model and purpose of the model, the modeler chooses an appropiate implementation.

Having several implementations for different subsets of the language also influences the structure of the manuals. This document, the Chi Language Reference Manual (CLRM), describes the full language. The subset of the language supported by a specific tool is not described here, but in the manual of that tool instead. In general, the latter is a very short document, it mainly describes which subset is supported without going into details. For example, a tool manual may state that the tool supports the list data type. Details of the list data type (what it is, its syntax, operations on lists, etc) are not provided, they should be looked up here, in the CLRM.

1.1 Page references

Since this document is a reference manual, it is structured around key aspects of the language such as definitions, statements, and expressions. When one aspect is explained, it is assumed that all other aspects are known. To assist the reader in finding more details about the aspects he/she may not be familiar with, the manual contains page references to its definition in the text, for example railroad diagrams[page 2]. If you want to know more about railroad diagrams, you can go to page 2 to read about its details.

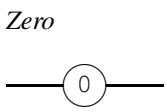

Figure 1.1: Raildiagram of *Zero*.

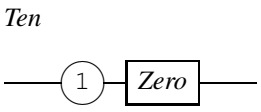

Figure 1.2: Raildiagram of *Ten*.

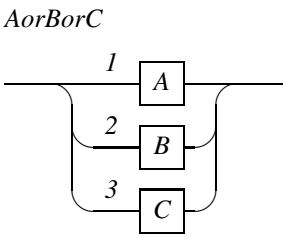

Figure 1.3: Raildiagram of *AorBorC*.

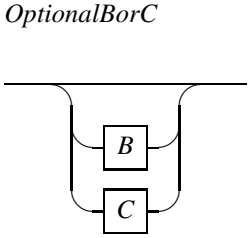

Figure 1.4: Raildiagram of *OptionalBorC*.

In addition, there is an index added at the back of the manual.

1.2 Reading railroad diagrams

The syntax and the grammar of the Chi language are explained using syntax diagrams, also known as rail(-road) diagrams. One of the smallest diagrams is shown in Figure 1.1. The name of the rule in the diagram is *Zero*. It is read by starting at the left, and following the line without making sharp turns until it ends at the right hand side. In this case, as you go from left to right the line passes through a rounded rectangle[1] with a '0' in it. This means that the syntax of *Zero* is '0'.

Diagrams can be nested. When the contents of another diagram should be used, the name of the needed diagram is shown in a rectangular box. An example is shown in Figure 1.2. This diagram starts with a '1', followed by the contents of diagram (also called *Block*) *Zero*. *Ten* is thus written as '1' '0'.

Choice between two or more alternatives is also possible in a diagram. An example can be seen in Diagram *AorBorC* in Figure 1.3. This diagram denotes that either Block *A*, Block *B*, or Block *C* can be chosen. The numbers 1, 2, and 3 are track numbers. They have no meaning in the diagram other than allowing the accompanying text to refer to a certain point in the diagram (for example, *B* at Track 2).

In a diagram with choices, the first alternative (at Track 1) is sometimes empty, which indicates an optional piece of text (choose between nothing, *B*, or *C*). Such a rule is shown in Figure 1.4.

Repetition looks similar to choice, as in the *ManyAB* diagram in Figure 1.5. The difference is that the circle segments at the top are in the opposite direction compared to the previous diagram. Since you are following the lines like a train, you may not make sharp turns, so you

[1] The rectangle is so short in this example that it looks like a circle.

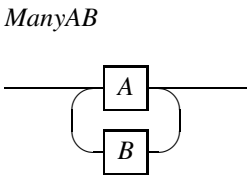

Figure 1.5: Raildiagram of *ManyAB*.

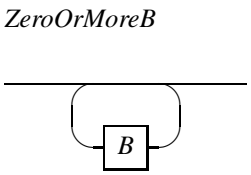

Figure 1.6: Raildiagram of *ZeroOrMoreB*.

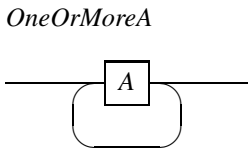

Figure 1.7: Raildiagram of *OneOrMoreA*.

must first pass through Block *A* before you can go down through Block *B* (from right to left), then back up and again through Block *A*, etc.

Sometimes, the *A* part is empty, which means that there is choice between *zero or more* times *B* as shown below by Diagram *ZeroOrMoreB* in Figure 1.6. If the *B* part is empty instead, it means that there is a choice for *one or more* times *A* as shown in Figure 1.7.

All the above constructs (sequence of Diagram *Ten*[page 2], choice of Diagram *AorBorC*[page 2], and repetition of Diagram *ManyAB*[page 2]) can be combined and nested arbitrarily. The resulting railroad diagrams describe the order of tokens in a Chi specification.

Chapter 2
Lexical syntax

The syntax of the language exists at two levels. The bottom level is *lexical syntax*. This syntax is defined at character level. In particular, white-space (the precise definition is in Section 2.2) is not allowed between the boxes. All diagrams in this chapter use the lexical syntax level. The upper level is the context-free syntax level. At this level, white space constructs may be added (or in some cases must be added) between boxes in the diagram, such as space or new-line characters, or comments. Except for this chapter, all chapters use the context-free syntax level.

The input to the tools is the ASCII character set. That is also the format in which the input has to be delivered to the tools. To prevent misunderstanding about the characters allowed in this chapter, references are made to Unicode characters. Such a reference starts with an uppercase 'U' followed by a four digit hexadecimal Unicode[1] character identification, for example U005C.

2.1 Lexical tokens

The Chi language has a number of basic tokens at lexical level. These are explained in the following sections.

2.1.1 Unsigned decimal numbers

Unsigned decimal numbers are a sequence of one or more decimal digits (U0030 upto and including U0039) as shown in the lexical *Number* diagram in Figure 2.1. The node 0 | 1 | ... | 9 means selection of one of the characters '0' or '1' or ... or '9' (one of the U0030 upto and including U0039). The node 1 | 2 | ... | 9 has a similar meaning, except that the '0' (U0030) may not be chosen.

As you can see in the diagram, value 0 is entered as the single character '0'. All other (un-

[1] The entire Unicode character set is available at the Internet at `http://www.unicode.org/charts/`.

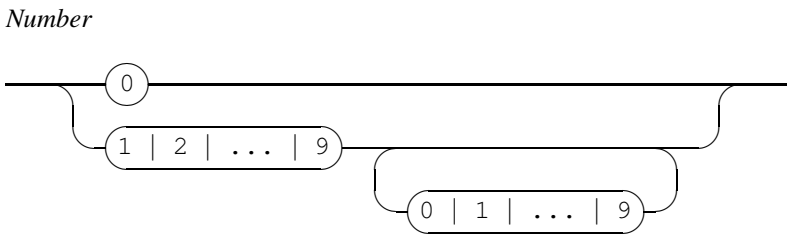

Figure 2.1: Raildiagram of *Number*.

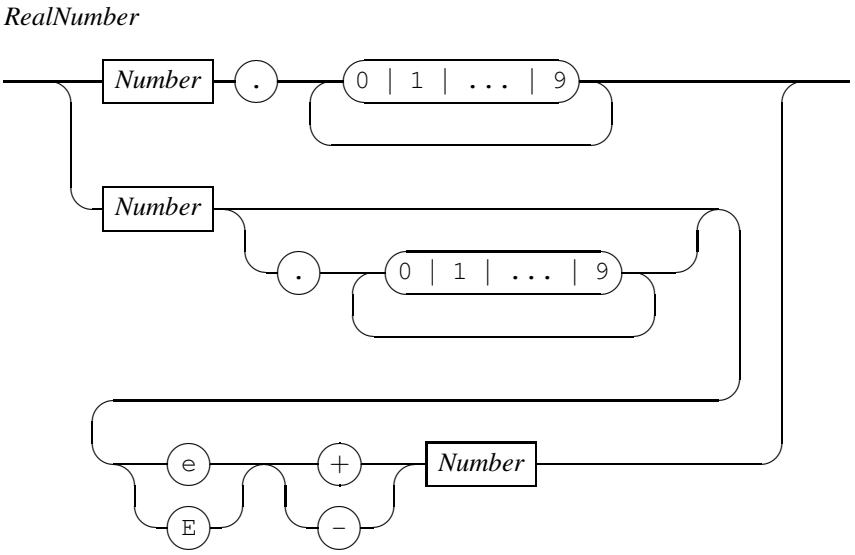

Figure 2.2: Raildiagram of *RealNumber*.

signed) values start with a non-zero digit. In other words, the '0' may not be used as prefix for an unsigned decimal number.

Examples

Example	Explanation
`30458`	Correct
`0`	Correct
`023`	Incorrect, unsigned decimal number may not use '0' prefix
`1 23`	Incorrect, no white space allowed in a number

□

2.1.2 Real numbers

The syntax of a real number (a value of type `real`) is built on top of the lexical syntax of *Number*[page 5]. Figure 2.2 shows the lexical syntax of a real number in the *RealNumber* diagram.

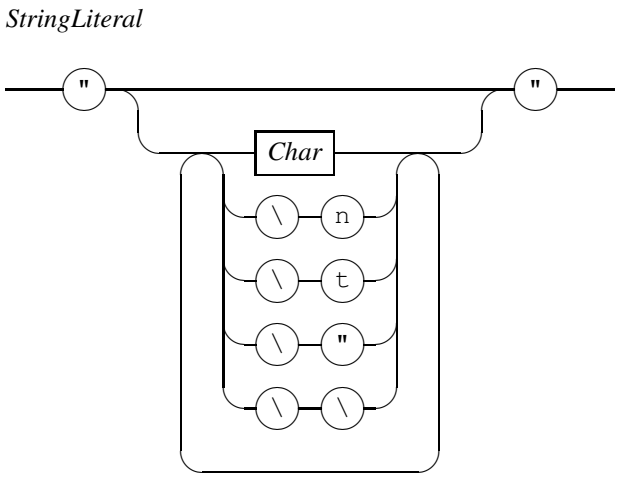

Figure 2.3: Raildiagram of *StringLiteral*.

As you can see, a real number is a sequence of digits with a dot somewhere, or a sequence of digits optionally with a dot and an exponent suffix with a signed exponent.

Examples

Example	Explanation
`0.0`	Correct
`1e+5`	Correct
`3e2`	Incorrect, need sign of exponent
`8 e-4`	Incorrect, white space not allowed in real number

□

2.1.3 String literals

The syntax of literal strings is shown in the *StringLiteral* diagram in Figure 2.3. A string literal starts with a double quote character (the QUOTATION MARK character U0022), followed by zero or more characters (explained next), and ends with another double quote character U0022.

A character between the double quote characters is a single printable ASCII character (U0020 upto and including U007E, except for the double quote character '"' U0022 and the backslash character '\' U005C) represented by the *Char* block in the diagram. In addition, you can add a LF character U000A by using a sequence of '\' and 'n' (U005C followed by U006E) characters. A TAB character U0009 can be inserted by writing a sequence of a '\' and a 't' characters (U005C followed by U0074). Finally, you can insert a double quote character U0022 by prefixing it with a backslash (that is, write '\"', U005C followed by U0022), and a backslash character also by prefixing it with itself (that is, write '\\', two U005C characters).

The value of a string literal is the sequence of characters between the two double quotes with the backslash sequences translated to the character they represent.

Identifier

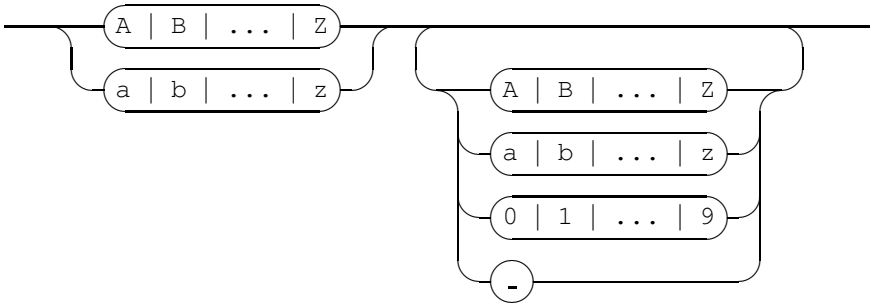

Figure 2.4: Raildiagram of *Identifier*.

2.1.4 Identifiers and keywords

An identifier starts with a letter (U0041 upto and including U005A, and U0061 upto and including U007A) or an underscore character U005F, followed by zero or more letters (U0041 upto and including U005A, and U0061 upto and including U007A), digits (U0030 upto and including U0039), and/or underscore characters U005F. The *Identifier* diagram is shown in Figure 2.4.

Some identifiers are special in the sense that they are reserved to be used for a particular purpose. Such identifiers are called keywords. Keywords are shown explicitly as a terminal box in the railroad diagrams[page 2], and may not be used for any other purpose. In particular, a keyword may not be used as name of a (user-defined) object in the specification.

Examples

Example	Explanation
`ABCDEF`	Correct
`An identifier`	Incorrect, white space not allowed in an identifier
`variable3`	Correct
`3f`	Incorrect, identifier may not start with a digit

□

2.1.5 Filenames

A filename or path is a non-empty sequence of path elements seperated by forward slashes U002F, optionally prefixed by a forward slash U002F. A path element is a non-empty sequence of letters (uppercase letters U0041 upto and including U005A, and/or lowercase letters U0061 upto and including U007A), digits (U0030 upto and including U0039), minus-sign U002D, underscore character U005F, and/or dots U002E. The lexical syntax is shown in the *Filename* railroad diagram in Figure 2.5.

In general, *filename* is used when the character sequence refers to a file. The term *path* is a more general notion, and may also refer to other things at the file system, such as directories. A filename or path starting with a forward slash is called an *absolute filename* or *absolute path*, since it states the entire path to a file (or directory) from the root of the file system. A *relative*

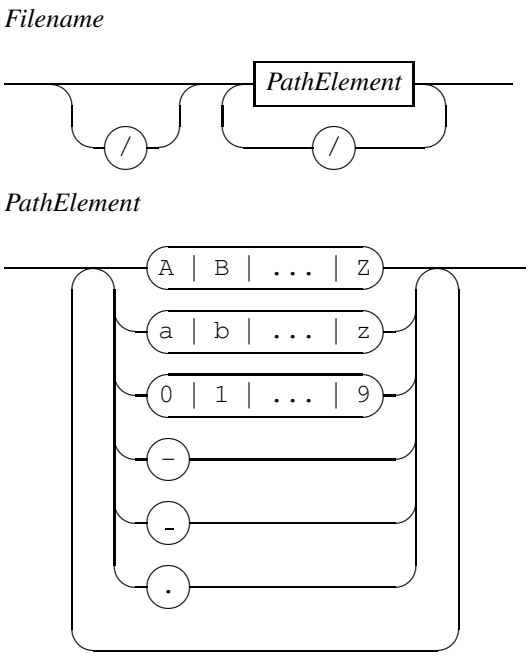

Figure 2.5: Raildiagram of *Filename*.

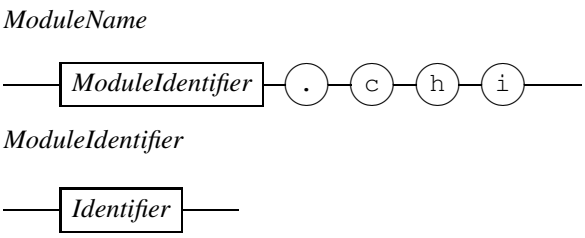

Figure 2.6: Raildiagram of *ModuleName*.

filename or *relative path* does not start from the root, but from the current working directory (refered to as '`.`') instead.

When a filename refers to a Chi file, the final path element **must** match with the *ModuleName* diagram shown in Figure **??**. The final path element must start with a *ModuleIdentifier* (which is the same as an *Identifier*[page 8]), and end with '`.chi`'. This requirement is necessary to allow Chi modules to be imported using the import statement[page 76].

Examples

Example	Explanation
`/opt/se/chi-1.0//standardlib.dast`	An absolute filename
`mymodel.chi`	A relative filename
`extension.py`	Filename of a Python program

□

2.2 White-space

White-space is a sequence of one or more constructs that is considered to be 'unimportant' at the context-free syntax level. In other words, arbitrary amounts of white space may be inserted between boxes in the diagrams at the context-free syntax level without affecting the meaning of the specification. Normally, white space is used to enhance readability of the source, by formatting (grouping) of pieces of code, or by (textual, informal) explanation of the code.

There are three forms of white space, namely white-space characters, line comment, and block comment. White-space characters are the SP character U0020, the TAB character U0009, the CR character U000D, and the LF character U000A. Often, sequences of such characters are encountered in a source file. Technically, each character is a seperate white-space construct. In practice, a sequence of white-space characters is normally considered to be one unit.

Comment in Chi exists in two forms, line comment and block comment. Comment is never interpreted by the tools, in particular, you cannot start or end a comment inside another comment. Line comment starts with two forward slash characters U002F. Everything behind the second slash character upto (but not including) the first LF character[2] is considered to be comment, and not interpreted by the tools. The second form of comment is the block comment construct. It starts with the character sequence forward slash and asterisk (U002F followed by U002A), and ends with the first occurence of the character sequence asterisk, forward slash (U002A followed by U002F) after the start sequence. In particular, the asterisk character of the start sequence may not be used as the first character of the end sequence (this case is shown as the last example below). All characters between the start sequence and the end sequence are not interpreted by the tools.

Examples

Example	Explanation
// line comment	Correct
/* block comment */	Correct
/*/	Incorrect, not a comment

□

[2] A special case is a line comment at the last line, where the last line is not terminated by a LF character. In that case, the line comment runs upto the end of the file.

Chapter 3
Types

The Chi language is a statically typed language, which means that all values and variables in a Chi specification have a single fixed type. In this chapter, the available types are explained.

The next chapter (about expressions) will introduce syntax to create and manipulate values of (most) types, thus allowing computations to be performed. Attaching a type to a variable is explained in Chapter 6,

The syntax of a type is defined by the *Type* diagram in Figure 3.1. There are four kinds of types, shown at Tracks 1 through 4. Basic[page 12] or elementary types[page 12] are defined in the *BasicType*[page 12] block, container types[page 12] (types whose values contain values of other types) are defined in the *ContainerType*[page 12] block, function types[page 14] (a type that has functions as value) are available from the *FunctionType*[page 14] block, and types for stochastic distributions[page 44] are available from the *DistributionType*[page 14] block. Finally, Track 5 allows grouping of types through the use of brackets.

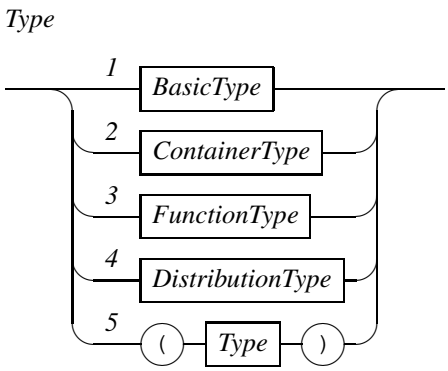

Figure 3.1: Raildiagram of *Type*.

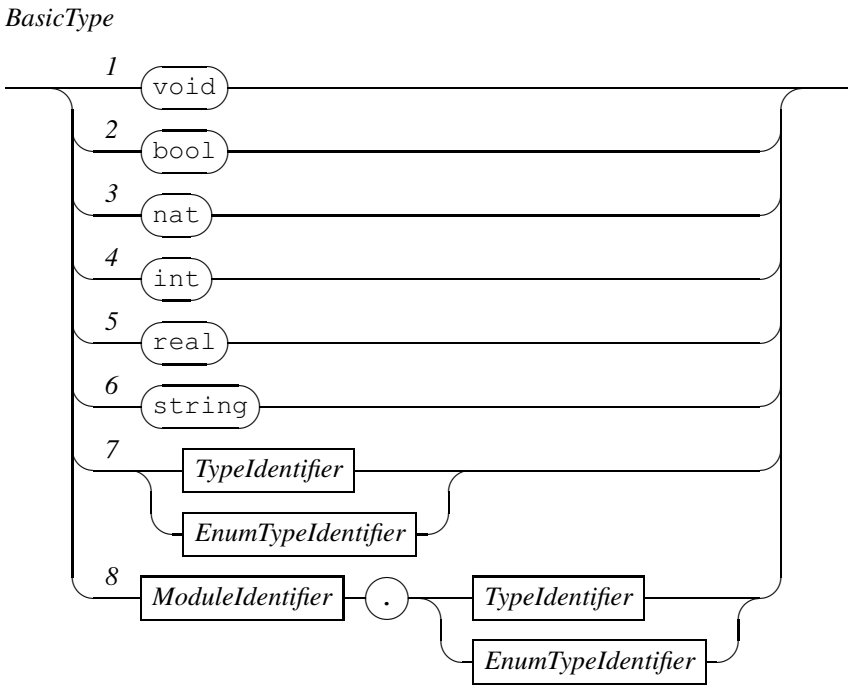

Figure 3.2: Raildiagram of *BasicType*.

3.1 Basic types

The *BasicType* diagram in Figure 3.2 shows the syntax of the basic types, also known as elementary types. They are expressed using a single keyword at Tracks 1 through 6, namely the empty *void type* (a type without any value), the booleans[page 23] (values of the *boolean type*), the natural numbers[page 24] (values of the *natural number type*, the set $\mathbb{N}$), the integer numbers[page 26] (values of the set *integer number type*, the set $\mathbb{Z}$), the real numbers[page 29] (values of the *real number type*, the set $\mathbb{R}$), and strings[page 32] (sequences of ASCII characters), values of the *string type*.

Track 7 is an identifier that refers to the name of a type. This may be either the name of an enumeration definition[page 74], or the name of a type definition[page 76]. Track 8 does the same, except that it refers to a type[page 11] imported[page 76][page 76] from another module.

3.2 Container types

A container type is a type whose values are often called *containers*, since it contains (many) values of another type. The latter values are called *elements*, and the latter type is called *element type*. There are different kinds of container types. They differ in the way they handle elements, in its own strong and weak points in giving access to the elements.

The syntax of container types available in Chi is shown in the *ContainerType* diagram in Figure 3.3. There are five container types in Chi. The first one at Track 1 is the *list type*, which allows an arbitrary number of elements to be stored in a sequential order. Access to elements

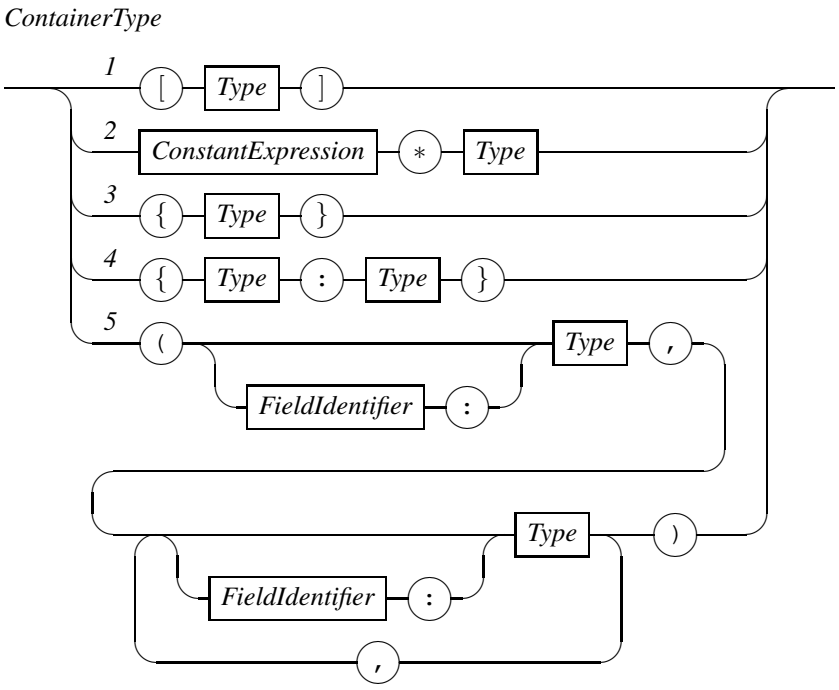

Figure 3.3: Raildiagram of *ContainerType*.

at the front and back of a *list*[page 34] (a value of type list type) is cheap (fast), access to elements at the middle is costly (slow) since you have to remove the elements in front of it first (or from the back, depending on where you start). In addition, you cannot modify an element stored in a list (you have to construct a new list with the modified element instead).

These disadvantages are adressed in the *vector type* shown at Track 2. The *Type*[page 11] block specifies the element type and the *ConstantExpression*[page 49] block denotes a constant[page 75] value of type `nat`[page 12] that defines the number of elements in the container (unlike the list type, the vector type has a fixed size).

The *set type* at Track 3 is useful when you (want to) have unique elements. A *set*[page 40] (a value of type set type) is an unordered container, there is no first or last element. The set type does however guarantee that each of its elements is unique (that is, the presence of each value of its element type is a boolean function, either the value is present exactly once or it is not).

The *dictionary type* shown at Track 4 is an extended form of the set type. It takes two types. The first type at the track is called the *key type*, the second type at the track is called the *value type*. A *dictionary* (a value of type dictionary type) treats its elements of the key type (commonly refered to as *keys*) in the same way as a set treats its elements, each value of the key type is either present in the dictionary once or it is not present. The extension with respect to sets is that for each key[page 13] (each value of the key type[page 13] present in the dictionary) an associated value of the value type[page 13] is available.

Last but not least in the container types is the *record tuple type*, shown at Track 5. Values of this type are commonly refered to as *record tuples*,[page 39] *tuples* or *records*. Its use is in merging a number of values (at least two) together and allow treatment of such a combination of values

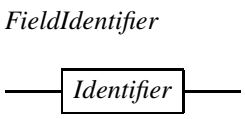

Figure 3.4: Raildiagram of *FieldIdentifier*.

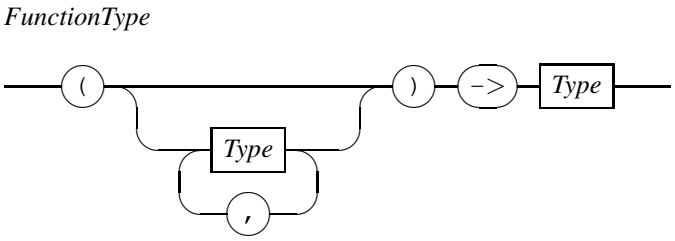

Figure 3.5: Raildiagram of *FunctionType*.

as a single entity. The unique capability of the record type is that each value may be of a different type. For each value in the record, its type must be specified. Access to a value in a record is by position, expressed as constant number. Since remembering the position of each value may be difficult if you have a lot of values in a record, you can optionally give a name to a value by prefixing its type with a *FieldIdentifier* and a colon. The syntax of a *FieldIdentifier* is shown in Figure 3.4. As you can see, it consists of an *Identifier*[page 8].

WARNING: *The dictionary type is not implemented yet.*

3.3 Function type

The next kind of type is the function type. This type has functions as its value, that is, you can save a function in a variable and use the variable to compute a function result. The syntax of the function type is shown in the *FunctionType* diagram in Figure 3.5. The syntax of a function type looks very similar to a header of a function definition[page 78] or declaration[page 80], except that only the type signature of the formal parameters[page 68] is given (that is, only the types of the formal parameters are stated rather than full variable declarations[page 67]).

3.4 Distribution type

The final type is the distribution type, the type used for storing stochastic distributions[page 44]. The syntax is shown in the *DistributionType* diagram in Figure 3.6. It consists of an arrow created from a minus sign and a bigger-than character followed by the element type (the type of values being drawn from the distribution).

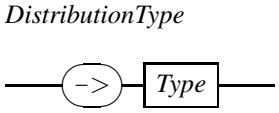

Figure 3.6: Raildiagram of *DistributionType*.

Chapter 4
Expressions

Expressions are the principle means to express computation of values in Chi. Since the language is rich in data types, expressions form a large part of the language.

To make the manual more valuable as a reference, this chapter is not organized on the syntax of expressions but on the data types available in the language. This results in operators being split over the data types they take as arguments, for example, the binary addition operator '+' is listed four times in this chapter, namely for the `nat` type, the `int` type, the `real` type, and the set type `{T}`. In addition, functions from the standard library (the `standardlib` module) are also added, thus creating a complete overview of available functionality with each data type.

Besides functionality for each data type, expressions in Chi also have a number of facilities available more than one data type. The facilities that come in the form of syntactical extensions are shown in the *BasicExpression*[page 17] diagram, explained in Section 4.1. The ability to combine several operators without having to write brackets is another facility, and is discussed in Section 4.17. Finally, some expressions are addressable (that is, they can be used as a destination to store value into), while others represent a pure value. This distinction is explained in more detail in Section 4.18.

The overal syntax of an expression is shown in the *Expression* diagram in Figure 4.1. The first track shows that you can surround an expression with round brackets to treat it as a single entity. This is particularly useful to override the default operator priorities[page 46] listed in Section 4.17. All other tracks of the *Expression*[page 17] diagram are shown and explained in more detail in the following sections.

4.1 Basic expressions

The basic expressions of the *BasicExpression* diagram are shown in Figure 4.2. With Track 1, named entities can be used in expressions. Most often they are variables, but this rule is also

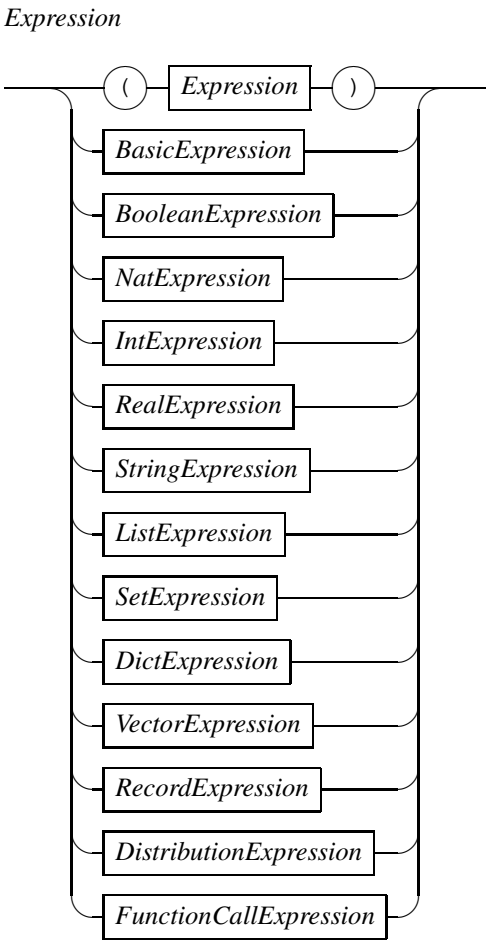

Figure 4.1: Raildiagram of *Expression*.

used for refering to functions, constants, and values of enumeration definitions. Track 2 does the same, except that the named entities are prefixed with the name of the module where they come from.

Track 3 shows the syntax of the unary derivative operator. This operator can only be applied to addressable (Section 4.18) continuous expressions of type `real`. Track 4 is the syntax for refering to the 'old value' of a variable (that is, the value of a variable just before assigning a new value to it). Track 5 shows the syntax of instantiating a templated function (using a *FunctionIdentifier*[page 78]). Finally, Track 6 shows the syntax of expression folding. Details of folding can be found in Section 4.3.

WARNING: *Module prefix of Track 2 and template instantiation of Track 5 are not implemented yet.*

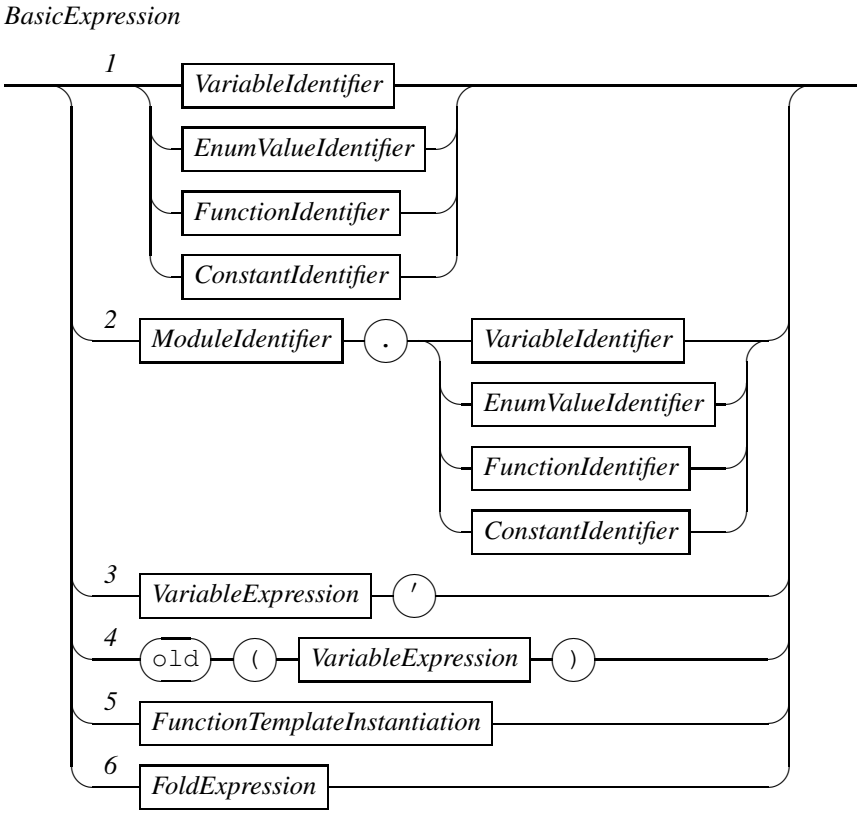

Figure 4.2: Raildiagram of *BasicExpression*.

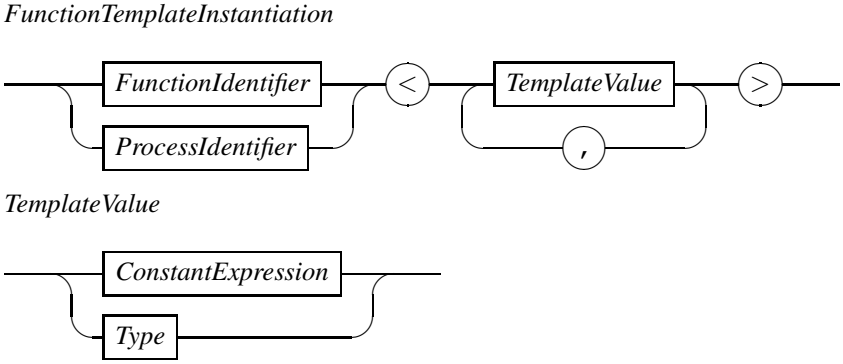

Figure 4.3: Raildiagram of *BasicExpression*.

4.2 Template instantiation

Template instantiation for explicit template parameters[page 84] gives concrete values to variables of template definitions[page 84] of templated functions and processes. It consists of the name of the templated function or process, followed by one or more *TemplateValue* blocks. The latter blocks are seperated from the identifier with triangular brackets. Since the template values are processed at compile time, they must be constant.

Examples

As an example, consider a templated process definition B parameterized with a maximal buffer size n and a type T of the buffered values.

```
proc B<n: nat, T: type>(chan a?, b!: T) =
|[ var x: T, xs: [T] = []
:: *( len(xs) < n -> a?x ; xs := xs ++ [x]
    | len(xs) > 0 -> b!hd(xs) ; xs := tl(xs)
    )
]|
```

This general definition can be instantiated for $n = 5$ and type $T =$ `lot` with input channel u and output channel v by the following process instantiation[page 57].

```
B<5, lot>(u, v)
```

□

WARNING: *Template instantiation for explicit parameters is not implemented yet.*

4.3 Expression folding

Folding of expressions is the process of iterating through values of a container (list, set, or vector) or a range of integral numbers, doing some processing on each value, and folding them into a new value. The syntax of a fold expression is shown in the *FoldExpression* diagram in Figure 4.4. As you can see in the diagram, the fold expression consists of four parts, a *FoldOperator*[page 22], an *ExpressionIterator*[page 20], a guard (the optional Track 1), and an *Expression*[page 17]. The container or range used to obtain values from is specified in the *ExpressionIterator* block, how to process a value is specified in the guard and the *Expression* block, and how to fold values together into a new value is defined by the *FoldOperator* block. Each of these steps is explained in more detail below.

4.3.1 Expression iterator

The *ExpressionIterator* diagram used to specify where values are obtained from, is shown in Figure 4.5. The folding starts with a variable denoted by a *VariableIdentifier*[page 68] which is assigned each value that must be processed. This variable can be used in the guard and the

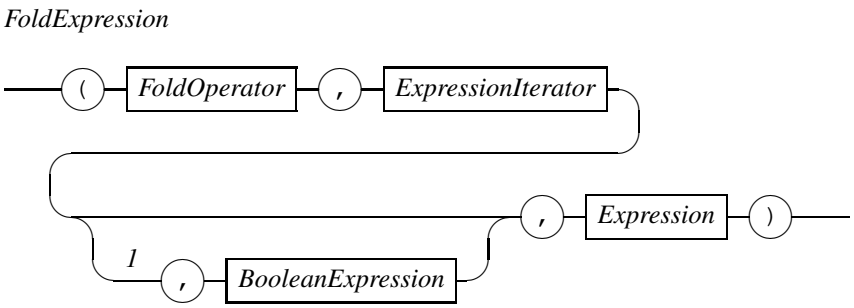

Figure 4.4: Raildiagram of *FoldExpression*.

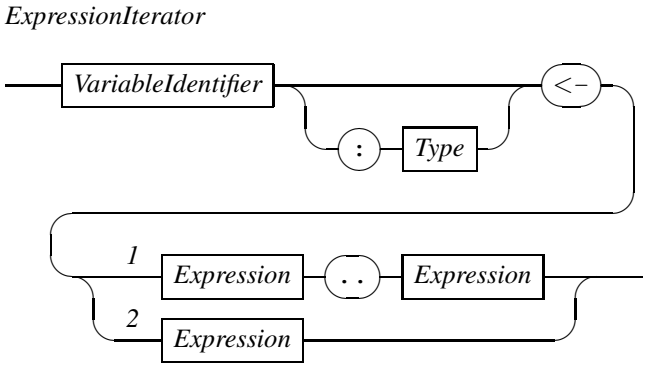

Figure 4.5: Raildiagram of *ExpressionIterator*.

Expression block as read-only variable to obtain the value being processed. After the identifier, you can optionally specify its type. If you do not specify its type, the compiler will compute it for you. The set of values being iterated over can be specified in two ways. The first way shown at Track 1 is to use a range similar to the iterator in statement folding[page 60] with the first expression denoting the lower bound, and the second expression denoting the upper bound. The values of both the lower bound and the upper bound are also processed (that is, the iteration is inclusive both bounds). The difference of a range in expression folding with respect to statement folding is that with expression folding, both expressions need not be constant, they may contain variables that get assigned a value during execution of the program. The second way of defining the values to iterate over is by means of stating a container (a value of a container type[page 12]), as shown at Track 2.

4.3.2 Processing

For processing values, the optional guard at Track 1 and the *Expression*[page 17] block (the third and fourth parts in the expression folding) are used. For each value obtained from the iterator, first the value of guard is computed. If it holds, the new value to be used for folding is computed using the *Expression* supplied as the fourth part.

If the guard consists of a *BooleanExpresssion* block, its value is computed. If it is true, the guard holds. If it is false, the guard does not hold, and the value is discarded. If the guard is empty (that is, Track 1 is bypassed), the guard always holds.

WARNING: *Support of the projection operator both in the guard and the Expression block is weak in the current implementation. As a result, you may get type errors that do not exist with the projection at this position.*

4.3.3 Folding

The values that are not discarded during processing are folded together using the *FoldOperator*, which is just one of seven possible binary operators in expressions. The table below lists them (first column).

Operator	Initial	Purpose
`+`	0	Add all values together
`*`	1	Multiply all values with each other
`and`	true	Test whether all values are true
`or`	false	Test whether (at least) one value is true
`max`	0 or $-\infty$	Obtain largest value
`min`	∞	Obtain smallest value
`++`	[]	Concatenate all values to a list
`\/`	{}	Collect all values into a set

The second column lists the initial value for each operator. For the `max` operator, the initial value for natural numbers is 0 and for the other numeric types, it is $-\infty$.

Examples

Example	Result
`(max, i <- 0..3, i)`	3, upper bound is also tried
`(max, i <- [0, 1, 2, 3], i)`	3, equivalent to previous example
`(max, i <- 0..3, i + 2)`	5, max is computed over $i + 2$
`(max, i <- 0..7, i < 4, i + 2)`	5, guard discards 4,5,6, and 7
`(and, i <- 8..7, i < 0)`	true, empty range (upper bound is smaller than lower bound)
`(*, i <- 1..5, i)`	$5! = 120$
`(++, x <- xs, x < 6, [ x*x ])`	$[1, 4, 25]$, if $xs = [1, 8, 2, 5]$
`(+, i <- 3..5, i*2)`	$((1 + 3 * 2) + 4 * 2) + 5 * 2 = 25$

□

In the final example, you can see the actual computation that is performed. The iterator first assigns value 3 to variable *i*. Since the guard is omitted, the value of $i * 2$ is computed, and added to the initial value of the '+' operator. Next, value 4 is assigned, causing $4 * 2$ to be added to the result of the previous addition. Finally, the same happens with value 5. Then the iterator is finished, and the result at that moment is returned as result of the fold expression.

Note: Some implementations may not have $-\infty$ or ∞ available, they will use an approximation instead.

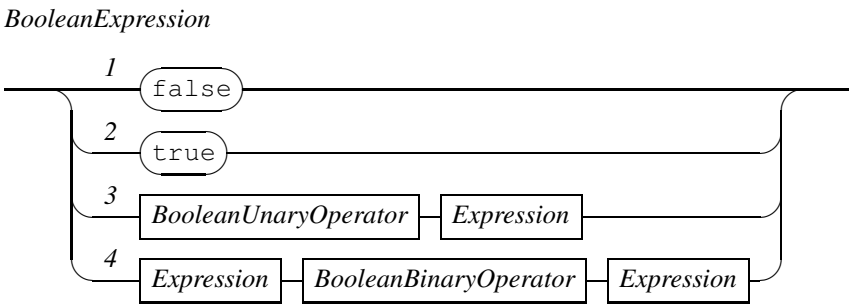

Figure 4.6: Raildiagram of *BooleanExpression*.

4.4 Boolean values

Values of the boolean data type[page 12] `bool` can be created and manipulated using the syntax of the *BooleanExpression* diagram in Figure 4.6. The type has two values, `false` and `true`, which may be entered literally in a Chi specification via Track 1 and 2. The other tracks manipulate existing boolean values using the rules of propositional logic. Through Track 3 the boolean unary expressions become available, explained further in Section 4.4.1. The syntax of the boolean binary expressions is shown at Track 4 and explained in Section 4.4.2.

4.4.1 Boolean unary expressions

The boolean unary expressions of Track 3 in the *BooleanExpression*[page 23] diagram are listed in the table below. In the first column, the contents of the *BooleanUnaryOperator* block is shown in fixed width font, the *Expression*[page 17] block is represented with *e*. The second column states the allowed type of expression *e*, and the third column states the type of the result after applying the operator. Finally, the fourth column explains the meaning of the operator.

Unary expression	Type *e*	Result type	Description
`not` *e*	`bool`	`bool`	$\neg e$, boolean negation operator

There is only one boolean unary operator, namely the `not` operator which inverts the value of its boolean argument.

Examples

Example	Result
`not true`	`false`

□

4.4.2 Boolean binary expressions

The boolean binary expressions of Track 4 in the *BooleanExpression*[page 23] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is

represented by the e_1 symbol, the *BooleanBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	Type e_1	Type e_2	Result type	Description
e_1 = e_2	`bool`	`bool`	`bool`	$e_1 = e_2$, equality test
e_1 /= e_2	`bool`	`bool`	`bool`	$e_1 \neq e_2$, inequality test
e_1 and e_2	`bool`	`bool`	`bool`	$e_1 \wedge e_2$, conjunction operator
e_1 or e_2	`bool`	`bool`	`bool`	$e_1 \vee e_2$, disjunction operator
e_1 => e_2	`bool`	`bool`	`bool`	$e_1 \rightarrow e_2$, implication operator ($\equiv \neg e_1 \vee e_2$)

The semantics of the boolean logic operators, also known as *propositional logic operators* are explained in books about logic, for example [1]. Below, the truth tables of the logic operators are listed for ease of reference.

e_1	e_2	$\neg e_1$	$e_1 = e_2$	$e_1 \neq e_2$	$e_1 \wedge e_2$	$e_1 \vee e_2$	$e_1 \rightarrow e_2$
`false`	`false`	`true`	`true`	`false`	`false`	`false`	`true`
`true`	`false`	`false`	`false`	`true`	`false`	`true`	`false`
`false`	`true`	`true`	`false`	`true`	`false`	`true`	`true`
`true`	`true`	`false`	`true`	`false`	`true`	`true`	`true`

4.4.3 Boolean functions

The only function in the standardlib for booleans is the conversion of a boolean value to its string representation. Its signature is listed below.

Function	Description
`func b2s(val b: bool) -> string`	Convert boolean value to string value

Examples

Example	Result
`b2s(true)`	`"true"`

In this example, the value `true` is converted to the string `"true"`.

□

4.5 Natural numbers

The natural numbers type is the Chi representation of the mathematical set $\mathbb{N}$, the non-negative numbers. The type of natural numbers is the natural numbers type,[page 12] written as the basic type `nat`.

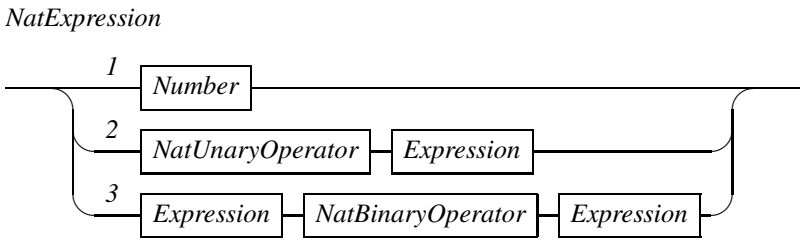

Figure 4.7: Raildiagram of *NatExpression*.

The syntax of natural numbers is shown in the *NatExpression* diagram in Figure 4.7. Track 1 states that literal natural number values are written as a *Number*[page 5] block. Tracks 2 and 3 show the unary and binary expressions on natural numbers. These operators are explained in Sections 4.5.1 and 4.5.2.

4.5.1 Natural number unary expressions

The unary expressions on natural numbers at Track 2 in the *NatExpression*[page 25] diagram are listed in the table below. In the first column, the contents of the *NatUnaryOperator* block is shown in fixed width font, the *Expression*[page 17] block is represented with *e*. The second column states the allowed type of expression *e*, and the third column states the type of the result after applying the operator. Finally, the fourth column explains the meaning of the operator.

Unary expression	**Type** *e*	**Result type**	**Description**
`-` *e*	`nat`	`int`	Unary negation operator, $-e$ as integer value
`+` *e*	`nat`	`int`	Unary addition operator, *e* as integer value

There are two unary operators on natural numbers. Both promote their value to the integer type (using these operators is the only way to obtain literal integer values). In addition, the first operator negates its argument.

Examples

Example	**Result**
`- 23`	Integer value -23

□

4.5.2 Natural number binary expressions

The binary expressions on natural numbers of Track 3 in the *NatExpression*[page 25] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *NatBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	Type e_1	Type e_2	Result type	Description
e_1 `^` e_2	`nat`	`nat`	`nat`	$e_1^{e_2}$
e_1 `*` e_2	`nat`	`nat`	`nat`	$e_1 \times e_2$
e_1 `/` e_2	`nat`	`nat`	`real`	Real division e_1/e_2
e_1 `div` e_2	`nat`	`nat`	`nat`	Integer division $e_1 \div e_2 \equiv \lfloor e_1/e_2 \rfloor$, the biggest integral number smaller or equal to e_1/e_2
e_1 `mod` e_2	`nat`	`nat`	`nat`	Integer remainder $e_1 \bmod e_2 \equiv e_1 - e_2 \times (e_1 \div e_2)$
e_1 `+` e_2	`nat`	`nat`	`nat`	$e_1 + e_2$
e_1 `-` e_2	`nat`	`nat`	`nat`	$e_1 - e_2$
e_1 `min` e_2	`nat`	`nat`	`nat`	$e_1 \min e_2$
e_1 `max` e_2	`nat`	`nat`	`nat`	$e_1 \max e_2$
e_1 `<` e_2	`nat`	`nat`	`bool`	$e_1 < e_2$
e_1 `<=` e_2	`nat`	`nat`	`bool`	$e_1 \leq e_2$
e_1 `=` e_2	`nat`	`nat`	`bool`	$e_1 = e_2$
e_1 `/=` e_2	`nat`	`nat`	`bool`	$e_1 \neq e_2$
e_1 `>=` e_2	`nat`	`nat`	`bool`	$e_1 \geq e_2$
e_1 `>` e_2	`nat`	`nat`	`bool`	$e_1 > e_2$

The subtraction e_1 `-` e_2 is defined only for $a \geq b$.

Examples

Example	Result
`7/4`	1.75
`7 div 4`	1
`7 mod 4`	3

□

The natural number type is the work horse of many specifications. As you can see from the list, all the usual mathematical operations can be performed.

4.5.3 Natural number functions

Function	Description
`func n2s(val n: nat) -> string`	Convert natural number value to string value

4.6 Integer numbers

The integer number type[page 12] contains all integral numbers, that is, all negative numbers, zero, and all positive numbers. In mathematics, this type is written as $\mathbb{Z}$. The syntax of integer numbers is shown in the *IntExpression* diagram in Figure 4.8. The literal integer numbers at Track 1 are a *Number*[page 5] prefixed with a '+' or a '-' sign, expressions with unary operators on integer numbers at Track 2 are described in Section 4.6.1, and the expressions with binary operators are shown in Section 4.6.2.

Note: Technically, the literal integer values as shown at Track 1 do not exist, they are a combi-

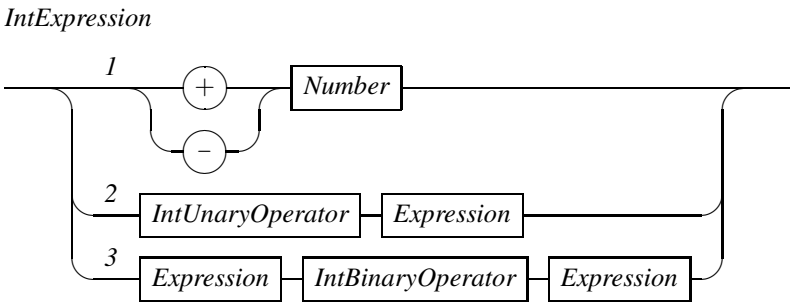

Figure 4.8: Raildiagram of *IntExpression*.

nation of a *IntUnaryOperator* and a literal natural number value.

4.6.1 Integer number unary expressions

The integer number unary expressions of Track 2 in the *IntUnaryOperator*[page 27] diagram are listed in the table below. In the first column, the contents of the *IntUnaryOperator* block is shown in fixed width font, the *Expression*[page 17] block is represented with *e*. The second column states the allowed type of expression *e*, and the third column states the type of the result after applying the operator. Finally, the fourth column explains the meaning of the operator.

Unary expression	Type *e*	Result type	Description
`-` *e*	`int`	`int`	$-e$, unary negation operator
`+` *e*	`int`	`int`	*e*, unary addition operator

The unary operators of the integer numbers are the same as the unary operators of the natural numbers[page 25]. The only difference is that the argument must of of type `int`.

4.6.2 Integer number binary expression

The binary expressions on integer numbers of Track 3 in the *IntExpression*[page 26] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *IntBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	Type e_1	Type e_2	Result type	Description
e_1 `^` e_2	`int` {`nat`, `int`}	`nat` `int`	`int` `real`	$e_1^{e_2}$
e_1 `*` e_2	{`nat`, `int`} `int`	`int` `nat`	`int` `int`	$e_1 \times e_2$
e_1 `/` e_2	{`nat`, `int`} `int`	`int` `nat`	`real` `real`	Real division e_1/e_2
e_1 `div` e_2	`int`	`int`	`int`	Integer division $e_1 \div e_2 \equiv \lfloor e_1/e_2 \rfloor$, the biggest integral number smaller or equal to e_1/e_2
e_1 `mod` e_2	`int`	`int`	`int`	Integer remainder $e_1 \bmod e_2 \equiv e_1 - e_2 \times (e_1 \div e_2)$
e_1 `min` e_2	`int`	`int`	`int`	$e_1 \min e_2$
e_1 `max` e_2	`int`	`int`	`int`	$e_1 \max e_2$
e_1 `+` e_2	{`nat`, `int`} `int`	`int` `nat`	`int` `int`	$e_1 + e_2$
e_1 `-` e_2	{`nat`, `int`} `int`	`int` `nat`	`int` `int`	$e_1 - e_2$
e_1 `<` e_2	{`nat`, `int`} `int`	`int` `nat`	`bool` `bool`	$e_1 < e_2$
e_1 `<=` e_2	{`nat`, `int`} `int`	`int` `nat`	`bool` `bool`	$e_1 \leq e_2$
e_1 `=` e_2	{`nat`, `int`} `int`	`int` `nat`	`bool` `bool`	$e_1 = e_2$
e_1 `/=` e_2	{`nat`, `int`} `int`	`int` `nat`	`bool` `bool`	$e_1 \neq e_2$
e_1 `>=` e_2	{`nat`, `int`} `int`	`int` `nat`	`bool` `bool`	$e_1 \geq e_2$
e_1 `>` e_2	{`nat`, `int`} `int`	`int` `nat`	`bool` `bool`	$e_1 > e_2$

The usual arithmetic operators can take a combination of a natural number argument and an integer number argument. The less used `div`, `mod`, `min`, and `max` operators require two integer number arguments to make extremely clear that it is an integer operation. Prefixing a natural number value with an unary `+` or `-` operator[page 25] will perform the conversion to the `int` type.

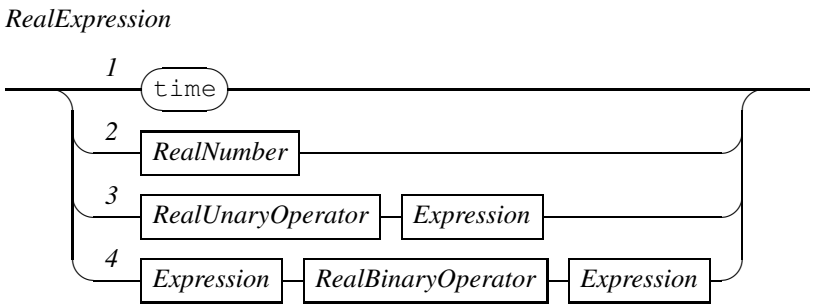

Figure 4.9: Raildiagram of *RealExpression*.

Examples

Example	Result
`+7 div +4`	+1
`+7 mod +4`	+3
`+7 div -4`	−2 ($\lfloor +7/-4 \rfloor = \lfloor -1.75 \rfloor = -2$)
`+7 mod -4`	−1 ($+7 - -4 \times (+7 \div -4) = +7 - -4 \times -2 = +7 - +8 = -1$)
`-7 div +4`	−2 ($\lfloor -7/+4 \rfloor = \lfloor -1.75 \rfloor = -2$)
`-7 mod +4`	+1 ($-7 - +4 \times (-7 \div +4) = -7 - +4 \times -2 = -7 - -8 = +1$)
`-7 div -4`	+1 ($\lfloor -7/-4 \rfloor = \lfloor +1.75 \rfloor = +1$)
`-7 mod -4`	−3 ($-7 - -4 \times (-7 \div -4) = -7 - -4 \times +1 = -7 - -4 = -3$)

Since confusion may exist in the result of applying `div` and `mod` operators on integer values, this example includes all cases of usage of the `div` and `mod` operators.

□

4.6.3 Integer number functions

Function	Description
`func n2i(val n: nat) -> int`	Convert natural number value to integer number value
`func i2n(val i: int) -> nat`	Convert integer number value to natural number value
`func i2s(val i: nat) -> string`	Convert integer number value to string value
`func abs(val i: int) -> int`	Return absolute value of integer number

4.7 Real numbers

The syntax of an expression with real numbers (value of type `real`)[page 12] is shown in diagram *RealExpression* in Figure 4.9. The `time` keyword at Track 1 gives the current time of the execution. For simulators, the current time is often an increasing number starting from 0.0. Other tools may use a different starting point for `time`. Literal real values can be introduced using the *RealNumber*[page 6] block at Track 2. Unary operators (`+` and `-`) can be applied on

real values using Track 3. More details about real expressions with unary operators can be found in Section 4.7.1. Finally, binary operators can be applied by using the syntax of Track 4. These expressions are further explained in Section 4.7.2.

4.7.1 Real number unary expressions

The unary expressions on real numbers at Track 3 in the *RealExpression*[page 29] diagram are listed in the table below. In the first column, the contents of the *RealUnaryOperator* block is shown in fixed width font, the *Expression*[page 17] block is represented with *e*. The second column states the allowed type of expression *e*, and the third column states the type of the result after applying the operator. Finally, the fourth column explains the meaning of the operator.

Unary expression	Type *e*	Result type	Description
`+` *e*	`real`	`real`	Unary addition
`-` *e*	`real`	`real`	Unary negation

4.7.2 Real number binary expressions

The binary expressions on natural numbers of Track 4 in the *RealExpression*[page 29] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *RealBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	Type e_1	Type e_2	Result type	Description
e_1 `^` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1^{e_2}$
e_1 `*` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 * e_2$
e_1 `/` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	Real division e_1/e_2
e_1 `+` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 + e_2$
e_1 `-` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 - e_2$
e_1 `min` e_2	`real`	`real`	`real`	$e_1 \min e_2$
e_1 `max` e_2	`real`	`real`	`real`	$e_1 \max e_2$
e_1 `<` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 < e_2$
e_1 `<=` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 \leq e_2$
e_1 `=` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 = e_2$
e_1 `/=` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 \neq e_2$
e_1 `>=` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 \geq e_2$
e_1 `>` e_2	{`nat`, `int`, `real`} `real`	`real` {`nat`, `int`}	`real` `real`	$e_1 > e_2$

The binary `div` and `mod` operators do not exist for arguments of type `real`. Instead, for computing the (integer) division of real arguments, the `floor` function may be used and the remainder of real arguments can be computed using the `rmod` function.

4.7.3 Real number functions

In the **standardlib** module, the following functions exist for values of type `real`.

4.7.3.1 Conversion functions

Function	Description
`func ceil(val r: real) -> int`	The ceiling $\lceil r \rceil$, the smallest integer value not less than r
`func floor(val r: real) -> int`	The floor $\lfloor r \rfloor$, the biggest integer value smaller than or equal to r
`func round(val r: real) -> int`	Round r to the nearest integer value
`func n2r(val n: nat) -> real`	Convert natural number value n to value of type `real`
`func i2r(val i: int) -> real`	Convert integer number value i to value of type `real`
`func r2s(val r: real) -> string`	Convert r to its string representation
`func r2s(val r: real, n: nat) -> string`	Convert r to its string representation, with n characters

Examples

Example	Result
`ceil`(3.2)	+4
`floor`(3.2)	+3

□

4.7.3.2 Geometric functions

Function	Description
`func sin(val r: real) -> real`	sin(r)
`func cos(val r: real) -> real`	cos(r)
`func tan(val r: real) -> real`	tan(r)
`func asin(val r: real) -> real`	acos(r)
`func acos(val r: real) -> real`	asin(r)
`func atan(val r: real) -> real`	atan(r)

4.7.3.3 Hyperbolic functions

Function	Description
`func sinh(val r: real) -> real`	sinh(r)
`func cosh(val r: real) -> real`	cosh(r)
`func tanh(val r: real) -> real`	tanh(r)
`func asinh(val r: real) -> real`	acosh(r)
`func acosh(val r: real) -> real`	asinh(r)
`func atanh(val r: real) -> real`	atanh(r)

4.7.3.4 Other math functions

Function	Description
`func abs(val r:real) -> real`	$\|r\|$, the absolute value of real number `r`
`func exp(val r: real) -> real`	e^r
`func ln(val r: real) -> real`	$\ln$ `r`
`func log(val r: real) -> real`	$\log_{10}$ `r`
`func sqrt(val r: real) -> real`	$\sqrt{r}$
`func cbrt(val r: real) -> real`	$r^{\frac{1}{3}}$
`func rmod(val r, s: real) -> real`	$r \bmod s \equiv r - s \times \lfloor r/s \rfloor$

4.8 Strings

Strings (value of type `string`[page 12]) are sequences of characters. They are mainly used to add text to the output of the model in print statements. The syntax of string expressions is shown in the *StringExpression* diagram in Figure 4.10. A literal string[page 7] is written using a *StringLiteral*[page 7] block at Track 1. At Track 2 expressions with binary operators on values of type `string` are introduced. These are explained in the next section.

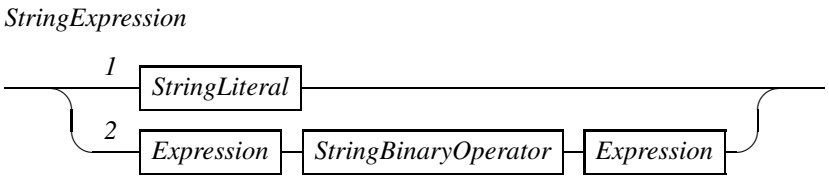

Figure 4.10: Raildiagram of *StringExpression*.

4.8.1 String binary expressions

The binary expressions on strings of Track 2 in the *StringExpression*[page 32] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *StringBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	**Type e_1**	**Type e_2**	**Result type**	**Description**
e_1 `min` e_2	`string`	`string`	`string`	$e_1 \min e_2$
e_1 `max` e_2	`string`	`string`	`string`	$e_1 \max e_2$
$e_1 < e_2$	`string`	`string`	`bool`	$e_1 < e_2$
e_1 `<=` e_2	`string`	`string`	`bool`	$e_1 \leq e_2$
$e_1 = e_2$	`string`	`string`	`bool`	$e_1 = e_2$
e_1 `/=` e_2	`string`	`string`	`bool`	$e_1 \neq e_2$
e_1 `>=` e_2	`string`	`string`	`bool`	$e_1 \geq e_2$
$e_1 > e_2$	`string`	`string`	`bool`	$e_1 > e_2$
e_1 `++` e_2	`string`	`string`	`string`	String concatenation

As you can see, string values can be compared with each other. Lexograpihcally ordering based on the ASCII characters set is used to decide order of strings.

Examples

Example	**Description**
`"abc" min "bdf"`	`"abc"`, since it is lexographically the smallest string
`"abc" ++ "bdf"`	`"abcbdf"`

□

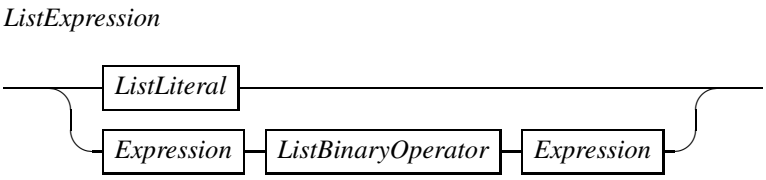

Figure 4.11: Raildiagram of *ListExpression*.

4.8.2 String functions

4.8.2.1 Conversion functions

Function	Description
`s2b(val s: string) -> bool`	Convert string with a boolean literal to its equivalent value
`s2n(val s: string) -> nat`	Convert string with a natural number literal to its equivalent value
`s2i(val s: string) -> int`	Convert string with an integer number literal to its equivalent value
`s2r(val s: string) -> real`	Convert string with a real number literal to its equivalent value

4.8.2.2 Other string functions

Function	Description
`nl() -> string`	Returns string containing NL character (ASCII value 10, U000A), *obsolete, use* `\n` *instead*
`tab() -> string`	Returns string containing TAB character (ASCII value 9, U0009), *obsolete, use* `\t` *instead*
`dquote() -> string`	Returns string containing double quote character (ASCII value 34, U0022), *obsolete, use* `\"` *instead*
`len(val s: string) -> nat`	Returns the number of characters in the string
`take(val s: string, n: nat) -> string`	Returns the first up to `n` characters of string `s`
`drop(val s: string, n: nat) -> string`	Returns string `s`, except for the first up to `n` characters

4.9 Lists

Lists are values of the list container type[page 12]. Syntax of lists is shown in Figure 4.11. A list value is either a literal list or it is an expression with a binary operator on lists. The syntax of a literal list is expressed in the *ListLiteral* diagram in Figure 4.12. List expressions with a binary operator are explained in Section 4.9.1. A literal list is zero or more, comma-seperated expressions between square brackets. Each expression must have the same type (called α

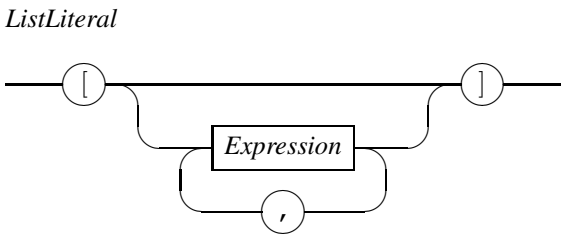

Figure 4.12: Raildiagram of *ListLiteral*.

here).

Examples

Input	Description
`[]`	The empty list value for all types of lists (that is, for a list containing any element type α)
`[1, 2, 7, 2]`	A list of natural numbers containing natural number values 1, 2, 7, and 2 (in that order)

□

4.9.1 List binary expressions

The binary expressions on lists of the bottom track in the *ListExpression*[page 34] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *ListBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	Type e_1	Type e_2	Result type	Description
e_1 `in` e_2	α	$[\alpha]$	`bool`	Element test on lists
e_1 `++` e_2	$[\alpha]$	$[\alpha]$	$[\alpha]$	List concatenation
e_1 `--` e_2	$[\alpha]$ $[\alpha]$	$[\alpha]$ $\{\alpha\}$	$[\alpha]$ $[\alpha]$	Subtract list e_2 from list e_1 Subtract set e_2 from list e_1
e_1 `=` e_2	$[\alpha]$	$[\alpha]$	$[\alpha]$	Equality test of lists e_1 and e_2
e_1 `/=` e_2	$[\alpha]$	$[\alpha]$	$[\alpha]$	In-equality test of lists e_1 and e_2

In this table, α represents any type (except `void`).

The list subtraction operator is similar to the difference[page 42] operator in sets. They differ in how element values are removed from the left argument. The basic idea of the list subtraction operator is for each element value of the right argument, the first corresponding value from the left argument is removed. More precisely, e_1 `--` e_2 with $e_1 = [x_1, x_2, \ldots, x_n]$ and $e_2 = [y_1, y_2, \ldots, y_m]$ is defined as

1. If $m = 0$ (that is, the case e_1--[ ]), e_1 is returned unmodified.

2. If $m > 1$, the right argument is split, its return value is the result of $(e_1 -- [y_1]) -- [y_2, y_3, \ldots, y_m]$.
3. If y_1 does not occur in e_1 (ie $m = 1 \wedge \forall x_i : x_i \mathrel{/=} y_1$ for $1 \leq i \leq n$), the left argument e_1 is returned unmodified.
4. Finally, if y_1 matches with x_j for the first time (ie $m = 1 \wedge x_j = y_1 \wedge \forall x_i : x_i \mathrel{/=} y_1$ for $1 \leq i < j$), x_j is removed from the result (ie $[x_1, x_2, \ldots, x_{j-1}, x_{j+1}, \ldots x_n]$ is returned).

Examples

Example	Result
`+3 in [+1, -4]`	false, +3 is not available in the list
`[1] ++ [5, 6]`	`[1, 5, 6]`
`[1, 2, 3] -- [4]`	`[1, 2, 3]`
`[1, 2, 3] -- [3]`	`[1, 2]`
`[1, 2, 3, 2] -- [2, 1]`	`[3, 2]` (= ([1, 2, 3, 2]--[2])--[1] = [1, 3, 2]--[1] = [3, 2])
`[1, 2, 3, 2] -- {2, 1}`	`[3, 2]`

□

4.9.2 Functions

4.9.2.1 Conversion functions

Function	Description
`func l2s[T: type](val xs: [T]) -> {T}`	Convert list *xs* to a set

WARNING: *The list to set function* `l2s` *has not been implemented. You can use expression folding*[page 20] *instead.*

4.9.2.2 Other list functions

Function	Description
`func len[T: type](val xs: [T]) -> nat`	Number of elements of list `xs`
`func hd[T: type](val xs: [T]) -> T`	First element of non-empty list `xs`
`func tl[T: type](val xs: [T]) -> [T]`	List `xs` except for the first element (`xs` must be non-empty)
`func hr[T: type](val xs: [T]) -> T`	Last element of non-empty list `xs`
`func tr[T: type](val xs: [T]) -> [T]`	List `xs`, except for the last element (`xs` must be non-empty)
`func take[T: type](val xs: [T], n: nat) -> [T]`	First upto `n` elements of list `xs`
`func drop[T: type](val xs: [T], n: nat) -> [T]`	List `xs`, except for the first upto `n` elements
`func sort[T: type](val xs: [T], f: (T,T) -> bool) -> [T]`	Return a sorted version of list `xs` using predicate function `f` (with `f`$(x, y) = x < y$)
`func insert[T: type](val xs: [T], x: T, f: (T,T) -> bool) -> [T]`	Insert element `x` into sorted list `xs` using predicate function `f`, (with `f`$(x, y) = x < y$)

Examples

Below, an example of using `sort` and `insert` functions is provided.

```
from standardlib import sort, insert

func cmp(val a,b: nat) -> bool = |[ ret a < b ]|

model M() =
|[ var xs: [nat] = [52, 79, 45, 18, 93, 85, 31, 67, 84, 45]
:: xs := sort(xs, cmp)
 ; !! xs, "\n"
 ; xs := insert(xs, 53, cmp)
 ; !! xs, "\n"
]|
```

After importing the `sort` and `insert` functions from the `standardlib` module, first the compare function `cmp` is defined that compares two arbitrary elements from the list and returns true iff the first argument is strictly smaller than the second argument.

The model definition `M` demonstrates use of the sort function and insert function. Note that the compare function is given to both functions as a value of type function (that is, `sort(xs, cmp)` and *not* `sort(xs, cmp())`). See Section 4.16 for more details about this use of functions.

□

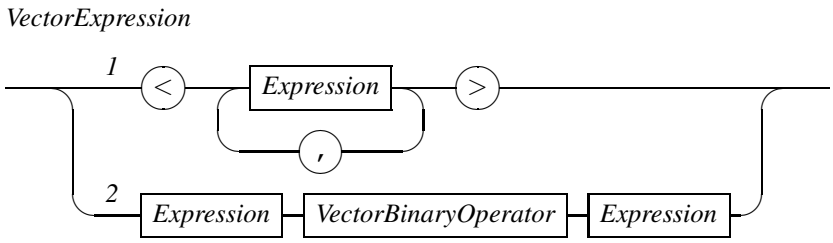

Figure 4.13: Raildiagram of *VectorExpression*.

4.10 Vectors

Vectors or arrays allow a fixed number of values of the same type to be stored together. The data type is intended primarily for replication, you want to keep a number of instances of the same thing together, for example a number of buffers. Each value in the vector can be accessed quickly, making the vector ideal for individual manipulation of each value in the vector. The syntax of vector expressions is shown in the *VectorExpression* diagram in Figure 4.13. The syntax of literal vector values is shown at Track 1. It is a comma-seperated list of expressions each of the same type, surrounded by two angular brackets. The type of such a literal vector is written as n*α, where n ($n \geq 1$) is the number of expressions between the angular brackets and α is the type of the expressions.

Examples

Example	Description
`< 1, 2, 3+18 >`	Vector of three natural numbers ie `3*nat`
`<false>`	Vector of `1*bool`

□

Binary expressions that can be used to get a vector value, shown at Track 2, are explained below.

4.10.1 Vector binary expressions

The binary expressions on vectors of Track 2 in the *VectorExpression*[page 38] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *VectorBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	Type e_1	Type e_2	Result type	Description
e_1 `.` e_2	n*α	`nat`	α	Projection into a vector
e_1 `=` e_2	n*α	n*α	`bool`	Equality test of a vector
e_1 `/=` e_2	n*α	n*α	`bool`	Inequality test of a vector

The projection operator is the primary operator for accessing the contents of a vector. At the

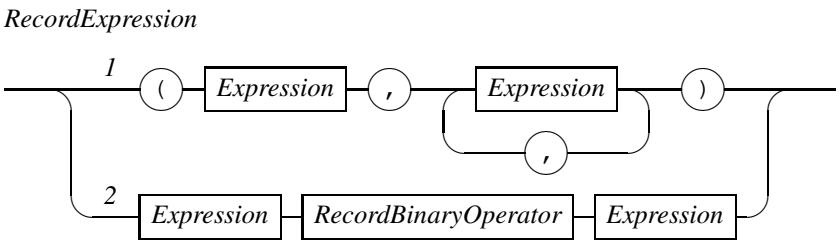

Figure 4.14: Raildiagram of *RecordExpression*.

left of the projection operator the vector to access should be the vector to access, at the right of the operator should be a natural number value indicating the field to access. The first field is accessed with natural number value 0, the second field with value 1, and so on. The natural number value used for indexing may be dynamic, that is, be computed at run time. This makes it easy to iterate over a vector.

Equality (and inequality) tests of vectors occur at a field-by-field basis, that is, two vectors are equal iff all their corresponding fields are equal.

Examples

Example	Description
`<1, 2, 3+18> . 2` `<1.0, 3.0> = <1.0, 5.0>`	Access third field (21) of vector false, since 3.0 /= 5.0

□

There are no functions in the standardlib module for vectors.

4.11 Record tuples

Record tuples or records are used to keep a set of related but (logically) different values together, for example the lowest, the highest and the average value of some computation. The syntax of a literal record tuple value is shown at Track 1 of the *RecordExpression* diagram in Figure 4.14. It consists of two or more expression, seperated from each other with comma's and surrounded by a pair of parenthesis. Since the type of each value in a record tuple can be different, the type of a record is described by listing the type of each value explicitly. The second track states the syntax of expressions with binary operators of record tuples. These are explained in more detail in the following section.

Examples

Example	Description
`(1, "data", 15, true)`	Literal record tuple (of type `(nat, string, nat, bool)`).

□

4.11.1 Record binary expressions

The binary expressions on records of Track 2 in the *RecordExpression*[page 39] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *RecordBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	Type e_1	Type e_2	Result type	Description
e_1 `.` e_2	$(t_1, t_2, \ldots, t_n)$ $(\alpha_1, \alpha_2, \ldots, \alpha_n)$	`nat` identifier	t_i α_i	Projection into record tuple e_1 with index e_2
e_1 `=` e_2	$(\alpha_1, \alpha_2, \ldots, \alpha_n)$	$(\alpha_1, \alpha_2, \ldots, \alpha_n)$	`bool`	Equality test
e_1 `/=` e_2	$(\alpha_1, \alpha_2, \ldots, \alpha_n)$	$(\alpha_1, \alpha_2, \ldots, \alpha_n)$	`bool`	Inequality test

The operators of a record tuple are almost the same as the operators of a vector, namely projection, equality test, and inequality test. The differences are in the use of a record type in the argument type with $\alpha_1, \alpha_2, \ldots, \alpha_n$ the type of the first, second, $\ldots n-1^{\text{th}}$ field for a record tuple of length n and $n \geq 2$. The other difference is that the natural number value at the right of the projection operator must be a fixed (static) value. Finally, instead of using a number to index a field, you can also use its name if you specified it in the type definition of the record tuple.

Examples

Example	Description
`(1, "data", 15, true) . 0` `x.data := []`	1, accessing first field with '`var x:(nat, data:[real])`', set the second field of the record to the empty list

□

There are no functions in the standardlib module for records.

4.12 Sets

Sets keep collection of values of one data type. For each value, it is recorded whether or not the value is present in the set. For this reason, each value can be present at most once in

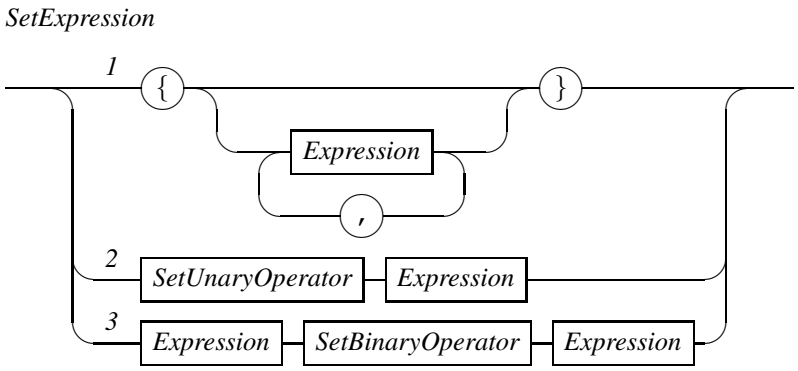

Figure 4.15: Raildiagram of *SetExpression*.

a set. Also, there is no order in a set, that is there is no first, second, or last element of a set. The syntax of set expressions is shown in the *SetExpression* diagram in Figure 4.15. At Track 1 of the diagram, the syntax of a set literal value is shown, it is a (possibly empty) list of expression seperated by comma's, surrounded by curly brackets. Each expression normally results in a different value of the same type. Expressions that result in a value already present in the set are silently ignored. At Track 2 the syntax of expressions with the unary set operator is shown. It is explained in more detail in Section 4.12.1. Finally, the syntax of expressions with the binary set operators is defined at Track 3. They are explained in Section 4.12.2.

Examples

Example	Description
`{}`	The empty set (set without any values present)

`{1, 2, 4}`
`{2, 4, 1}`
`{2, 2, 1, 4}`
	Set with values 1, 2, and 4 present

□

4.12.1 Set unary expressions

There is one unary operator for sets, namely `pick`. Expressions with this operator are shown in the table below. In the first column, the contents of the *SetUnaryOperator* block is shown in fixed width font, the *Expression*[page 17] block is represented with *e*. The second column states the allowed type of expression *e*, and the third column states the type of the result after applying the operator. Finally, the fourth column explains the meaning of the operator.

Unary expression	Type e	Result type	Description
`pick` e	$\{\alpha\}$	α	Pick an element

The `pick` operator is the only way to obtain an element from a non-empty set. The value returned by the operator is present in the set (that is, (`pick` S) $\in S$ holds for all non-empty sets S), but which value is returned is not fixed and may differ between implementations or between invocations of the operator. The set S being picked is not modified.

4.12.2 Set binary expressions

The binary expressions on sets of Track 3 in the *SetExpression*[page 41] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *SetBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	**Type e_1**	**Type e_2**	**Result type**	**Description**
e_1 `\/` e_2	$\{\alpha\}$	$\{\alpha\}$	$\{\alpha\}$	Union operator, $e_1 \cup e_2$
e_1 `/\` e_2	$\{\alpha\}$	$\{\alpha\}$	$\{\alpha\}$	Intersection operator, $e_1 \cap e_2$
e_1 `-` e_2	$\{\alpha\}$	$\{\alpha\}$	$\{\alpha\}$	Set-difference operator, $e_1 \setminus e_2$
e_1 `sub` e_2	$\{\alpha\}$	$\{\alpha\}$	`bool`	Sub-set operator, $e_1 \subseteq e_2$
e_1 `in` e_2	α	$\{\alpha\}$	`bool`	Element-test operator, $e_1 \in e_2$
e_1 `=` e_2	$\{\alpha\}$	$\{\alpha\}$	`bool`	Equality test of sets
e_1 `/=` e_2	$\{\alpha\}$	$\{\alpha\}$	`bool`	Inequality test of sets

The union operator merges both its arguments much like the or operator on booleans, a value is in the resulting set if it is in the left argument, in the right argument, or both. The intersection operator works much like the and operator on booleans, a value is in the resulting set if it is both in the left and in the right argument. Set difference works much like a subtraction, except that subtracting elements that are not available has no effect. A value is in the resulting set if it was in the left argument and not in the right argument. The sub-set operator returns true if the left argument is a subset or equal to the right argument, that is, if all elements of the left argument are also present in the set of the right argument. The element-test operator tests whether the value at the left of the operator is an element of the set at the right of the operator. Finally, the equality (and inequality) operators compare sets and return whether or not both sets are (not) equal (have the same collection of values present).

Examples

Example	**Description**
`{1, 2} \/ {2, 3}`	$\{1, 2, 3\}$, second 2 ignored
`{1, 2} /\ {2, 3}`	$\{2\}$, only one common value
`{1, 2} - {2, 3}`	$\{1\}$
`{1, 2} sub {2, 3}` `{ 2} sub {2, 3}`	false, 1 is not in the set at the right side true
`{ 2, 3} sub {2, 3}` `{ } sub {}`	true, all elements at the left are also at the right
`1 in {1, 2}` `5 in {1, 2}`	true, 1 is available in the set false, 5 is not available in the set

□

4.12.3 Set functions

There is one function for sets in the standardlib module, namely the `size` function to obtain the number of elements in a set. Its definition is shown below.

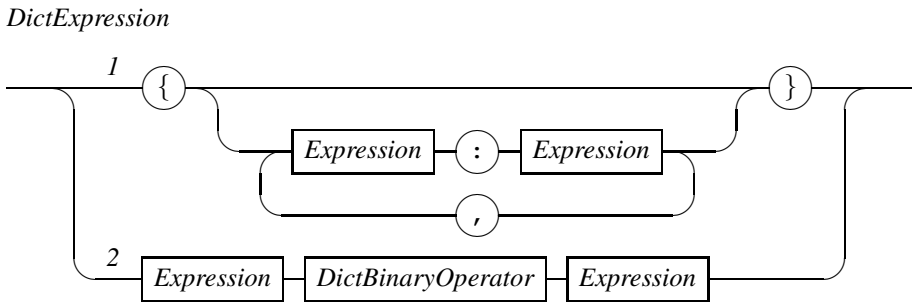

Figure 4.16: Raildiagram of *DictExpression*.

Function	Description
`func size[T: type](val xr: {T}) -> nat`	Number of elements of set `xr`

4.13 Dictionaries

Dictionaries are like sets,[page 40] but with each value in the dictionary you can store another value. Both values need not be of the same type. To prevent confusion, the former values are called 'keys', and the latter values are simply called 'values'. The type of the keys is called the key type, and the type of values is called the value type. If the key type is K and the value type is V, the type of the dictionary is $\{K : V\}$.

The syntax of dictionary expressions is shown in the *DictExpression* diagram in Figure 4.16. At Track 1 a literal dictionary value is shown. One element consists of two Expressions *Expression*[page 17] seperated by a colon. The first expression represents the key of the element, the second expression represents the value of the element. Elements are seperated from each other by a comma. All elements are between two curly brackets. At Track 2 the syntax of expressions with binary operators of dictionaries are shown. The available operators are shown in the next section.

The association of key and value, combined with fast access to a key makes dictionaries ideal for looking up values based on key values.

4.13.1 Dictionary binary expressions

The binary expressions on dictionaries of Track 2 in the *DictExpression*[page 43] diagram are listed in the table below. In the first column, the contents of the first *Expression*[page 17] block is represented by the e_1 symbol, the *DictBinaryOperator* contents is shown in fixed width font, and the second *Expression*[page 17] block is represented with e_2. The second column states the allowed type of expression e_1, the third column states the allowed type of expression e_2, and the fourth column lists the type of the result after applying the operator. Finally, the fifth column explains the meaning of the operator.

Binary expression	Type e_1	Type e_2	Result type	Description
e_1 `.` e_2	$\{\alpha : \beta\}$	α	β	Projection operator

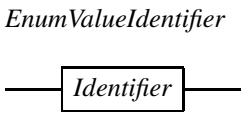

Figure 4.17: Raildiagram of *EnumValueIdentifier*.

4.13.2 Dictionary functions

Currently, there are no functions for dictionaries available.

WARNING: *Dictionaries are not yet implemented.*

4.14 Enumeration values

An enumeration value is an *EnumValueIdentifier* shown in Figure 4.17. It is an identifier defined in an enumeration definition[page 74]. The only operators allowed on these values is equality and inequality tests shown in the following table.

Binary operator	Type e_1	Type e_2	Result type	Description
e_1 = e_2	*E*	*E*	*E*	Equality test of two enumeration values
e_1 /= e_2	*E*	*E*	*E*	Inequality test of two enumeration values

Type *E* in the above table is an enumeration type[page 74].

Examples

With the following enumeration definition[page 74] for defining an enumeration type[page 74] `flagcolors` with values[page 44] `red`, `white`, and `blue`, the following expressions can be written:

Example	Description
`red = white`	false
`blue /= red`	true

□

WARNING: *Enumeration values are not yet implemented.*

4.15 Distributions

A distribution is used to obtain stochastic behavior. You cannot write a literal value for a variable of a distribution type[page 14]. Instead, you must use one of the distribution functions from the standardlib module listed in Appendix A.

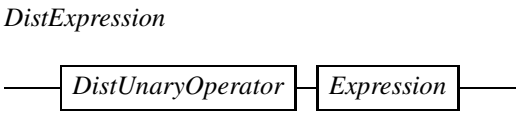

Figure 4.18: Raildiagram of *DistExpression*.

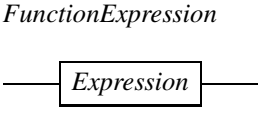

Figure 4.19: Raildiagram of *FunctionExpression*.

4.15.1 Unary distribution expressions

There is one unary operator for distributions, namely `sample`. Its syntax is shown in the *DistExpression* diagram in Figure 4.18. In the first column of the table below, the contents of the *DistUnaryOperator* block is shown in fixed width font, the *Expression*[page 17] block is represented with *e*. The second column states the allowed type of expression *e*, and the third column states the type of the result after applying the operator. Finally, the fourth column explains the meaning of the operator.

Unary expression	Type *e*	Result type	Description

`sample` *e*	`-> bool`
	`-> nat`
	`-> int`
	`-> real`
		`bool`
		`nat`
		`int`
		`real`
			Draw a sample of distribution *e*

4.16 Functions

A function can be declared[page 80] or defined[page 78] using the `func` keyword, as explained in Chapter 7. Besides calling a function (also known as function application), you can also use a function as if it is data, and assign it to a variable, give it as actual parameter to another function (which results in so-called higher order functions such as `sort`[page 37] or `insert`[page 37]) or send a function over a channel.

Before discussing function calls, first the *FunctionExpression* block is explained. The diagram of it is shown in Figure 4.19. A *FunctionExpression* is an expression, with the additional requirement that the type of the expression must be a function type[page 14].

The most common use of a *FunctionExpression* is to call it. The syntax of such a call is shown in the *FunctionCallExpression* diagram in Figure 4.20. A function call starts with a *FunctionExpression*[page 45] that states which function should be called. After it, between round brackets, the actual parameters of the call are listed. The result of a function call is a value of the type stated by the first (function) expression.

Function expressions can also be treated as data. Variables or parameters of the function type can be assigned a function (through normal assignment[page 52], formal parameter[page 68] in

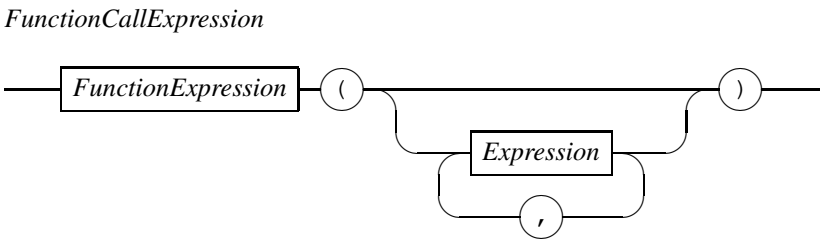

Figure 4.20: Raildiagram of *FunctionCallExpression*.

a function call, or the receiving end of a communication[page 54]). Once assigned a value, the function can be called through the variable as if the variable is a normal function name.

Examples

```
func f(val x: nat) -> nat = |[ ret 1 + x ]|

model P() =
|[ var p: (nat) -> nat
     , fv, pv: nat
:: p := f
 ; fv := f(1)
 ; pv := p(2)
 ; !! fv, "\n", pv, "\n"
]|
```

The above Chi specification defines a simple function `f` which returns its argument after incrementing it by one.

In the model, variable `p` has a function type[page 14]. The `p := f` assignment assigns the **function** to the variable. After the assignment you can still call the function directly, as is demonstrated by the `fv := f(1)` statement, but you can also call it through variable `p` as demonstrated in the `pv := p(2)` statement. (Note how variable `p` is treated as if it is a normal function name.) Finally, both function-call results are printed.

□

4.17 Expression operator priorities

Until now, it is assumed that only one operator is used in an expression. In real programs however, operators are normally used more often, or different operators are combined with each other. The question is what operator is applied in what order while computing the value of an expression. The answer to those questions is generally known as *expression operator priority*,

Examples

As an illustration, consider the possible meaning of the following expressions

Expression	Possible meaning
$1 + 2 * 3$	'$(1 + 2) * 3$' or '$1 + (2 * 3)$'
$10 - 2 - 3$	'$(10 - 2) - 3$' or '$10 - (2 - 3)$'
$\{1, 2, 3\} - \{1, 2\} - \{1\}$	'$(\{1, 2, 3\} - \{1, 2\}) - \{1\}$' or '$\{1, 2, 3\} - (\{1, 2\} - \{1\})$'
$\{1, 2, 3\} - \{1, 2\} /\backslash \{1\}$	'$(\{1, 2, 3\} - \{1, 2\}) /\backslash \{1\}$' or '$\{1, 2, 3\} - (\{1, 2\} /\backslash \{1\})$'

□

The rules of how to interpret an expression are as follows:

1. If an expression has brackets, the (sub-)expression inside the brackets is evaluated first.
2. If an expression has more than one operator, the operator with the highest priority (lowest level in the table below) is applied first.
3. If an expression has more than one operator at the same priority (the same level), the binding order at the level decides which operator is applied. If the binding order is *left*, the left-most operator is applied first. If the binding order is *right*, the right-most operator is applied first. If the binding order is *none*, Chi does not allow operators at the level to be combined without using brackets to clarify the order.

Repeat applying these rules in the stated order, until the expression is reduced to its value.

The priority level of the operators and the binding order at each level is listed in the following table.

Level	Binding order	Operators	Description
1	none	$e_0 < e_1, e_2, \ldots, e_n >$	Template instantiation[pages 18, 20]
2	left	$e_0(e_1, e_2, \ldots, e_n)$	Function call[page 45]
3	left	$e_1.e_2$	Projection
4	left	e'	Derivative
5	right	`sample` e, `pick` e, $+ e$, $- e$	Unary operators
6	right	e_1 `^` e_2	Power operator
7	left	e_1 `*` e_2, $e_1 /\backslash e_2$, e_1 / e_2, e_1 `div` e_2, e_1 `mod` e_2	Multiplication operators
8	left	$e_1 + e_2$, $e_1 \backslash/ e_2$, e_1 - e_2, e_1 ++ e_2, e_1 -- e_2	Addition operators
9	left	e_1 `min` e_2, e_1 `max` e_2	Min and max operators
10	right	`not` e	Boolean negation operator
11	left	$e_1 < e_2$, $e_1 <= e_2$, $e_1 = e_2$, $e_1 /= e_2$, $e_1 >= e_2$, $e_1 > e_2$, e_1 `in` e_2, e_1 `sub` e_2	Relational operators
12	none	$e_1 => e_2$	Boolean implication operator
13	left	e_1 `and` e_2	Boolean conjunction operator
14	left	e_1 `or` e_2	Boolean disjunction operator

VariableExpression

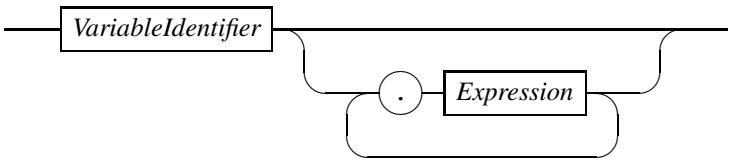

Figure 4.21: Raildiagram of *VariableExpression*.

Examples

To get an understanding og how the priority operators work, here are a few examples that may show surprising behavior, but that can fully be explained by applying the priority rules.

Expression	Meaning	Explanation
`3 + 2 min 1`	(3 + 1) min 1	The addition operator is applied first
`not 5 < 2`	Error	Interpreted as (¬5) < 2, which has no meaning (the 'not' operator at level 10 is applied first)
`5 = 3 = false`	(5 = 3) = false	Compare the comparison result of (5 = 3) with 'false' (both operators at level 11, and left binding order)

□

4.18 Addressable expressions

In the fold statement[page 60], fold expressions, the assignment statement[page 52], and the receive statement[page 54] computed or received values should be stored. Expressions that can be used as storage location are called *addressable expressions*. Depending on the statement, different form of addressable expressions may be used.

The simplest addressable expression is a *VariableExpression* block, defined in Figure 4.21. A *VariableExpression* is a *VariableIdentifier*[page 68] (an identifier that refers to a variable or formal parameter), optionally followed by zero or more projection operations (if the variable or formal parameter is vector[page 38] or record tuple).

At assignment statements and receive statements, you can assign multiple variables at a time. Such an expression is also an addressable expression, and it is defined by the *AddressableExpression* block in Figure 4.22. As you can see, an *AddressableExpression* is either a *VariableExpression*[page 48] (at Track 1), or it is a tuple or a vector (at Tracks 2 and 3) of addressable expressions. In some situations, you may omit the tuple brackets.

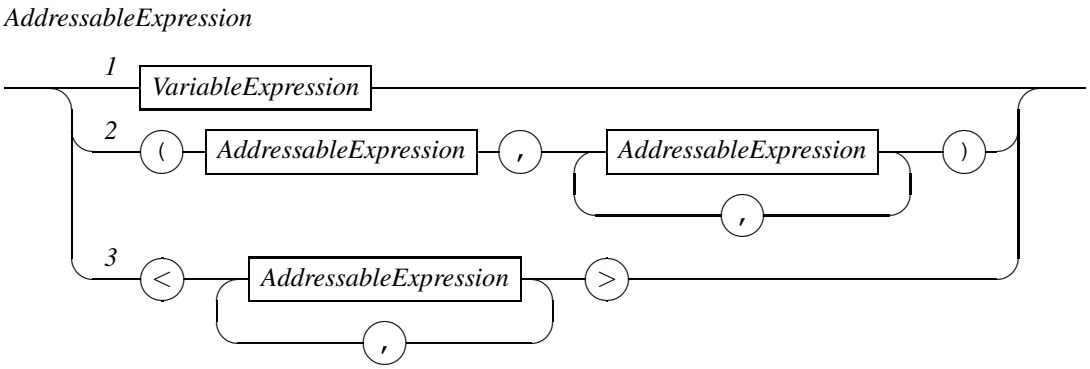

Figure 4.22: Raildiagram of *AddressableExpression*.

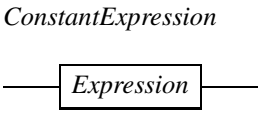

Figure 4.23: Raildiagram of *ConstantExpression*.

Examples

Example	Description
`v.2`	Third field in vector or tuple variable `v`
`x, y := y, x`	Tuple `x, y` is adressable
`((a, b), c)`	A nested addressable expression with variables `a`, `b`, and `c`

□

4.19 Constant expressions

At a number of places, a constant expression is needed. As you can see in the *ConstantExpression* diagram in Figure 4.23, a constant expression is an *Expression*[page 17]. The difference with the latter is that a constant expression may not change. To ensure that, only literal values such as 132 or (2.71828, 3.14159), and references to other constants (through the *ConstantIdentifier*[page 75] block) may be used in constant expressions.

WARNING: *The tools only support literal values.*

Chapter 5
Statements

Since Chi is a language based on process algebra, statements are considered to be behavioral expressions, much like 'normal' expressions are expressions over values. You can see this idea in the grammar of a statement shown in the *Statement* diagram in Figure 5.1. Just like normal expressions, you can group statements together by surrounding them with a pair of round brackets, as shown at Track 1. The elementary statement is represented by the *BasicStatement*[page 51] block at Track 2, which is explained in more detail in the next section. Tracks 3 and 4 show the syntax of unary and binary statement operators. These are explained in sections 5.10 and 5.11.

5.1 Basic statements

Rather than having one long list of basic statements, they have been split into different kinds of basic statement according to their use. The *BasicStatement* diagram in Figure 5.2 shows the kinds of basic statement available. The *AssignmentStatement*[page 52] block at Track 1 defines statements for assigning values to variables, it is further explained in Section 5.2. State-

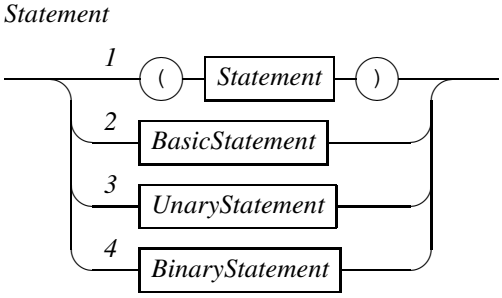

Figure 5.1: Raildiagram of *Statement*.

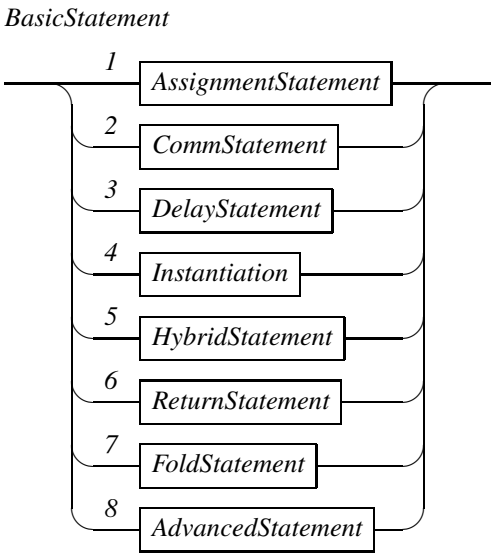

Figure 5.2: Raildiagram of *BasicStatement*.

ments for communicating between processes are defined in the *CommStatement*[page 55] block at Track 2, explained in Section 5.3. Delaying for some time can be specified by the delay statements in the *DelayStatement*[page 55] block at Track 3 explained in Section 5.4. New processes can be started by using process instantiation, defined in the *Instantiation*[page 57] block at Track 4. The statements are further explained in Section 5.5. Statements needed for hybrid models are made available at Track 5 by the *HybridStatement*[page 58] block and explained in Section 5.6 of this chapter. The *ReturnStatement*[page 60] block at Track 6 defines the `ret` statement, used exclusively in functions. Details can be found in Section 5.7. Finally, statements for advanced users are shown at Tracks 7 and 8. The *FoldStatement*[page 60] block explained in Section 5.8 shows how to fold several statements into one and the *AdvancedStatement*[page 61] block introduces a number of seldomly used statements, details can be found in Section 5.9.

5.2 Assignment statements

Assignment statements assign values to variables. There are three such statements. Their syntax is shown in the *AssignmentStatement* diagram in Figure 5.3. At Track 1, the most often used form of assignment is shown, namely the normal assignment statement. Normally, the statement is used without guard, and without channel expression (that is, both the *OptGuard* and the *OptChannel* blocks are empty). In that case, the statement consists of a comma-seperated list of variables (addressable expressions[page 48]), the `:=` symbol, and finally a comma-seperated list of expression values. The semantics of the statement is that that the values at the right are copied (assigned) to the variables at the left. Note that first **all** values at the right are computed before assigning them to the variables at the left.

In the official language definition ([3]), the number of values at the right must be the same as the number of variables at the left. The implementation is more flexible, you can also expand one record tuple[page 39] value from the right to a (nested) list of variables at the left, or vice versa, merge several (nested) values from the right into one record tuple[page 39] variable. The *OptGuard* block can be used to conditionally assign values to variables, the statement can only

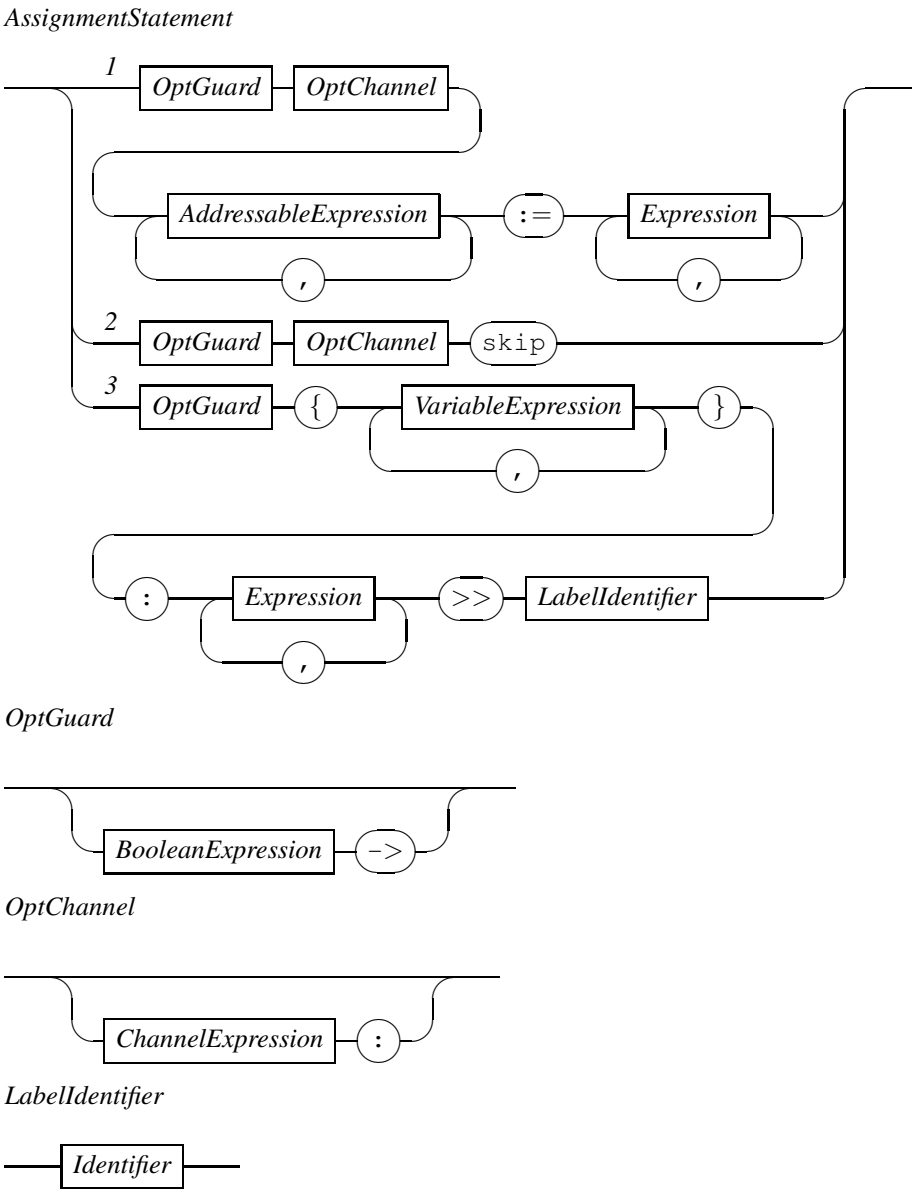

Figure 5.3: Raildiagram of *AssignmentStatement*.

be executed when the guard expression (the *Expression*[page 17] block before the `->` symbol) is true. If the guard is false, the statement can only wait. The *OptChannel* block is seldomly used. Its main use is in results of process-algebraic reasoning about Chi programs where a matching pair of communication statements is replaced by an assignment statement (that is, '`h!!8 || h?x`' is rewritten to '`h: x := 8`').

At the second track, you can see the 'skip' statement. The reason it is part of the *AssignmentStatement*[page 52] diagram is that it is a special case of assignment, namely the case that the list variables and the list values are both empty. The *OptGuard* and *OptChannel* blocks have the same function as with the normal assignment statement, except that 'skip' is generated from a synchronization instead of a communication.

Finally, at Track 3, the action predicate statement is shown. Although it is not used often, it forms the foundation of assignments. The *OptGuard* block has the same function as at previous tracks, it allows the assignment to be performed conditionally. The assignment it self is expressed in two lists of expressions. The first list between curly brackets is a list addressable expressions[page 48]. It states which variables may change. The second list of expressions state the conditions that must hold after the assignment has taken place. The final *LabelIdentifier* block defines the label that is attached to the action predicate statement. This label is emitted when the assignment takes place.

Examples

Example	Description
`x := 1`	Assignment of the value 1 to variable x
`x, y := y, x`	Swapping variables x and y
`t := x, y`	Merging values to a record tuple t
`t.0, t.1 := x, y`	Equivalent assignment
`x, y := myfunc(a, b)`	Splitting a record tuple returned by a function call to seperate variables
`x = 5 -> y := 3`	Condition assignment, if `x = 5` assign 3 to variable y
`{x, y}: x=old(y), y=old(x) » tau`	Action predicate statement equivalent to the swapping example

The variable swapping example works due to the fact that first the values of both y and x are saved from the right-hand side, before assigning them to the variables at the left-hand side. In the action predicate example, the `old` annotation explicitly refers to the value prior to the assignment.

□

5.3 Communication statements

With communication statements, you can exchange information between two different processes, or between a process and the environment. Communication between two processes occurs at a common channel, communication with the environment happens without channel. All communication in Chi is synchronous, that is, sending of the information and receiving of the information occurs in the same (atomic) step. In other words, either the exchange of information has occurred or it has not occurred.

In addition, a notion of direction exists with the statements. Sending information away is done with a send statement, and receiving information is done with a receive statement. The direction is indicated in the statement with an exclamation mark (‘`!`’) for sending of information, and with a question mark (‘`?`’) for receiving of information.

Since communication is synchronous, it may happen that a send statement could be executed in a process, but there is no matching receive statement (that is, with a common communication channel) or vice versa. In those cases, the statement waits until a matching partner statement can also be executed. In some cases, waiting for a partner process is unwanted behavior, if the communication statement can be executed, there should be a partner process at the same time, delaying would be bad behavior. In other words, you want express *communicate now*. In the syntax this behavior is indicated with double exclamation marks for

the sending statements and with double question marks for the receiving statements. Before discussing the syntax of the communication statements, it should be noted that the choice between communication and delaying is not hard-coded into the language, it is enforced by the urgent communication operator[page 61] that is normally placed at one of the outermost levels in the model. You can override this behavior by replacing the outermost levels in an uncooked model[page 83]. Note however that not all tools support these models.

In the *CommStatement* diagram in Figure 5.4, the syntax of the communication statements is shown. Tracks 1 through 3 show the send statements and tracks 4 through 6 show the receive statements. All statements have an *OptGuard* block, indicating that the statement can be conditionally executed. If the block is not empty, the *Expression*[page 17] block in the *OptGuard* must evaluate to true to allow the communication statement to be executed.

Tracks 1, 2, 4, and 5 have an expression in front of the direction. This expression defines which communication channel is used. If the channel expression is missing (Tracks 3 and 6), the statement communicates with the environment. The expressions behind the direction indicate the information to exchange, expressions behind a send statement refer to values, expressions behind a receive statement refer to variables (addressable expressions[page 48]). If there are no expressions (Tracks 2 and 5) only synchronization takes place without exchange of data.

5.3.1 Channel expressions

The channel expressions are defined in the *ChannelExpression* diagram in Figure 5.5. A channel expression consists of an channel identifier by means of the *ChannelIdentifier*[page 70] block, optionally followed by a number of projection operations, that selects channels from a bundle.

5.4 Delay statements

The *delay statement* at the first track in the *DelayStatement* diagram in Figure 5.6 is used most often. The statement delays for the amount of time units indicated by the *Expression*[page 17] value (a non-negative value of type `nat`, `int`, or `real`), and then terminates.

Examples

```
model M() =
|[ cont x: real = 2.0
:: x' = 1
|| delay x + 1
]|
```

At the start of the execution of the delay statement, the value of the expression `x + 1` is 3.0. The delay statement will thus wait for 3.0 time units, despite the fact that the value of the expression changes as time progresses. After 3.0 time units the statement terminates. Since it is the last statement of the model, the execution of the model also terminates.

□

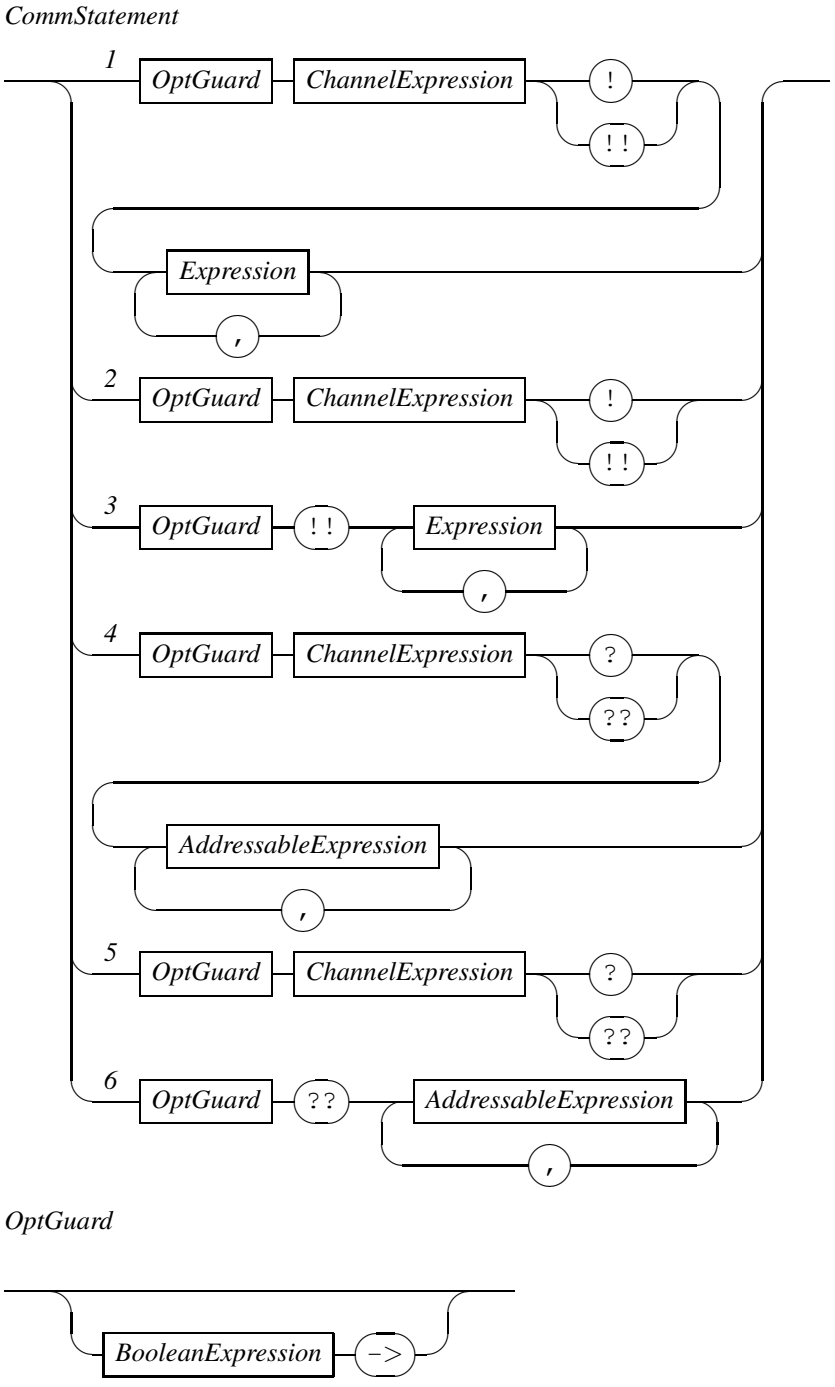

Figure 5.4: Raildiagram of *CommStatement*.

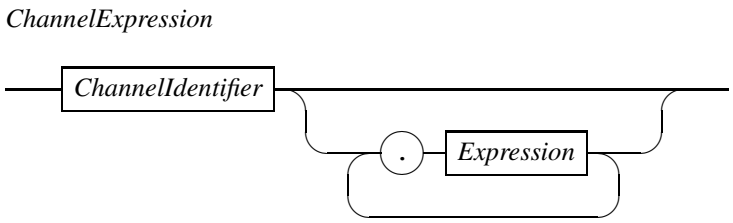

Figure 5.5: Raildiagram of *ChannelExpression*.

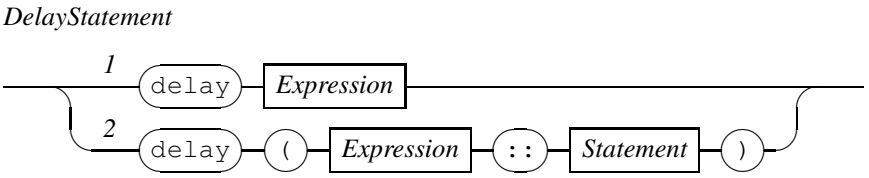

Figure 5.6: Raildiagram of *DelayStatement*.

At Track 2 the more generic form of a delay statement is shown. This form is called the *delay operator*. After delaying for a number of time units (expressed by the value of *Expression*[page 17]), the statement expressed by *Statement*[page 51] is executed.

5.5 Instantiation statements

In Chi you can instantiate process definitions[page 81], process declarations[page 81], and mode definitions[page 71]. Their syntax is shown in the *Instantiation* diagram in Figure 5.7 at Tracks 1 and 2. At Track 1, a process definition or declaration is instantiated. The *ProcessIdentifier* must match the name of the process definition or declaration. If it is a templated process, the *TemplateValue*[page 20] blocks are used to define the actual values of the template parameters. Giving values for template parameters is called 'template instantiation'[pages 18, 20], and is discussed in more detail in Section 4.2. The *ProcessIdentifier* block and the optional *TemplateValue*[page 20] blocks together refer to a concrete process definition or declaration that can be instantiated. The list of expressions between round brackets are the actual arguments of the instantiated definition. The number of arguments must match the number of formal parameters. Also their class and type must match. The actual expressions are assigned to the formal parameters in a component-wise fashion (the first expression is assigned to the first formal parameter, the second expression to the second formal parameter, etc). How each expression is assigned depends on the class[page 67] of the formal parameter. Reference parameters[page 67] (parameters of classes `cont`, `alg`, `chan`, or `var`) refer to the variable stated in the actual expression, value parameters[page 67] (parameters of class `val`) create a local variable, and use the value of the actual expression as initial value.

At Track 2, a mode definition is instantiated.[page 71] The *ModeIdentifier*[page 71] block must match with the identifier of the mode definition being instantiated. Since mode definitions have no parameters, there are no round brackets and no list of expression list is present.

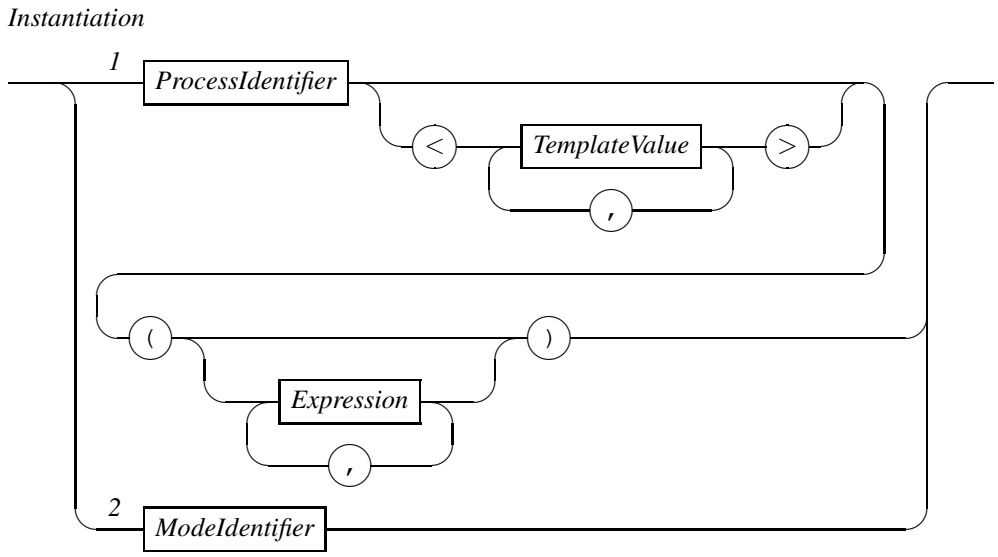

Figure 5.7: Raildiagram of *Instantiation*.

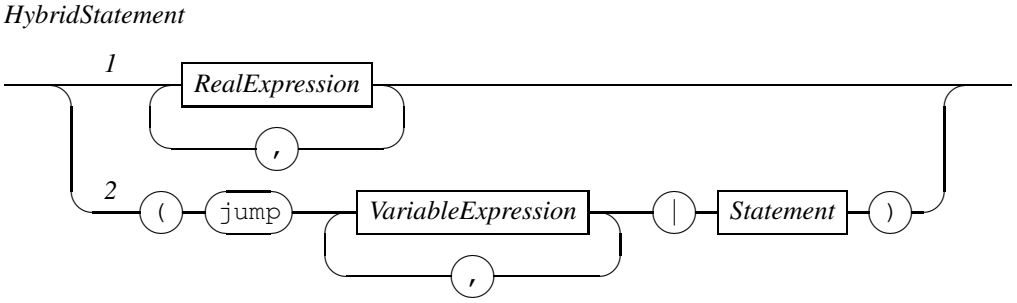

Figure 5.8: Raildiagram of *HybridStatement*.

5.6 Hybrid statements

Hybrid statements are (as the name already suggests) statements used exclusively in hybrid or continuous models. The syntax of the hybrid statements is shown in the *HybridStatement* diagram in Figure 5.8. The hybrid statement at Track 1 is the *delay predicate*, more commonly known as *equation* or *invariant*. It is a comma-seperated list of boolean expressions that will always hold as long as the statement is active.

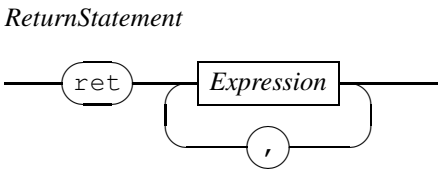

Figure 5.9: Raildiagram of *ReturnStatement*.

Examples

```
model M() =
|[ cont x: real = 2.0
:: x' = 1
|| delay 3.0
]|
```

After initializing the continuous variable x to the value 2.0, this model delays for 3.0 time units. During the delay, the delay predicate $x' = 1$ holds, which means that during those 3.0 time units, the value of x increases linearly to 5.0.

□

The *jump enabling* statement shown at Track 2 adds the list of variables listed (using *VariableExpression*[page 48] blocks) before the vertical bar to the set of jumping variables while the statement behind the bar is executed.

Examples

```
model M() =
|[ cont x, y: real = (2.0, 2.0)
 , chan c: real
:: x = y
|| c ! 1.0
|| (jump x, y | c ? x )
]|
```

In this program two continuous variables x and y exist. Via the equation x = y, both variables are always equal to each other. If a new value for the continuous variables is received from a channel, the values of x and y must both be able to jump to the received value, otherwise the new value will not be accepted.

□

Figure 5.10: Raildiagram of *FoldStatement*.

5.7 Return statement

The return statement shown in the *ReturnStatement* diagram in Figure 5.9 is used in the statement part of function definitions[page 78]. Upon execution, the computation performed by the function ends, and the result of the computation returned to the caller is the value of the expression behind the `ret` keyword.

5.8 Fold statement

Frequently, a statement is needed several times, often with small changes between different instantiations of the statement. For the case that these changes can be expressed in a (changing with each instantiation) constant value, the fold statement may be used to instantiate the statement (that is, unfold the fold statement), reducing the amount of code that needs to be written and maintained.

The syntax of the fold statement is shown in the *FoldStatement* diagram in Figure 5.10. It consists of three arguments between a pair of round brackets. The first argument is the binary statement operator[page 64] that is to be used *between* different instantiations of the statement, the second argument is the *StatementIterator*[page 60] block which defines the iterator to use (the range of values used in the unfolding process as well as the name of the iterator value). The third argument is a *Statement*[page 51] block which states the statement to instantiate each time. In the statement, the iterator value may be used (as read-only constant). This value gets a different value each time the statement is unfolded

Semantically, the entire unfolding process is performed *before* execution, that is, during compilation of the Chi specification. The statement is copied once for each instantiation, where the name of the iterator is replaced by its value. Different instantiations are seperated using the connector of the first argument.

Examples

The following example sends a sequence of values to another process. The folded statement is written as `( ; , i <- 0..2 , c!i )`. This is equivalent to `c!0 ; c!1 ; c!2`.

□

The *StatementIterator* diagram in Figure 5.11 defines the range used for unfolding the fold

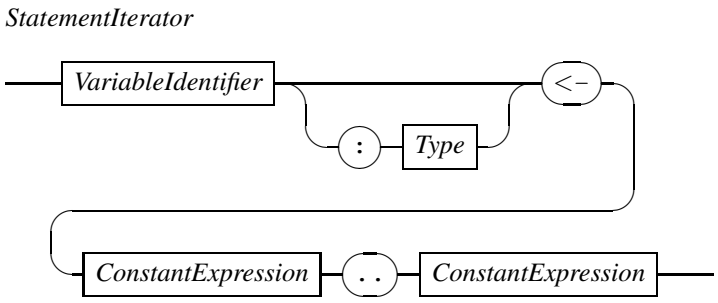

Figure 5.11: Raildiagram of *StatementIterator*.

statement[page 60]. The first expression defines the lower bound, the second expression defines the upper bound. Both expressions must be constant values either both of type `nat` or both of type `int`. Also, the second value must be at least as large as the first value (the range may not be empty). The unfolding process iterates over all values between and including both bounds.

WARNING: *The current implementations limits the type of the bounds to* `nat` *type only.*

5.9 Advanced statements

Most of the advanced statements are derived directly from statements in the formal semantics ([3]), and have little practical value for modelers. The statements are shown in Figure 5.12. At Track 1 of the *AdvancedStatement* diagram, you can see the urgent communication operator $v_H(p)$ to give communication priority above delays. At Track 2 the syntax of the encapsulation operator $\partial_A(p)$ is shown. It is used to prevent actions from the *Action*[page 61] block from happening. The syntax of the blocked actions is shown below. At Track 3, you can see the local scope operator which is used to introduce local variables[page 69], channels[page 70], and mode definitions[page 71]. At Track 4, the syntax of the any-delay operator is shown. It overrides delay behavior of the *Statement*[page 51] inside. The signal-emission operator $u \curvearrowright p$ shown at Track 5 is used to intialize variables to satisfy the condition expressed in the *BooleanExpression*[page 23] block prior to executing the *Statement*[page 51]. Track 6 shows the `deadlock` statement δ, a statement that is consistent but can neither perform an action nor delay. Finally, the `inconsistent` statement $\perp$ at Track 7 is much like the deadlock statement, except that it is also not consistent.

In the encapsulation operator[page 61], actions can be suppressed. In Figure 5.13, the syntax that specifies which actions should be suppressed is shown in the *Action* diagram. An action starts with the name of the action, followed by one or more action patterns (one for each argument of the action). Actions that match with the name and all the argument patterns are suppressed (If the pattern in *ActionPattern* is an expression, the actual value of the action must be equal to the value of the expression. If the pattern is a asterisk, all values match.

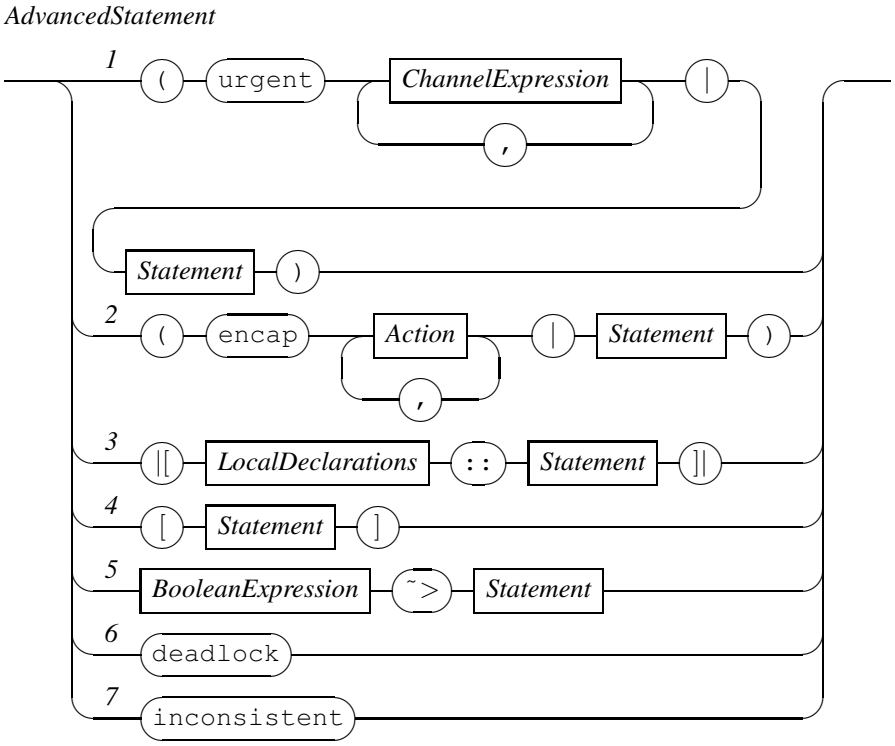

Figure 5.12: Raildiagram of *AdvancedStatement*.

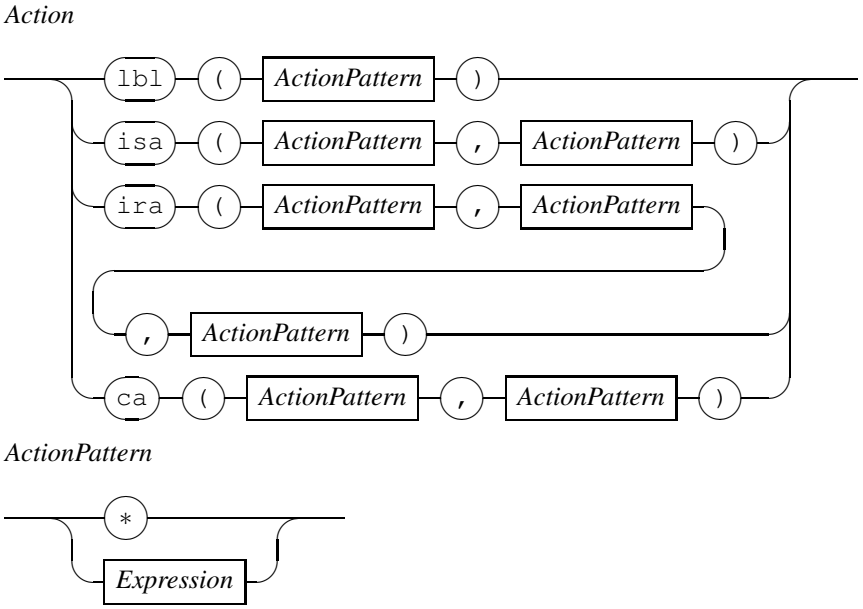

Figure 5.13: Raildiagram of *Action*.

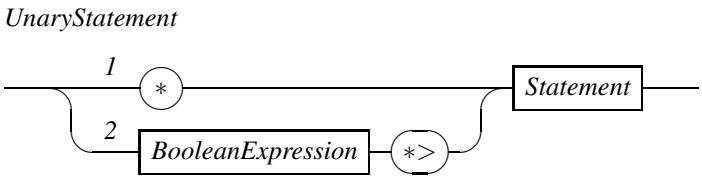

Figure 5.14: Raildiagram of *UnaryStatement*.

5.10 Unary statements

There are two unary statement operators, shown in the *UnaryStatement* diagram in Figure 5.14. (In Section 5.9 a few additional unary operators are defined, these are however of little practical value.)

The loop statement is shown at Track 1. This prefix causes the statement after it to be repeated forever. For example

```
* c?x
```

will forever receive values from channel `c` into variable `x`.

The while statement shown at Track 2 behaves very much like the loop statement previously discussed, except that the loop ends when the *BooleanExpression*[page 23] of the construct evaluates to false.

Examples

```
x := 1 ; x<100 *> x := x*2
```

After initializing `x` to 1, the assignment `x := x*2` is repeatedly executed, until the condition `x` < 100 becomes false. At that moment, the value of `x` is 128.

□

The while statement also checks the value of the expression before entering the loop for the first time.

Examples

```
false *> x := 1
```

Since the expression `false` does not evaluate to true, the loop is never entered, and the `x := 1` statement is never executed.

□

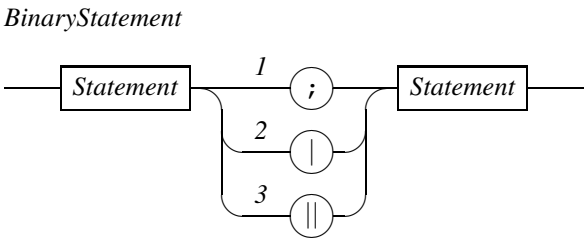

Figure 5.15: Raildiagram of *BinaryStatement*.

5.11 Binary statements

There are three binary operators on statements, as shown in Diagram *BinaryStatement* at Figure 5.15. At Track 1, the sequential composition operator is shown, Track 2 displays the syntax of the alternative composition operator, and at Track 3 defines the syntax of the parallel composition operator. All three operators are explained below.

Statements seperated by a sequential composition operator are executed sequentially, that is, one after the other.

Examples

```
x := 1 ; y := 2
```

In this example, the assignment statement `x := 1` and the assignment statement `y := 2` are connected with sequential compostion, meaning that the latter assignment will take place after the former has ended.

□

The second way of connecting statements is by *alternative composition*. When two statements are connected in this way, both statements wait together. Execution (with an action) is done by one of the statements only, the other statement is silently discarded at that moment.

Examples

```
x := 1 | y := 2
```

Here the assignment statement to `x` and the assignment statement to `y` are connected with alternative composition. The former statement can perform an action (and assign the value 1 to variable `x`). The latter statement can also perform an action (and assign the value 2 to variable `y`). The alternative composition operator makes a *non-deterministic choice*, and excutes either the former or the latter assignment, but not both.

□

To show the difference between waiting and performing an action, consider the following

example.

Examples

```
delay 5 | delay 3
```

Both statements can only delay, thus the alternative composition of both statements can also (only) delay (only if both statements of an alternative composition can delay, the composition can delay). After three time units, the `delay 5` statement[page 55] can still delay for two time units, but the `delay 3` cannot. It can only perform an action (and terminate). As a result, the combined statements cannot delay, the only option is to perform the action of the second delay statement, silently discarding the first delay statement.

□

The parallel composition operator is the third way of connecting statements. Its meaning is that both statements delay together (the same as with the alternative composition operator), but actions are executed in arbitrary order. The combined statement terminates when both sides have terminated.

Examples

```
x := 1 || y := 2
```

The first and the second assignment are executed in arbitrary order, that is, it does either `x := 1 ; y := 2` or `y := 2 ; x := 1`.

□

5.12 Statement operator priorities

For each statement operator a priority has been assigned to minimize the number of brackets needed for the average specification. The order is shown in the list below.

1. *Unary statement operators*: Operators of the loop, and the while statement (shown in the *UnaryStatement*[page 63] diagram have the highest priority. Since both operators are prefix operators, there is never confusion of the order between them.
2. *Sequential composition operator*: The sequential composition operator comes directly after the unary operators. It is shown in the *BinaryStatement*[page 64] diagram.

Examples

```
|[ var n: nat = 0 :: n < 5 *> c ! n ; n := n + 1 ]|
```

means

```
|[ var n: nat = 0 :: ( n < 5 *> c ! n ) ; n := n + 1 ]|
```

since the unary while statement has a higher priority than the sequential composition operator.

If you want to include the increment of `n` in the loop, you must write brackets around the statements in the loop explicitly, as in

```
|[ var n: nat = 0 :: n < 5 *> ( c ! n ; n := n + 1 ) ]|
```

□

3. *Alternative and parallel composition operators*: The remaining two binary operators are alternative composition operator and the parallel composition operator. These operators bind the least strongly.

   There are no priority rules between the alternative composition operator and the parallel composition operator.

Examples

This means that you *must* use brackets to define the binding, for example

```
x := 1 | x := 2 || x := 3 ; x := 4
```

is **incorrect**, and must be rewritten to either

```
( x := 1 | x := 2 ) || x := 3 ; x := 4
```

or

```
x := 1 | ( x := 2 || x := 3 ; x := 4)
```

In all cases above, the `x := 3` and `x := 4` statements are bound together, since the sequential composition operator binds stronger.

□

Chapter 6
Declarations

Variable declarations occur at two places, at formal parameter declarations of functions, processes and models, and at local variable declarations.

Each variable declared in a specification has a variable class associated with it. There are four different variable classes:

Keyword	Variable class	Description
`var`	*Discrete variable*	Its derivative is always 0, and its value does not jump while delaying. In local variable declarations, it is used to declare discrete variables. In formal parameter declarations, it is used to link the declared variables with discrete variables from its caller.
`val`	*Discrete value*	Behaves as a discrete variable but cannot be used in local variable declarations. When used in formal parameter declarations, variables are not linked with each other.
`cont`	*Continuous variable*	Variables of this class have a derivative, and their value does not jump while delaying.
`alg`	*Algebraic variable*	It has no derivative, its value is entirely controlled by the set of active equations. Can be used to declare algebraic variables in local variable declarations, and to link algebraic variables to algebraic or continuous variables in its caller.
`chan`	*Channel variable*	Used in local variable declarations to declare new channels, and in formal parameter declarations to link channels with each other.

The meaning of a variable being linked to another variable becomes clear when new values are assigned. A change in value of one of the linked variables (by assignment for example) causes the same change to be propagated to the other variable(s) as well. In other words, variables and channels linked together always have the same value. Parameters with the linking property are also known as *reference parameter*. Parameters without linking property are called *value parameter*.

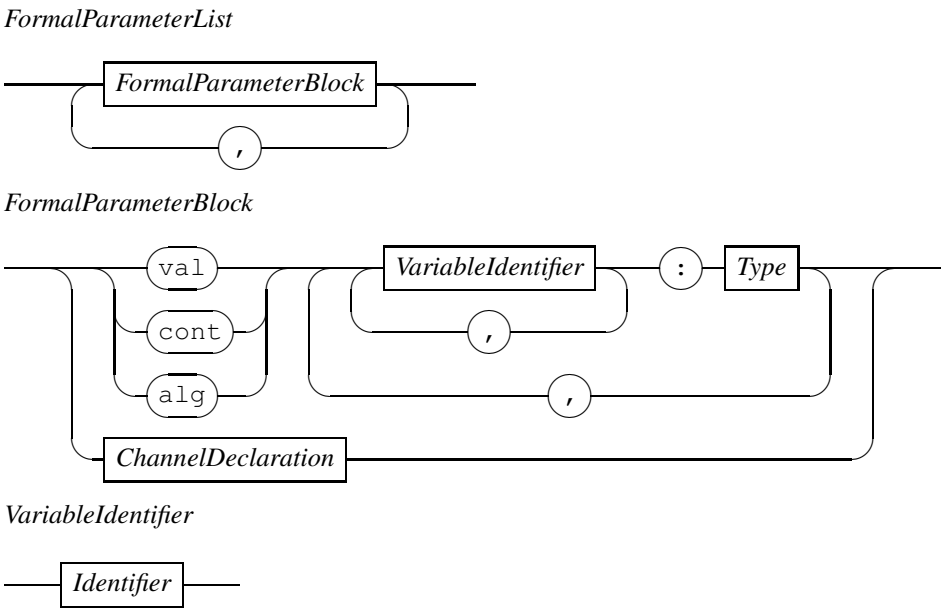

Figure 6.1: Raildiagram of *FormalParameterList*.

6.1 Formal parameter declarations

The syntax of formal parameter declarations is defined in the *FormalParameterList* diagram in Figure 6.1. It consists of one or more *FormalParameterBlock* blocks. Each block is either a channel declaration (explained in Section ??), or it is a list of variables being declared to belong to a variable class and having type *Type*[page 11]. The *VariableIdentifier* block is an *Identifier*[page 8] except that it refers to a variable.

Examples

As an example of formal parameter declarations, consider the process definition below

```
proc P(chan c?: real, cont w: real, val inc: nat) =
|[ inc := inc + 1
 ; c?w
 ; w := w + inc
]|
```

The formal parameter declararations of process `P` introduce a channel `c`, a continuous variable `w`, and a discrete variable `inc`.

□

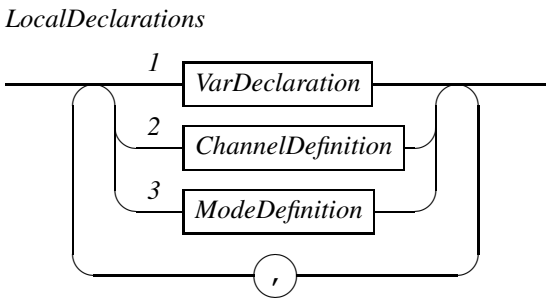

Figure 6.2: Raildiagram of *LocalDeclarations*.

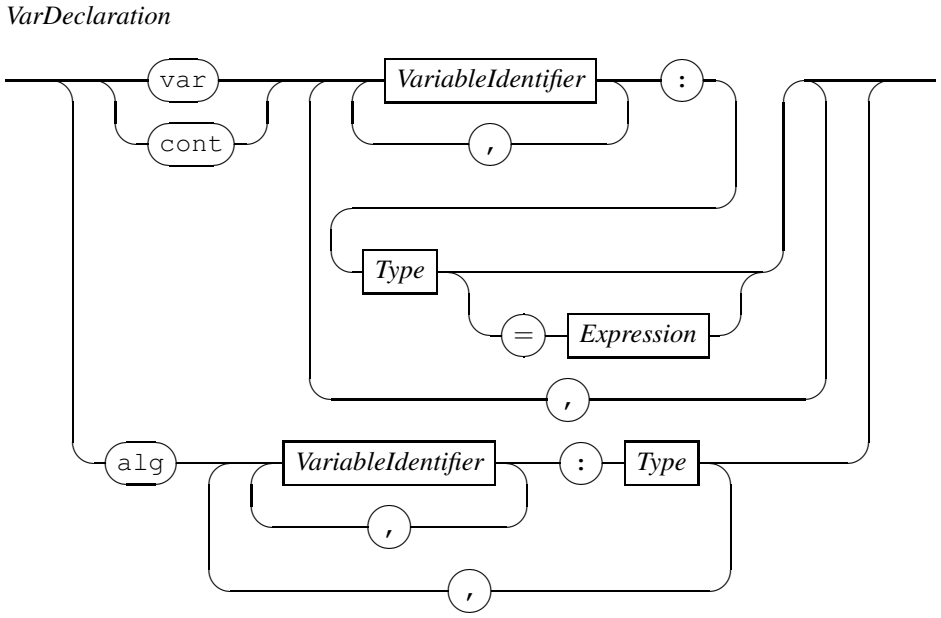

Figure 6.3: Raildiagram of *VarDeclaration*.

6.2 Local variable declarations

Local variable declarations introduce new variables inside a function definition,[page 78] process definition,[page 81] or model definition.[page 83] The syntax of local variable declarations is shown in the *LocalDeclarations* diagram in Figure 6.2. Local declarations is a comma-seperated list of *VarDeclaration*[page 69], *ChannelDefinition*[page 70], and *ModeDefinition*[page 71] blocks. Variable declarations are explained below, channel definitions are explained in Section 6.3, and mode definitions are explained in Section 6.4.

Local variables are defined by the *VarDeclaration* diagram in Figure 6.3. Discrete and continuous variables are declared with the '`var`' respectively '`cont`' keywords, followed by a list of identifiers (the *VariableIdentifier* blocks), and a type. Optionally, the variables can be given an initial value by means of an expression. Algebraic variables are declared in much the same way, except that they cannot be initialized with an expression.

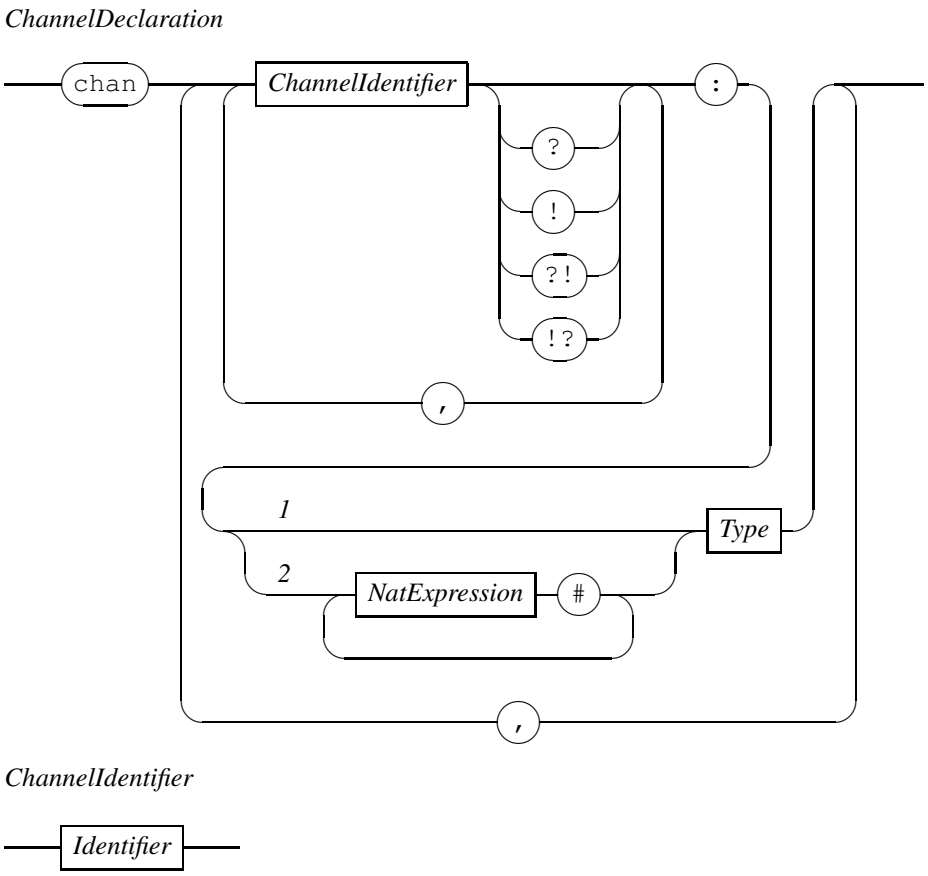

Figure 6.4: Raildiagram of *ChannelDeclaration*.

6.3 Channels

With channel declarations, you declare channels which can be used to synchronously exchange data between different processes. The syntax of a channel declaration is stated in the *ChannelDeclaration* block in Figure 6.4. Channel declarations are used in formal parameter declarations.[page 68] They start with the keyword `chan`, followed by a comma seperated sequence of declarations. Each declaration consists of a sequence of *ChannelIdentifier* blocks, a colon, and the type of the channel. Each channel identifier (which is just an *Identifier*[page 8]) may be followed by a specification of the allowed directions of data transfer on the channel. A question mark means that data is received from the channel. An exclamation mark means that data is sent. Specifying both a question mark and an exclamation mark (in either order) means that the channel can both be read from and be written to. The latter is also the default, which means that both read and write is allowed on a channel for which no direction is specified.

The type of the channel consists of two parts. The first part defines the structure of the channel(s). Either it is a single channel (use Track 1), or it is a bundle (use Track 2). The second part defines the type of the data being transferred of the channel(s).

Channel definitions shown in the *ChannelDefinition* diagram in Figure 6.5 are used in local variable declarations. They are like channel declarations, except that receive-only or write-only

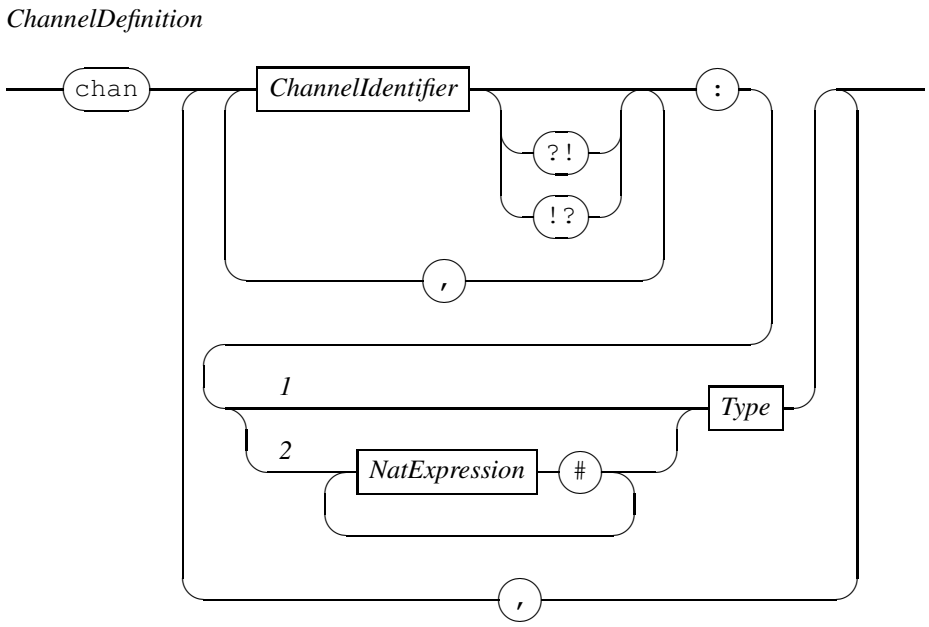

Figure 6.5: Raildiagram of *ChannelDefinition*.

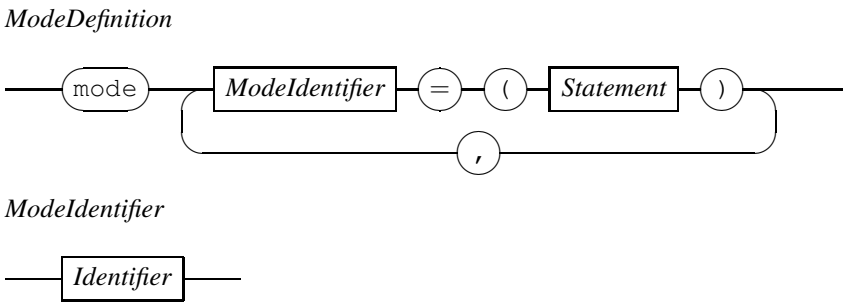

Figure 6.6: Raildiagram of *ModeDefinition*.

channels cannot be created.

6.4 Mode definitions

A mode definition gives a name to a piece of code. They are often used to program state machines in Chi. The syntax is shown in Figure 6.6. The *ModeDefinition* starts with the `mode` keyword, followed by a comma seperated list of named statement fragments. The name of each fragment is defined by the *ModeIdentifier* block. The identifier is called *mode variable*. These mode variables may be used at any place where a statement is expected, within the scope of the mode definitions.

Examples

```
|[ var n: nat
 , mode reset  = ( n := 0; adding )
      , adding = ( n := n + 1; ( n=5 -> reset | n < 5 -> adding ) )
:: reset
]|
```

This scope declares a discrete variable `n`, and two mode variables `reset` and `adding`. Execution of the scope implies execution of `reset`.

□

The meaning of execution of a mode variable is to execute the statements associated with the variable instead. As you can see in the example, it is allowed to use (names of) other mode variables including itself.

Binding of names in mode definitions is done at definition time rather than at the time of use. That means that in the following example

```
|[ var n: nat = 0
 , mode inc  = ( n := n + 1 )
:: |[ var n: nat = 50
   :: inc
    ; !! n
   ]|
 ; !! n
]|
```

the variable `n` declared at the first line is incremented, rather than variable `n` that exists at the moment that `inc` is executed (which is declared at the third line). Execution of the example will thus result in outputting values 50 and 1.

Chapter 7
Definitions

A Chi program is defined as a (non-empty) sequence of definitions and declarations as shown in the *ChiProgram* diagram in Figure 7.1.

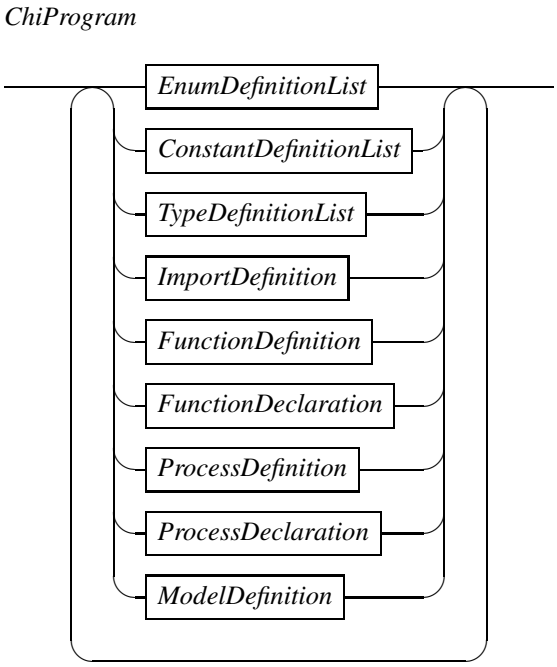

Figure 7.1: Raildiagram of *ChiProgram*.

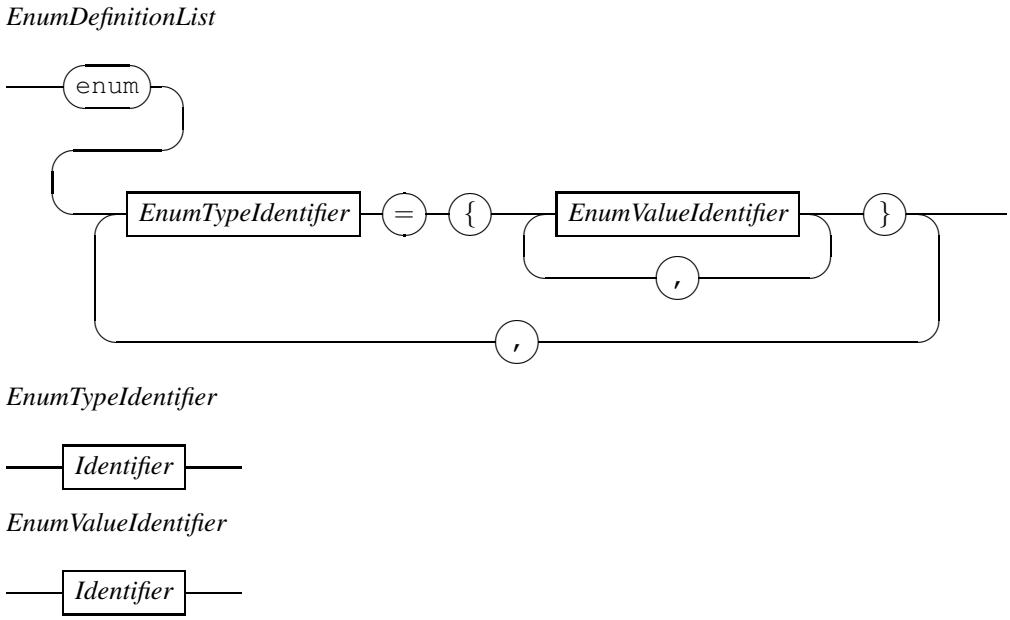

Figure 7.2: Raildiagram of *EnumDefinitionList*.

7.1 Enumeration definitions

Enumeration definitions introduce a new type called *enumeration type*, and a (normally small) set of values for that type. It is used for creating readable constant values. The syntax of an enumeration definition is shown in Figure 7.2. An enumeration definition starts with the keyword `enum`, followed by the name of the enumeration type that is defined by means of a *EnumTypeIdentifier* block. Between the curly brackets, the set of allowed values of the enumeration type is listed as a sequence of *EnumValueIdentifier* blocks.

Examples

The following enumeration definition creates a new enumeration type `flagcolors`, and three (color) value constants `red`, `white`, and `blue`.

```
enum flagcolors = { red, white, blue }
```

□

With this definition, you can create variables of type `flagcolors`, and assign colors to them, as in `x := red`.

Note: Unlike enumeration definitions in many other languages, in Chi there is no order between the different values, you cannot request the previous or next value. Testing for equality (and in-equality) is however allowed.

WARNING: *Enumeration definitions are not implemented yet.*

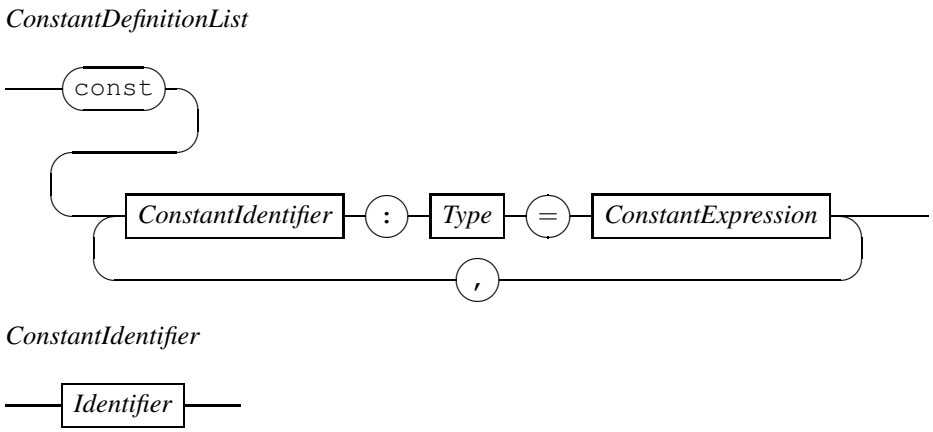

Figure 7.3: Raildiagram of *ConstantDefinitionList*.

7.2 Constant definitions

Constant definitions are used to give names to constant values. The syntax is shown in Figure 7.3. A list of constant definitions starts with a `const` keyword. Each constant consists of its name (a *ConstantIdentifier*, which is a (new) *Identifier*[page 8] that refers to a constant value), a description of the type of the constant, and the value of the constant defined by means of a *ConstantExpression*[page 49] expression.

Examples

An example of a constant definition list is

```
const pi: real = 3.14159265358979323846
    , day : nat = 24 * 60 * 60
```

which states that the (upto now unused) name 'pi' is attached to a real constant value slightly larger than 3 and the name 'day' is attached to a natural number with value 86400.

□

Semantically, using the name of a constant definition is equal to inserting its value at that point in the program.

The syntax of the language allows an arbitrary expression at the right hand side of the definition. In theory, you are allowed to write any expression as long as it evaluates to a constant value (that is, it consists of expression operators, literal values and other constant identifiers). In practice, most tools cannot handle expressions such as '24∗60∗60', and restrict the syntax of a value to literal values only (that is, you *must* write '86400' instead).

WARNING: *Constant definitions are not implemented yet.*

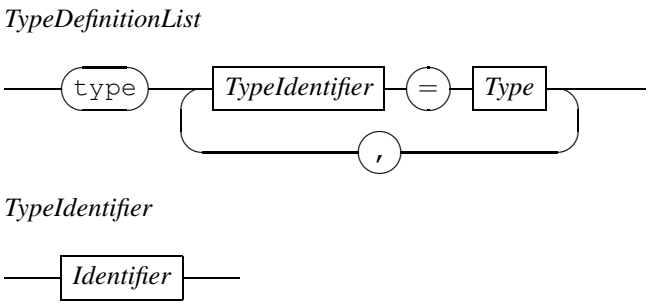

Figure 7.4: Raildiagram of *TypeDefinitionList*.

7.3 Type definitions

As with constant definitions, type definitions only attach a (new) name to an existing type. Its main use is to make specifications more readable by allowing to use a short identifier ratther than the (often complex) attached type. The syntax of a type definition is shown in Figure 7.4. The name of the new type is defined by the *TypeIdentifier* block, an *Identifier*[page 8] that refers to a type.

Examples

```
type lot = nat
   , batch = [lot]
```

The type definition attaches the type `nat` to the type identifier 'lot', and the type `[nat]` to the type identifier 'batch'.

□

Chi uses *structural type equivalence*, which means that usage of a type identifier is equivalent to stating its attached type at that point in the program.

Examples

```
var xs: batch, ys: [nat]
```

The example declares a new variables 'xs' and 'ys' both of the same type (`[nat]`).

□

7.4 Import definitions

Import definitions or import statements allow access to definitions written in other files. Its syntax is shown in Figure 7.5. The import has two forms. At Track 1, a module is imported. At

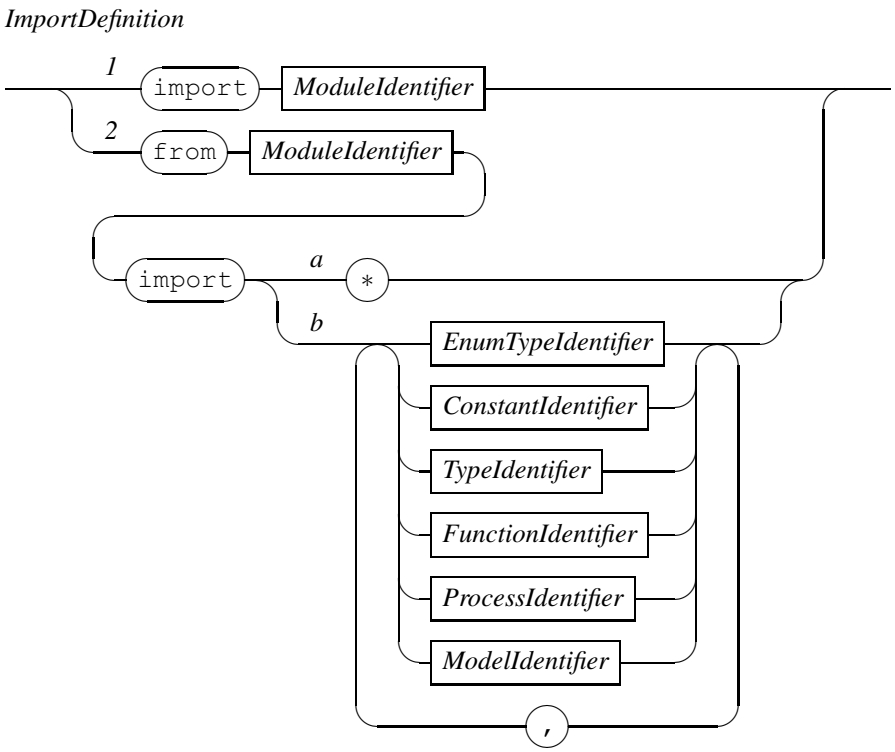

Figure 7.5: Raildiagram of *ImportDefinition*.

Track 2, definitions are imported. In the latter case the module is imported, and in addition, you can either import all definitions of the module (Track 2a), or you can explicitly list which definitions should be imported by giving a list of identifiers (at Track 2b).

Importing a module means that you can access the definitions in it by using the projection operator on the module.

Examples

```
import foo
```

imports the `foo` module using the syntax of Track 1. If this module contains for example a function definition `func inc(val n:nat) -> nat`, this function can be accessed using the projection operator, as in `y := foo.inc(x)`.

□

If you use the `inc` function a lot, prefixing each use may become a distraction rather than a help. By importing the definition as well (by using the syntax of Track 2b) you can drop the module name.

Examples

```
from foo import inc
```

This command imports the module `foo` (giving you access to its entire contents by means of the projection operator as before (that is, `y := foo.inc(x)` still works), and it imports the definitions named `inc` directly into the name space, giving you direct access as in '`y := inc(x)`'.

□

While it may seem easiest to always import all definitions from a module, it does have drawbacks due to name-space rules. For example after importing the `inc` definition from `foo`, you cannot define another `inc` because overloading of definitions is not allowed. For similar reasons, you cannot import definitions with the same name from different modules. Last but not least, a reader unfamiliar with the module structure has no way of finding out where an `inc` definition is coming from, he has to search through all imported modules to find the matching definition (with an explicit module reference such as `y := foo.inc(x)` or an explicit import such as `from foo import inc` this problem is much reduced).

WARNING: *Module imports are implemented (that is, the tools accept `import foo`), but cannot be used since the `module.name` construct does not work yet (that is, you cannot do `foo.inc`).*

7.5 Functions

A function definition[page 78] connects a name to a computation (an algorithm). Once this connection is made, the computation can be done by using its name as part of an expression evaluation (see Chapter 4 for details). The process of activating a computation by stating its name is known as *function application*,[page 45] or the more common phrase *function call*.[page 45] In Chi, the computations defined in a function are proper mathematical functions, that is, they are not influenced from the outside, and they do not have side effects.

Functions may come in existence in two ways. Either they are defined using a *FunctionDefinition* block (that is, they are stated complete with the implementation of the algorithm) or they are declared using a *FunctionDeclaration* block (that is, they are stated without implementation). The former way is used for defining functions in Chi, the latter way is used to declare that functions exist elsewhere and they may be used (called) by a Chi specification. Both function definitions and function declarations are explained below.

7.5.1 Function definition

The syntax of a function definition is shown in a *FunctionDefinition* diagram in Figure 7.6. The *FunctionIdentifier* block defines the name (which is the same as an *Identifier*[page 8]) of the computation. The optional *ExplicitTemplates*[page 84] block defines the statically decided function arguments, the declarations in the *FormalParameterList*[page 68] state the formal parameters of the function. The *Type*[page 11] block defines the type of the value that will be returned by the computation. The computation algorithm itself is defined in the *FunctionBody* block. The function body optionally starts with declarations of local variables in the

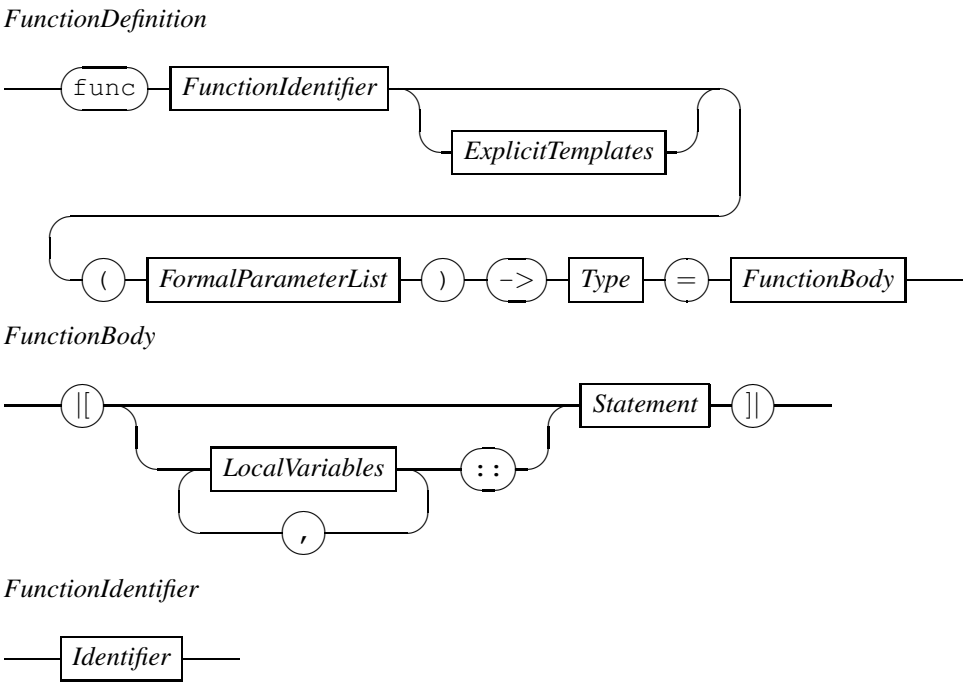

Figure 7.6: Raildiagram of *FunctionDefinition*.

LocalVariables block. It also contains a *Statement*[page 51] block.

Due to the mathematical nature of functions, the formal parameters[page 68] are always of the `val` variable class (that is, they are always call-by-value). The local variables[page 69] in the *LocalVariables* part of the *FunctionBody* may only be of class `var`. Other variable classes are useless, since time does not progress in a function. The *Statement*[page 51] block of a function body forms the implementation of the function, that is, it describes the algorithm used to compute the value. This should be done in a strictly mathematical way, that is, without side effects and without being influenced by external factors. To ensure this as much as possible, the formal parameters of the function definition are always call-by-value, and statements in a function definition may not block or delay (otherwise time would progress as side effect), the value `time` may not be used (otherwise the computed value may depend on its value), and communication with the outside world is also not possible (again to prevent influences from outside the function). In addition, the algorithm should be deterministic, that is, each time the function is called with the same arguments, the same answer should be returned. For this reason, drawing values from a distribution is also prohibited. Last but not least, the last statement executed in a function must be a return statement[page 60] to deliver the computed value to its caller.

Examples

```
func add7(val i: nat) -> nat =
|[ var k: nat = 7 :: ret i + k ]|
```

□

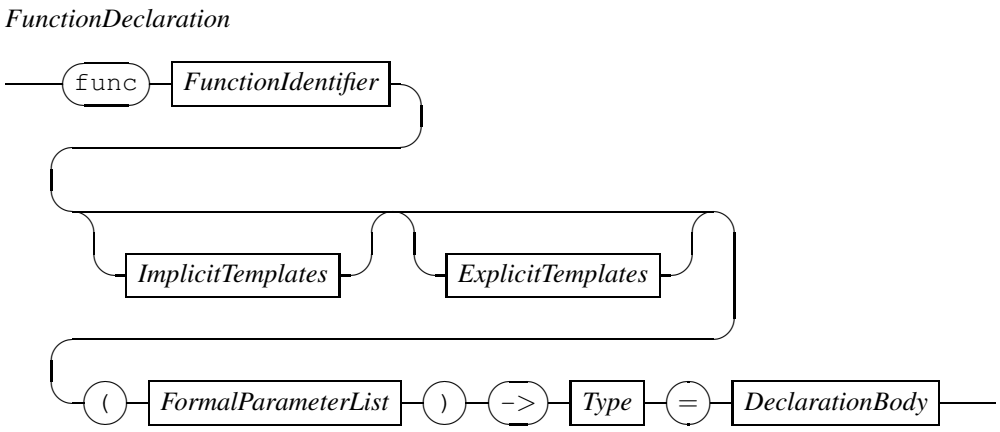

Figure 7.7: Raildiagram of *FunctionDeclaration*.

7.5.2 Function declaration

A *FunctionDeclaration* block shown in Figure 7.7 is used to declare a function (that is, state that a function exists somewhere without defining its implementation). Like the function definition above, a function declaration describes a computation without side effects. The *FunctionIdentifier*[page 78] states the name of the function being declared. The optional *ImplicitTemplates*[page 84] and *ExplicitTemplates*[page 84] are the statically decided (during the static semantics phase) parameters of the function. The parameters in the *FormalParameterList*[page 68] are (like the function definition) discrete call-by-value arguments filled in at the time of the function application (that is, at the moment the function is called as part of an expression evaluation).

Examples

```
func chisqrt(val v: real) -> real = { math.py, } :: sqrt
```

This declares the existence of a function 'chisqrt' which takes a `real` value and returns a `real` value. Its definition is called 'sqrt' in a file called math.py.

□

7.6 Processes

Processes are used to describe behavior (in time). Like functions, the same kind of behavior is often needed at multiple places in the specification. Rather than writing the same behavior more than once, behavior can be instantiated from a definition or declaration at run time. A *process definition* has an explicit description of its behavior (in its statement). A *process declaration* has no explicit behavioral description. The former is used to describe behavior in the Chi language, the latter is used to state that some behavior exists with a certain name elsewhere.

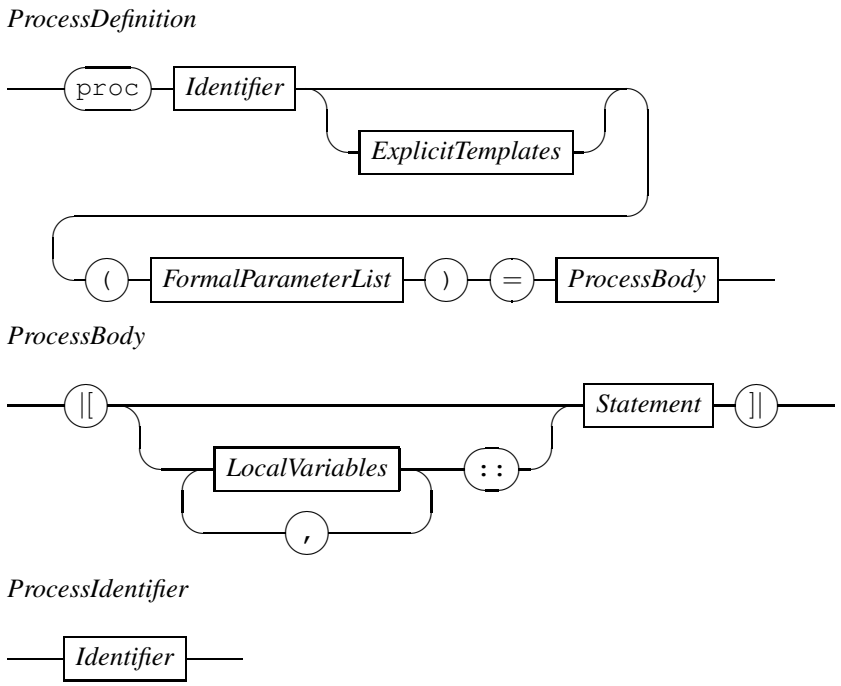

Figure 7.8: Raildiagram of *ProcessDefinition*.

The sections below explain the process definition and the process declaration syntax. Instantiation of behavior is commonly known as *process instantiation*, and is explained in Chapter 5 that explains the statements.

7.6.1 Process definition

A process definition groups a collection of behavior under a name, and allows this behavior to be instantiated many times with very little effort. Figure 7.8 shows the syntax of a process definition. The *ProcessIdentifier* block of a process definition states the name used to refer to this behavior. Its syntax is the same as an *Identifier*[page 8]. The *ExplicitTemplates*[page 84] and the *FormalParameterList*[page 68] are used to parameterize the behavior. The former parameters are constructed statically during static semantics checking, the latter parameters are exchanged during process instantiation. The behavior is described explicitly in the *ProcessBody* block. Like the body of a function definition, it allows introduction of local variables using *LocalVariables* block. The final *Statement*[page 51] block is used to describe the behavior of the process in time. Often, the description will use the parameters from the *FormalParameterList*[page 68] block to interact with other processes.

7.6.2 Process declaration

A process declaration is very similar to a process definition, except that there is no body. It just states 'somewhere, an implementation of behavior with the following name exists'. The syntax of a process declaration is shown in the *ProcessDeclaration* diagram in Figure 7.9. Like the process definition, the *ProcessIdentifier* block states the name of the process being declared. The optional *ImplicitTemplates*[page 84] and *ExplicitTemplates*[page 84] are the statically decided (during the static semantics phase) parameters of the process. The parameters in

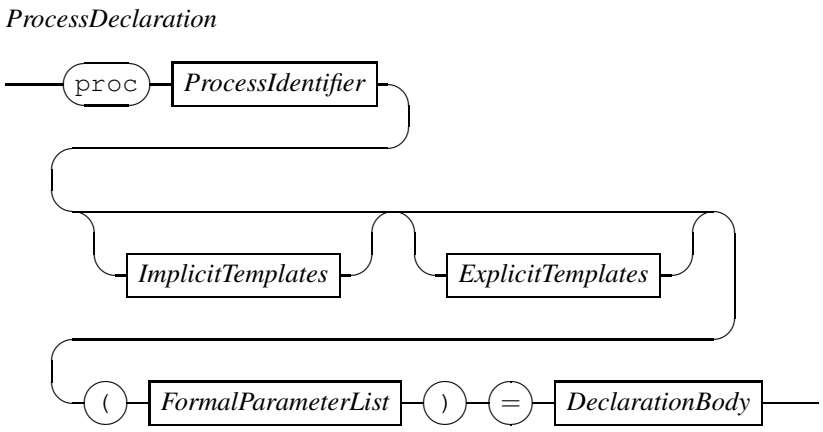

Figure 7.9: Raildiagram of *ProcessDeclaration*.

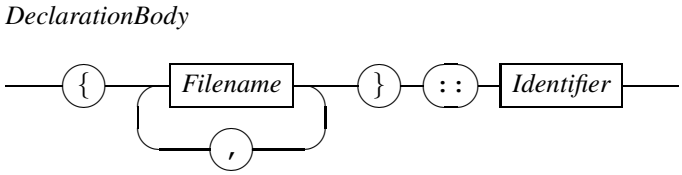

Figure 7.10: Raildiagram of *DeclarationBody*.

the *FormalParameterList*[page 68] are (as in the process definition) arguments filled in at the time of process instantiation. The implementation of the process is specified in the final *DeclarationBody*[page 82] block. The contents of this block is described in Section 7.7.

WARNING: *Currently, there is no tool that supports process declarations.*

7.7 Declaration body

The body of a function or process declaration is described by the *DeclarationBody* diagram in Figure 7.10. Such a body does not describe the contents, instead it describes where to find the contents of the function or process. The list *Filename*[page 8]s specify which file(s) contain the implementation, and the *Identifier*[page 8] is an additional identification within the files. The kind of files admissible for stating an implementation for a function or a process is not part of the language. Instead, each of the Chi tools has its own way of coupling a Chi function or process declaration to its implementation. More detailed information about this link can be found in the tool manual of each target.

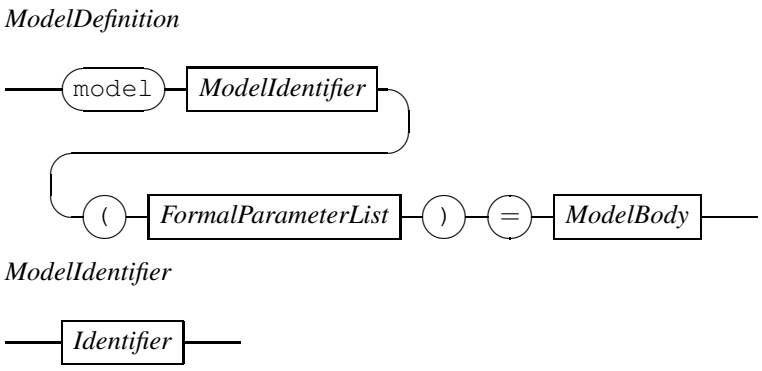

Figure 7.11: Raildiagram of *ModelDefinition*.

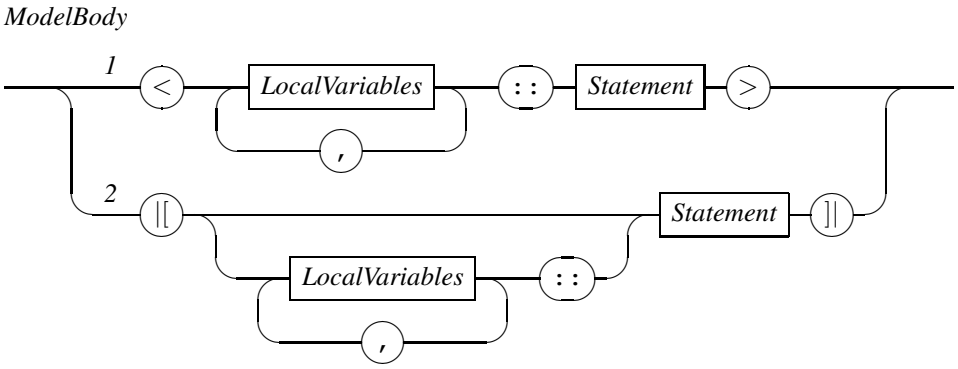

Figure 7.12: Raildiagram of *ModelBody*.

7.8 Models

Models are used to describe experiments, that is they define which behavior is performed by the specification. Such a model is defined using a model definition. Its syntax is shown in the *ModelDefinition* diagram in Figure 7.11. The name of the model is stated in the *ModelIdentifier* block (which is the same as an *Identifier*[page 8] block). Run-time experiment parameters can be defined in the *FormalParameterList*[page 68] block. The behavior of the specification is stated in the *ModelBody*[page 83] explained below. Unlike process and function definitions, a model has no template parameters. The formal parameters of the *FormalParameterList*[page 68] block are filled in at startup of the experiment.

The actual behavior of a model is defined in a *ModelBody*. The syntax of this block is shown in Figure 7.12. As shown in the figure, models come in two forms (as defined in [3]). The most basic form is the process triple $\langle p, \sigma, E \rangle$. The Chi equivalent of this triple is shown at Track 1, and often refered to as the *raw model definition*. The reason for this name is that the process triple is extremely basic. A number of properties normally assumed with Chi models are not ensured with this primitive (a raw model definition allows a send without a corresponding receive or a receive without a send, it does not favor actions above delays, and `time` does not exist by default, you must add it yourself).

To make life easier for users, [3] also defines a syntactical extension of such a process triple.

ImplicitTemplates

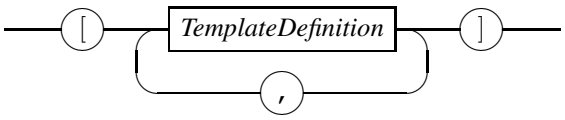

Figure 7.13: Raildiagram of *ImplicitTemplates*.

ExplicitTemplates

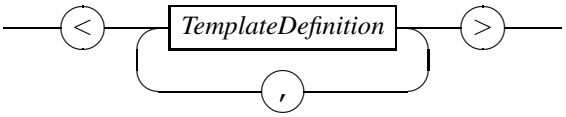

Figure 7.14: Raildiagram of *ExplicitTemplates*.

The Chi equivalent of this extended process triple is shown at Track 2. This form is known as the *cooked model definition*. The latter form ensures that only communication can happen (with a combined sender and receiver), delays only happen when no action can take place, `time` exists and is initialized to 0.

WARNING: *The raw model definition is currently not supported.*

7.9 Template definitions

Function and process definitions and declarations can have statically decided parameters to parameterize the definitions and declarations. Such parameters are called template parameters named after a similar mechanism in C++. There are two forms of definitions of template parameters, namely implicit template definitions and explicit template definitions. The former are defined in an *ImplicitTemplates* block shown in Figure 7.13, the latter are defined in an *ExplicitTemplates* block shown in Figure 7.14. The difference between both kinds of definitions is in how they obtain their value. Implicit template definitions are computed by the type system, explicit template definitions are (explicitly) stated in the specification by the modeler.

The syntax of a template definition is shown in the *TemplateDefinition* diagram in Figure 7.15. Template definitions exist in two variants, namely as value template definitions defined at Track 1, and type template definitions defined shown at Track 2. Value template definitions are used to introduce a value of the type indicated by the *Type*[page 11] block at Track 1. Type template definitions state a type.

WARNING: *Templates are not implemented yet except for implicit templates in libraries.*

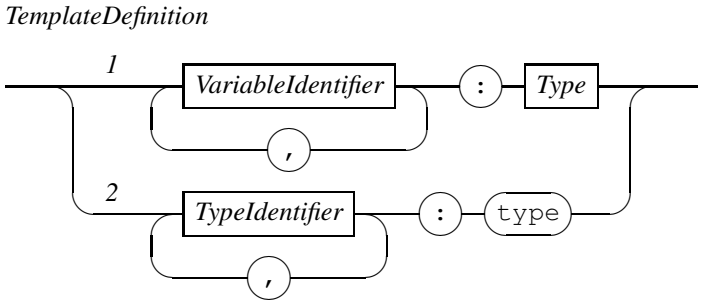

Figure 7.15: Raildiagram of *TemplateDefinition*.

Bibliography

[1] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge University Press, Cambridge, 2005.

[2] Averill M. Law and David Kelton. *Simulation modeling and analysis*. McGraw-Hill, New York, 1991.

[3] K.L. Man and R.R.H. Schiffelers. *Formal specification and analysis of hybrid systems*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, 2006.

Appendix A
Distributions

In the tables below, the available distributions are listed. The first table lists constant distributions, that is, distributions that always return the same value. While they are not very good at creating pseudo-random behavior, they may be useful for debugging purposes. The second table lists the discrete distributions, that is, distributions returning a discrete value such as a boolean or an integer number. Finally, the third and last table lists the available continuous distributions.

For ease of reference, the name of the distribution as it is defined in Law & Kelton ([2]), and the mean, variance, and range of the samples is also shown in terms of the arguments of the distribution function.

A.1 Constant distributions

Function	Description
Constant distribution returning the specified value `constant(b: bool) -> (-> bool)`	Law & Kelton: - Mean: b Variance: 0 Range: {true, false}
Constant distribution returning the specified value `constant(n: nat) -> (-> nat)`	Law & Kelton: - Mean: n Variance: 0 Range: $[0, \infty)$
Constant distribution returning the specified value `constant(i: int) -> (-> int)`	Law & Kelton: - Mean: i Variance: 0 Range: $(-\infty, \infty)$
Constant distribution returning the specified value `constant(r: real) -> (-> real)`	Law & Kelton: - Mean: r Variance: 0 Range: $(-\infty, \infty)$

A.2 Discrete distributions

Function	Description
Bernoulli distribution with chance $p \in [0, 1]$ for true	
`bernoulli(p: real) -> (-> bool)`	Law & Kelton: Bernoulli(p) Mean: p Variance: p*(1-p) Range: {true, false}
Bernoulli distribution with chance $p \in [0, 1]$ for 1	
`bernoulli(p: real) -> (-> nat)`	Law & Kelton: Bernoulli(p) Mean: p Variance: p*(1-p) Range: $\{0, 1\}$
Binomial distribution with t>0 experiments with chance $p \in [0, 1]$	
`binomial(p: real, t: nat) -> (-> nat)`	Law & Kelton: bin(t, p) Mean: $t * p$ Variance: $t * p * (1 - p)$ Range: $\{0, 1, 2, \ldots, t\}$
Geometric distribution, number of failed Bernoulli(p) experiments with chance $p \in [0, 1]$ before first succes	
`geometric(p: real) -> (-> nat)`	Law & Kelton: geom(p) Mean: $(1 - p)/p$ Variance: $(1 - p)/(p * p)$ Range: $\{0, 1, 2, \ldots\}$
Poisson distribution with rate r>0	
`poisson(r: real) -> (-> nat)`	Law & Kelton: P(r) Mean: r Variance: r Range: $\{0, 1, 2, \ldots\}$
Discrete uniform distribution with $a < b$	
`uniform(a, b: nat) -> (-> nat)`	Law & Kelton: DU(a, b-1) Mean: $(a + b - 1)/2$ Variance: $((b - a)^2 - 1)/12$ Range: $\{a, a + 1, a + 2, \ldots, b - 1\}$
Discrete uniform distribution with $a < b$	
`uniform(a, b: int) -> (-> int)`	Law & Kelton: DU(a, b-1) Mean: $(a + b - 1)/2$ Variance: $((b - a)^2 - 1)/12$ Range: $\{a, a + 1, a + 2, \ldots, b - 1\}$

A.3 Continuous distributions

Function	Description
Beta distribution with shape parameters $a > 0$ and $b > 0$	
`beta(a, b: real) -> (-> real)`	Law & Kelton: B(a, b) Mean: $a/(a+b)$ Variance: $a * b/((a+b)^2 * (a+b+1))$ Range: $[0, 1]$
Erlang distribution with parameter $m > 0$ and scale parameter $b > 0$, Also known as Gamma(m, b)	
`erlang(m: nat, b: real) -> (-> real)`	Law & Kelton: m-Erlang(b) Mean: $m * b$ Variance: $m * b^2$ Range: $[0, \infty)$

Continuous distributions, continued

Function	Description
Negative exponential distribution with scale parameter $b > 0$	
`exponential(b: real) -> (-> real)`	Law & Kelton: expo(b) Mean: b Variance: b^2 Range: $[0, \infty)$
Gamma distribution with shape parameter $a > 0$ and scale parameter $b > 0$	
`gamma(a, b: real) -> (-> real)`	Law & Kelton: Gamma(a, b) Mean: $a * b$ Variance: $a * b^2$ Range: $[0, \infty)$
Lognormal distribution with $b > 0$	
`lognormal(a, b: real) -> (-> real)`	Law & Kelton: LN(a, b) Mean: $\exp(a + b/2)$ Variance: $e^{2*a+b} * (e^b - 1)$ Range: $[0, \infty)$
Normal distribution with $b > 0$	
`normal(a, b: real) -> (-> real)`	Law & Kelton: N(a, b) Mean: a Variance: b Range: $(-\infty, \infty)$
Triangle distribution from a to c with the top at b, $a < b < c$, and $a > 0$	
`triangle(a, b, c: real) -> (-> real)`	Law & Kelton: triang(a, c, b) Mean: $(a + b + c)/3$ Variance: $(a^2 + b^2 + c^2 - a * b - a * c - b * c)/18$ Range: $[a, c]$
Random distribution, mainly intended for developers	
`random() -> (-> real)`	Law & Kelton: U(0, 1) Mean: $1/2$ Variance: $1/12$ Range: $[0, 1)$
Continuous uniform distribution from a to b, with $a < b$	
`uniform(a, b: real) -> (-> real)`	Law & Kelton: U(a, b) Mean: $(a + b)/2$ Variance: $(b - a)^2/12$ Range: $[a, b)$
Weibull distribution with shape parameter $a > 0$ and scale parameter $b > 0$	
`weibull(a, b: real) -> (-> real)`	Law & Kelton: Weibull(a, b) Mean: $b/a * \Gamma(1/a)$ Variance: $(b^2)/a * (2 * \Gamma(2/a) - (\Gamma(1/a)^2)/a)$ Range: $[0, \infty)$

In the `weibull` function, $\Gamma(z)$ is the gamma function, defined as $\Gamma(z) = \int_0^\infty t^{(z-1)} * e^{-t} dt$ for all real numbers $z > 0$.

Index

Below is the index of the reference manual. Bold page numbers refer to definitions of the term, normal page numbers refer to use of the term.