Chi 2.0 language reference manual

Chi 2.0 reference manual

A.T. Hofkamp, J.E. Rooda,
R.R.H. Schiffelers, and D.A. van Beek

SE-Report: 2008-02

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 1]. If you want to know more about railroad diagrams, you can go to page 1 to read about its details.

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

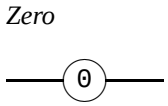

Figure 1.1: Raildiagram of *Zero*.

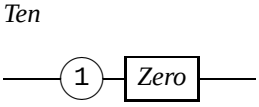

Figure 1.2: Raildiagram of *Ten*.

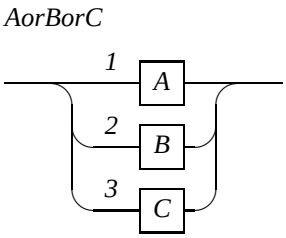

Figure 1.3: Raildiagram of *AorBorC*.

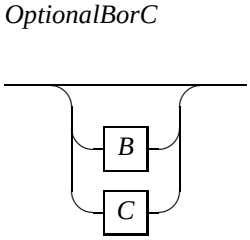

Figure 1.4: Raildiagram of *OptionalBorC*.

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 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.

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

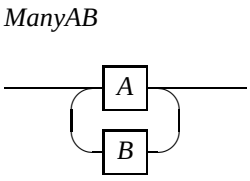

Figure 1.5: Raildiagram of *ManyAB*.

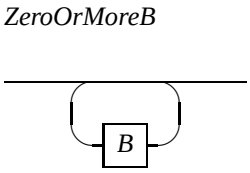

Figure 1.6: Raildiagram of *ZeroOrMoreB*.

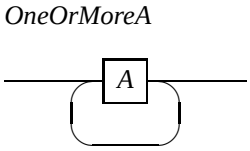

Figure 1.7: Raildiagram of *OneOrMoreA*.

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

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

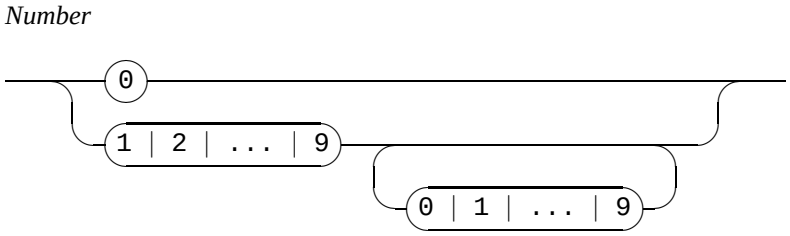

Figure 2.1: Raildiagram of *Number*.

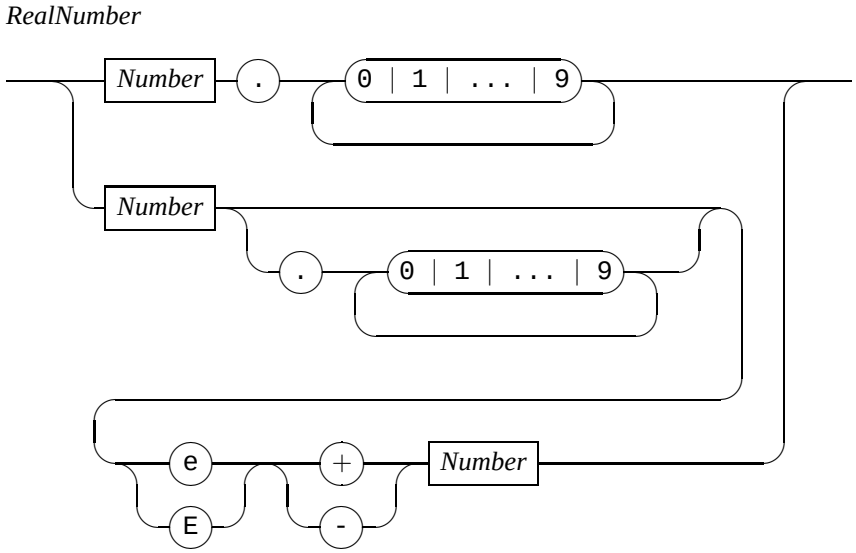

Figure 2.2: Raildiagram of *RealNumber*.

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.

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.

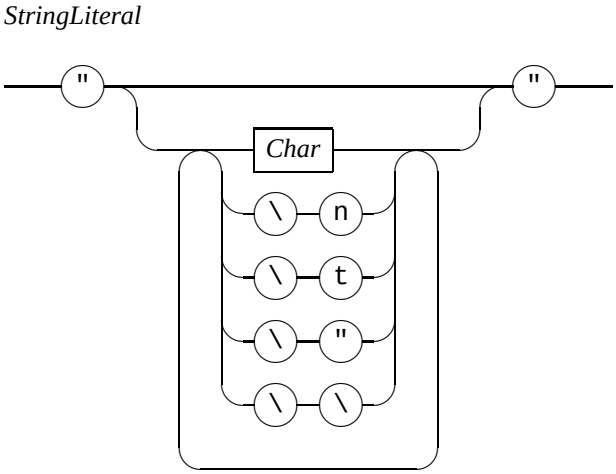

Figure 2.3: Raildiagram of *StringLiteral*.

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.

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

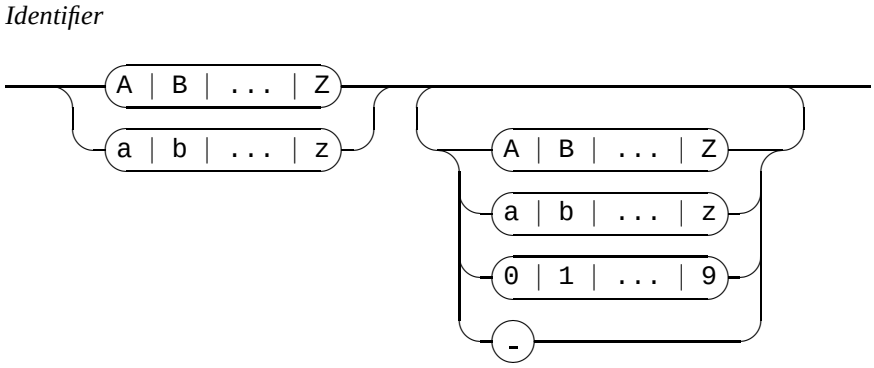

Figure 2.4: Raildiagram of *Identifier*.

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 1], 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 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 2.6. The final path element must start with a *ModuleIdentifier* (which

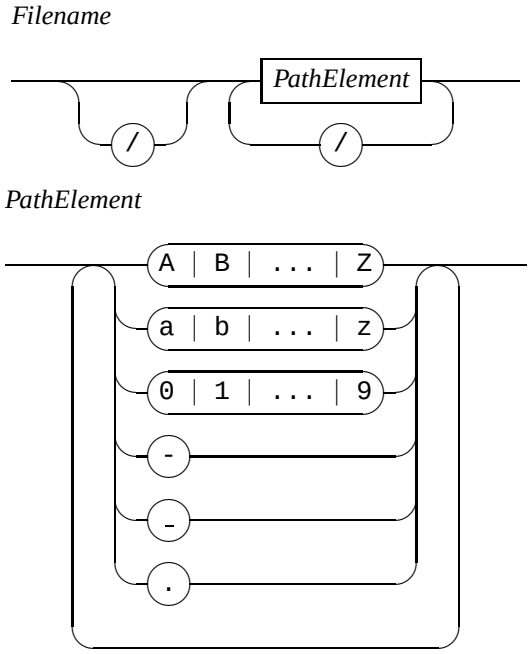

Figure 2.5: Raildiagram of *Filename*.

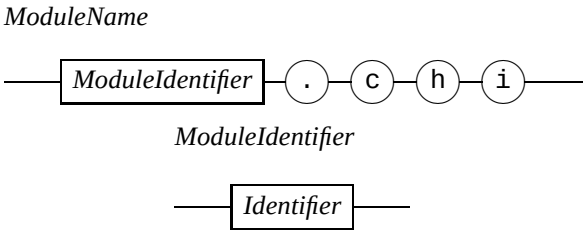

Figure 2.6: Raildiagram of *ModuleName*.

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/bin/chic`	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

In the Chi language, types consist of two parts, a static part and a dynamic part. The static part of a type, often abbreviated to *static type*, is commonly known as 'type' in many programming languages, for example `bool` or `string`. It defines the kind of data values that can be used.

The dynamic part of a type, also known as *dynamic type*, defines what happens with values when time progresses. The dynamic part is only relevant for variables and formal parameters of processes and models since only those entities contain data values within a running Chi program.

Static types are explained in Section 3.1, dynamic types are explained in Section 3.7.

3.1 Static type

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 5,

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

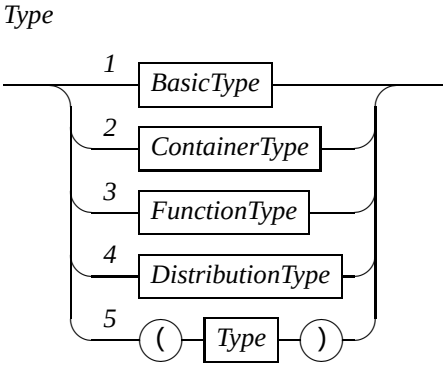

Figure 3.1: Raildiagram of *Type*.

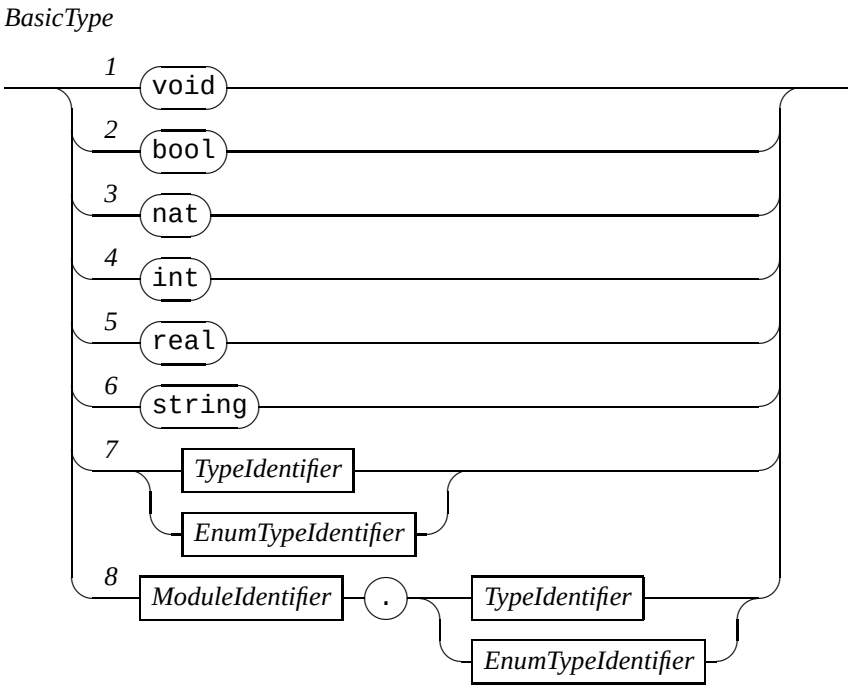

Figure 3.2: Raildiagram of *BasicType*.

the *BasicType*[page 12] block, container types[page 13] (types whose values contain values of other types) are defined in the *ContainerType*[page 13] 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 42] are available from the *DistributionType*[page 15] block. Finally, Track 5 allows grouping of types through the use of brackets.

3.2 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 22] (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 28] (values of the *real number type*, the set $\mathbb{R}$), and strings[page 31] (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 75]. Track 8 does the same, except that it refers to a type[page 11] imported[page 76][page 76] from another module.

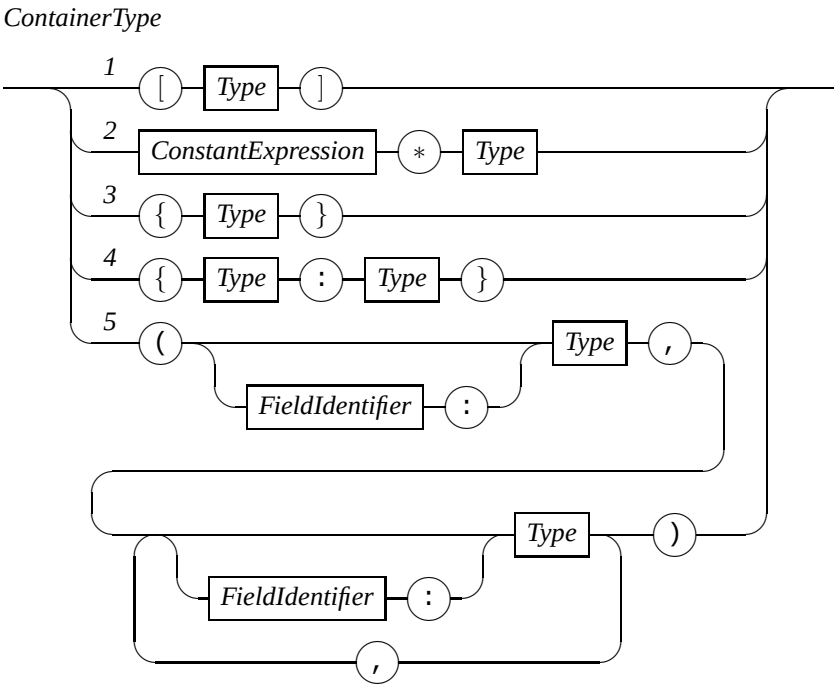

Figure 3.3: Raildiagram of *ContainerType*.

3.3 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 at the front and back of a *list*[page 32] (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 47] 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 38] (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

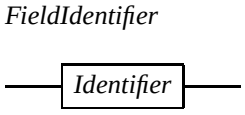

Figure 3.4: Raildiagram of *FieldIdentifier*.

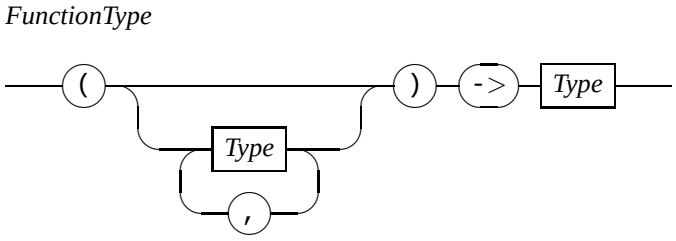

Figure 3.5: Raildiagram of *FunctionType*.

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 14] (each value of the key type[page 13] present in the dictionary) an associated value of the value type[page 14] is available.

Last but not least in the container types is the *record tuple type*, shown at Track 5. It consists of one or more *RecordField* blocks. A single *RecordField*[page 14] block contains one or more record field values of the specified type. In the latter case, each record field value is given a name. A record type *must have at least two record field values.* A value of a record type (consisting of two or more record field values) are commonly refered to as *record tuples*,[page 36] *tuples* or *records.* Its use is in combining a number of values together and allow treatment of such a combination of values 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.*

3.4 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 79] or declaration[page 80], except that only the type signature of the formal parameters[page 83] is given (that is, only the types of the formal parameters are stated rather than full variable declarations).

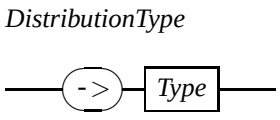

Figure 3.6: Raildiagram of *DistributionType*.

3.5 Distribution type

The final type is the distribution type, the type used for storing stochastic distributions[page 42]. 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).

3.6 Type widening

In the *BasicType*[page 12] diagram, there are three different numeric data types listed, namely `nat` (the set $\mathbb{N}$), `int` (the set $\mathbb{Z}$), and `real` (the set $\mathbb{R}$). In mathematics, these sets are sub-sets of each other, $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{R}$. In Chi, this relation also exists in the form of type widening. Informally, type widening means that at any place where you use a value, you can also use a value of a smaller type (a type that is a sub-set of the former value).

For example, in $1 + 1.5$, the addition is performed on two real numbers. The first argument (with value 1) is a natural number, but since $\mathbb{N} \subseteq \mathbb{R}$, the value is silently widened to a an equivalent real number before adding both values. The type widening strategy is 'locally minimal'. This means that at every point in the expression, types are widened as little as possible. For example the expression '`1.5 + 2 * 3`' is interpreted as '`1.5 + toreal(2 * 3)`', rather than '`1.5 + toreal(2) * toreal(3)`'.

Type widening is performed recursively over most data types. For example, a list with natural numbers may be used at any place where a list of real numbers is expected. The exceptions to this rule are distribution types and function types which are never widened (for example `-> nat` $\not\subseteq$ `-> int`).

For precise control of types, a function `settype` exists.

Function	Description
`func settype<T: type>(val x: T) -> T`	Fixate result-type to specified type `T`

Note: This function is special in the sense that its output may not be type widened. In this way, one can force a specifed type at some point in an expression.

3.7 Dynamic type

In the Chi language three kinds of dynamic type exist. They are listed in the table below.

Value	Description
`disc`	This kind describes the behavior of a discrete variable. It has no derivative, and does not change its value while time progresses.
`cont`	This kind describes the behavior of a continuous variable. Such a variable has a derivative, its value changes in a non-jumping fashion along a trajectory limited by the equations that are active at the moment a time step is taken.
`alg`	This kind describes the behavior of an algebraic variable. Such a variable has no derivative. Its value is fully controlled by the active equations, and may jump arbitrarily (within the limits set by the active equations).

The dynamic part of a type is used with variable declarations, explained in Section 5.1. The presence of a derivative in the `cont` kind of the dynamic types makes it necessary to limit the static type part of such variables to the types that can have a derivative, namely real numbers and vectors thereof.

In many cases, writing both the dynamic type and the static type gives little or no extra information. For this reason, two additional shorthands are allowed:

- If no dynamic type is provided with a variable or parameter, its dynamic type is `disc`.
- For variables and parameters of dynamic type `cont`, if no static type is given, it is of type `real`.

For readability of the diagrams, these shorthand notations are not used in this manual.

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 for the data type, thus creating a complete overview of available functionality with each data type.

Not included in the discussion here is the type-widening facility, where natural values may be used in places where integer or real values are expected, or where integer values may be used in places where real values are expected. For details of type widening, see Section 3.6 in the previous chapter.

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 4.18. 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 4.19.

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 45] listed in Section 4.18. 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 used for refering to 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. The variable identifier is not shown here since you cannot refer to variables living in another module.

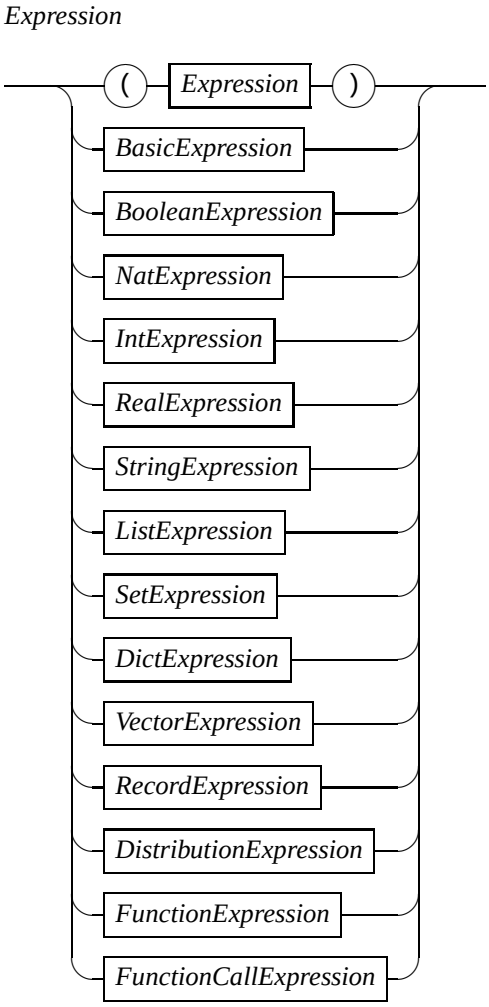

Figure 4.1: Raildiagram of *Expression*.

Track 3 shows the syntax of the unary derivative operator. This operator can only be applied to addressable (Section 4.19) 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 expression folding. Details of folding can be found in Section 4.2. Finally, Track 6 adds the syntax of conditional expression to the language. These expressions are explained in more detail in Section 4.3.

WARNING: *Module prefix of Track 2 and conditional expressions of Track 6 are not implemented yet.*

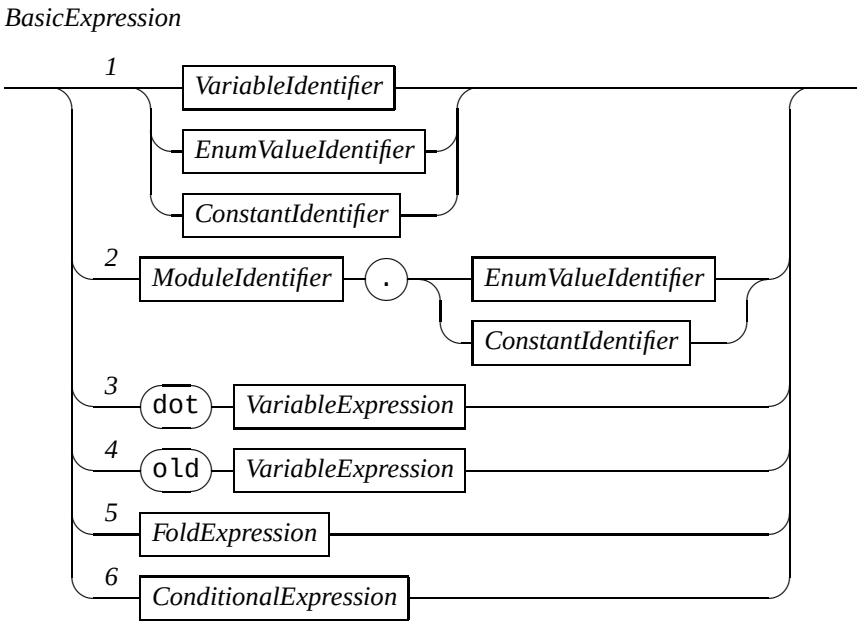

Figure 4.2: Raildiagram of *BasicExpression*.

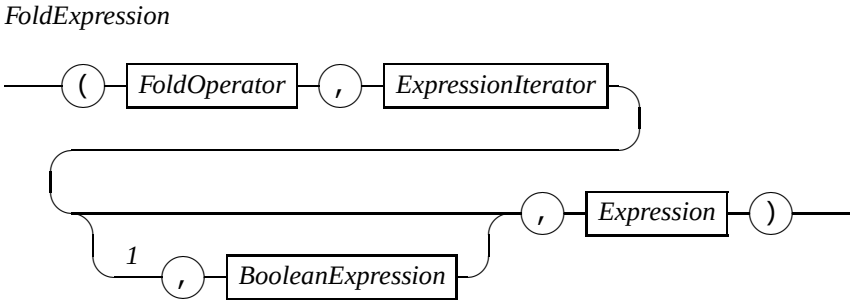

Figure 4.3: Raildiagram of *FoldExpression*.

4.2 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 4.3. As you can see in the diagram, the fold expression consists of four parts, a *FoldOperator*[page 20], 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.

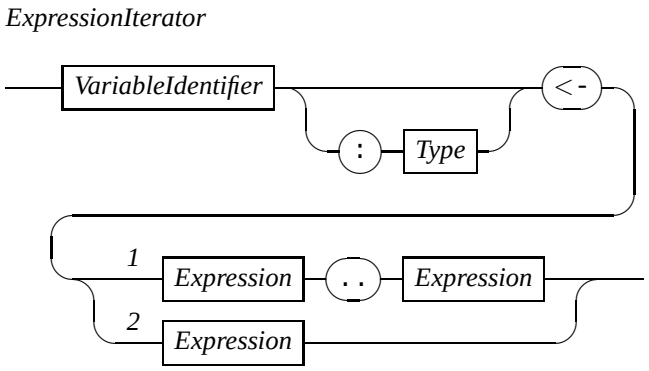

Figure 4.4: Raildiagram of *ExpressionIterator*.

4.2.1 Expression iterator

The *ExpressionIterator* diagram used to specify where values are obtained from, is shown in Figure 4.4. The folding starts with a variable denoted by a *VariableIdentifier* which is assigned each value that must be processed. This variable can be used in the guard and the *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 64] 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 13]), as shown at Track 2.

4.2.2 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 *BooleanExpression*[page 22] 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.*

4.2.3 Folding

The values that are not discarded during processing are folded together using the *FoldOperator*, which is either one of seven possible binary operators in expressions, or a user-defined function.

The table below lists them (first column), last row shows the user-defined function.

Operator	Initial	Purpose
$+$	0	Add all values together
$*$	1	Multiply all values with each other
`&`	true	Test whether all values are true
`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
$\backslash/$	{}	Collect all values into a set
(f, i)	i	Execute $i := f(v_j, i)$ for values v_j from the iterator

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)`	$((0 + 3 * 2) + 4 * 2) + 5 * 2 = 24$

□

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.

4.3 Conditional expressions

With conditional expressions, you can return different values of the same type based on conditions.

The syntax of the *ConditionalExpression* is shown in Figure 4.5. A conditional expression consists of a list of conditional alternatives seperated by a vertical bar. Each alternative consists of two expressions seperated by an arrow '`->`'. The first expression is a boolean guard, the second expression denotes the returned value if the guard is true. A conditional expression should be complete, that is, every time the construct is evaluated, the guard of at least one of the conditional alternatives must hold. Also, if more than one guard is true, their associated returned values

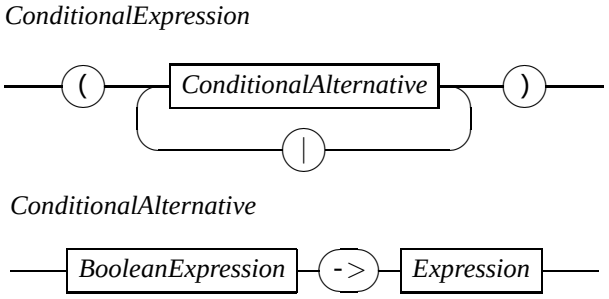

Figure 4.5: Raildiagram of *ConditionalExpression*.

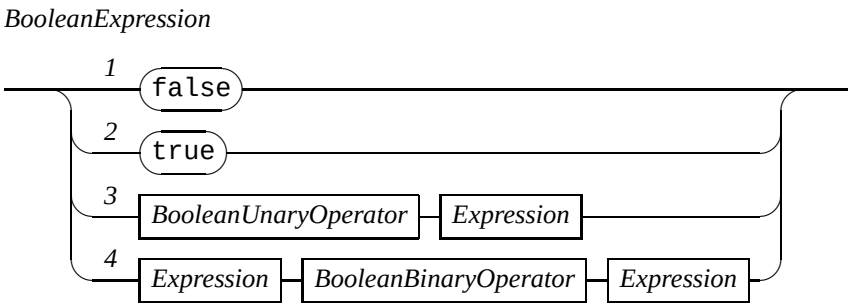

Figure 4.6: Raildiagram of *BooleanExpression*.

should be the same (that is, non-deterministically chosing a conditional alternative may not lead to different results).

Examples

Example	Result
`(x < 0 -> 0 | x >= 0 -> x)`	`x max 0`

□

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 22] 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 22] 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 `&` e_2	`bool`	`bool`	`bool`	$e_1 \wedge e_2$, conjunction operator
e_1 `and` e_2	`bool`	`bool`	`bool`	conditional conjunction operator, only evaluates e_2 when e_1 holds
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`

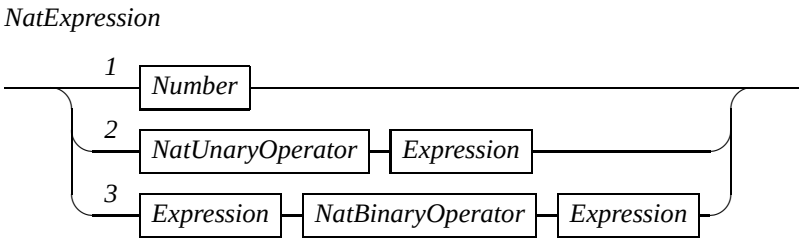

Figure 4.7: Raildiagram of *NatExpression*.

4.4.3 Boolean functions

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

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

Examples

Example	Result
`tostring(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`.

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 24] 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`	`nat`	Unary addition operator, does nothing

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 24] 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

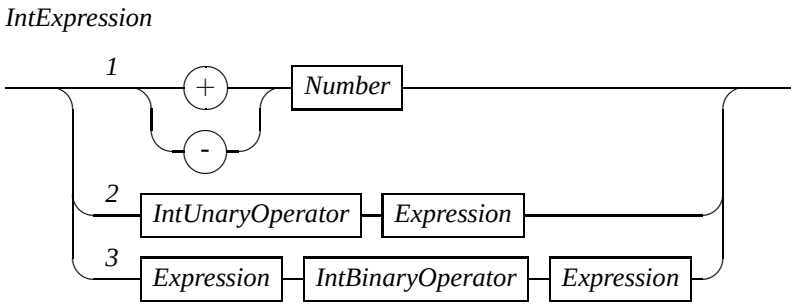

Figure 4.8: Raildiagram of *IntExpression*.

list, all the usual mathematical operations can be performed.

4.5.3 Natural number functions

Function	Description
`func tostring(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 combination 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 26] 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, does nothing

The unary operators of the integer numbers are the same as the unary operators of the natural numbers[page 24]. 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	`int`	`int`	`int`	$e_1 \times e_2$
e_1 `/` e_2	`int`	`int`	`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	`int`	`int`	`int`	$e_1 + e_2$
e_1 `-` e_2	`int`	`int`	`int`	$e_1 - e_2$
$e_1 < e_2$	`int`	`int`	`bool`	$e_1 < e_2$
e_1 `<=` e_2	`int`	`int`	`bool`	$e_1 \leq e_2$
e_1 `=` e_2	`int`	`int`	`bool`	$e_1 = e_2$
e_1 `/=` e_2	`int`	`int`	`bool`	$e_1 \neq e_2$
e_1 `>=` e_2	`int`	`int`	`bool`	$e_1 \geq e_2$
$e_1 > e_2$	`int`	`int`	`bool`	$e_1 > e_2$

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 on integer values.

□

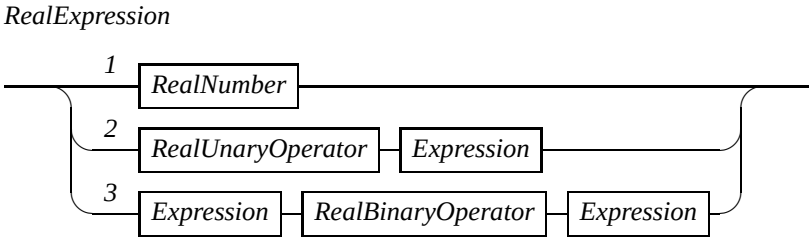

Figure 4.9: Raildiagram of *RealExpression*.

4.6.3 Integer number functions

Function	Description
`func tonat(val i: int) -> nat`	Convert integer number value to natural number value
`func tostring(val i: int) -> 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. Literal real values can be introduced using the *RealNumber*[page 6] block at Track 1. Unary operators (`+` and `-`) can be applied on real values using Track 2. 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 3. These expressions are further explained in Section 4.7.2.

Besides the declared real values, the continuous (read-only) real variable `time` is always available. It represents the current simulation time (in user-defined time-units). In simulators, its value starts at 0. Other tools may use a different starting point for `time`.

4.7.1 Real number unary expressions

The unary expressions on real numbers at Track 3 in the *RealExpression*[page 28] 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 28] 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	`real`	`real`	`real`	${e_1}^{e_2}$
e_1 `*` e_2	`real`	`real`	`real`	$e_1 * e_2$
e_1 `/` e_2	`real`	`real`	`real`	Real division e_1/e_2
e_1 `+` e_2	`real`	`real`	`real`	$e_1 + e_2$
e_1 `-` e_2	`real`	`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	`real`	`real`	`real`	$e_1 < e_2$
e_1 `<=` e_2	`real`	`real`	`real`	$e_1 \leq e_2$
e_1 `=` e_2	`real`	`real`	`real`	$e_1 = e_2$
e_1 `/=` e_2	`real`	`real`	`real`	$e_1 \neq e_2$
e_1 `>=` e_2	`real`	`real`	`real`	$e_1 \geq e_2$
e_1 `>` e_2	`real`	`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

The following functions exist for values of type `real`.

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 tostring(val r: real) -> string`	Convert r to its string representation
`func tostring(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

□

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`	$\text{acos}(r)$
`func acos(val r: real) -> real`	$\text{asin}(r)$
`func atan(val r: real) -> real`	$\text{atan}(r)$

Examples

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

□

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`	$\text{acos}(r)$
`func acos(val r: real) -> real`	$\text{asin}(r)$
`func atan(val r: real) -> real`	$\text{atan}(r)$

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`	$\text{acosh}(r)$
`func acosh(val r: real) -> real`	$\text{asinh}(r)$
`func atanh(val r: real) -> real`	$\text{atanh}(r)$

Other math functions

Function	Description
`func abs(val r:real) -> real`	$\lvert r\rvert$, 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 rmod(val r, s: real) -> real`	$r \bmod s \equiv r - s \times \lfloor r/s \rfloor$

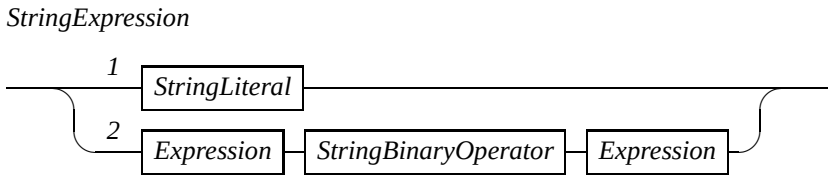

Figure 4.10: Raildiagram of *StringExpression*.

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.

4.8.1 String binary expressions

The binary expressions on strings of Track 2 in the *StringExpression*[page 31] 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"`

□

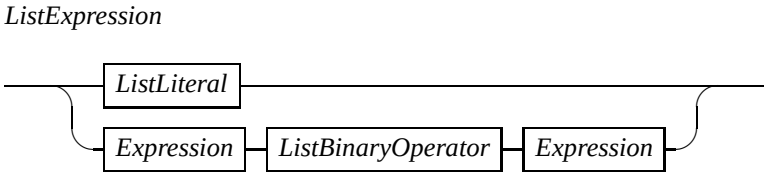

Figure 4.11: Raildiagram of *ListExpression*.

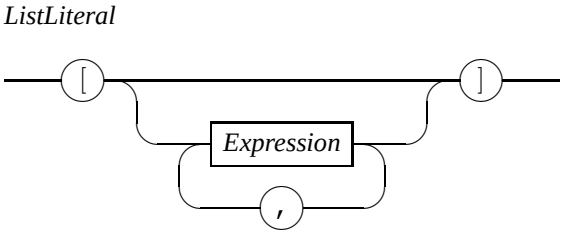

Figure 4.12: Raildiagram of *ListLiteral*.

4.8.2 String functions

Conversion functions

Function	Description
`tobool(val s: string) -> bool`	Convert string with a boolean literal to its equivalent value
`tonat(val s: string) -> nat`	Convert string with a natural number literal to its equivalent value
`toint(val s: string) -> int`	Convert string with an integer number literal to its equivalent value
`toreal(val s: string) -> real`	Convert string with a real number literal to its equivalent value

Other string functions

Function	Description
`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 13]. 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 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 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 *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 39] 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 $m = 1 \wedge \forall x_i : x_i \neq y_1$ 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 $m = 1 \wedge x_j = y_1 \wedge \forall x_i : x_i \neq y_1$ 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

Conversion functions

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

WARNING: *The list to set function* `toset` *has not been implemented. You can use expression folding*[page 19] *instead.*

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 $\texttt{f}(x, y) = x < y$)
`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 $\texttt{f}(x, y) = x < y$)

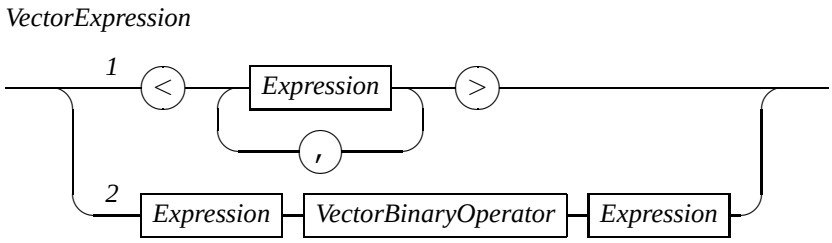

Figure 4.13: Raildiagram of *VectorExpression*.

Examples

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

```
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"
]|
```

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

□

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 35] 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 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`	Access third field (21) of vector
`<1.0, 3.0> = <1.0, 5.0>`	false, since $3.0 \neq 5.0$

□

There are no functions 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 expressions, seperated from each other with commas 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

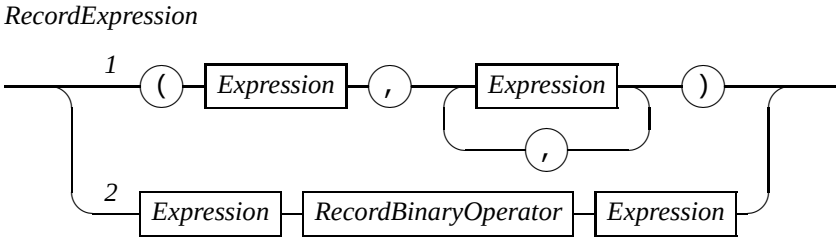

Figure 4.14: Raildiagram of *RecordExpression*.

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 36] 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	$(\alpha_1, \alpha_2, \ldots, \alpha_n)$ $(\alpha_1, \alpha_2, \ldots, \alpha_n)$	`nat` identifier	α_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 α_1, α_2, ..., α_n the type of the first, second, ... $n-1^{\text{th}}$ 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.

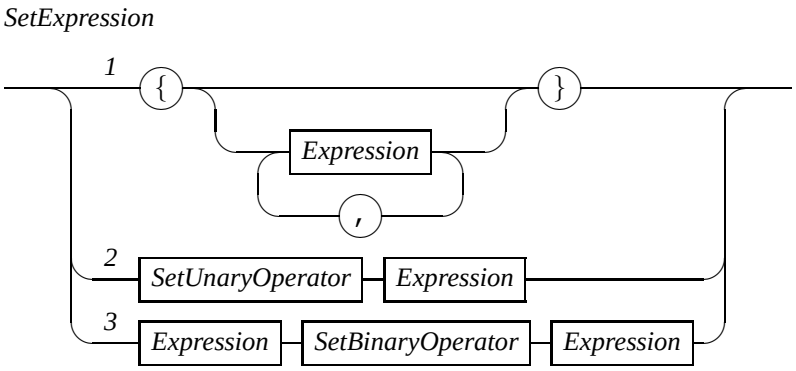

Figure 4.15: Raildiagram of *SetExpression*.

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 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 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 commas, 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

{}
`{1, 2, 4}`
`{2, 4, 1}`
`{2, 2, 1, 4}`
	The empty set (set without any values present)
	
	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 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 *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).

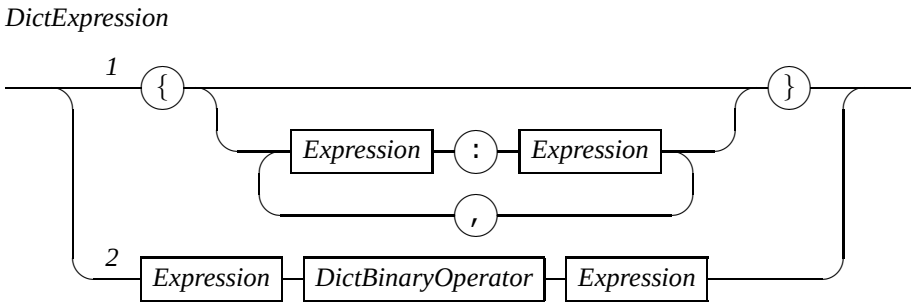

Figure 4.16: Raildiagram of *DictExpression*.

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, namely the `size` function to obtain the number of elements in a set. Its definition is shown below.

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

4.13 Dictionaries

Dictionaries are like sets,[page 38] 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 *Expression*[page 17] blocks 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 40] 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

4.13.2 Dictionary functions

Currently, there are no functions for dictionaries available.

WARNING: *Dictionaries are not yet implemented.*

4.14 Enumeration values

Enumeration values[page 74] are unique values defined in an enumeration definition[page 74]. The only operators allowed on enumeration values are the 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 74] `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.*

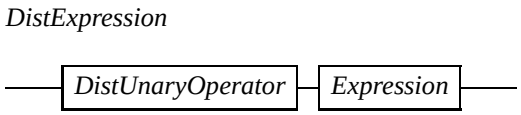

Figure 4.17: Raildiagram of *DistExpression*.

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 15]. Instead, you must use one of the distribution functions listed in Appendix A.

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 4.17. 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

4.15.2 Distribution functions

The library has two functions for manipulating values of the distribution type, namely `setseed()` and `reset()`. Their signature is listed in the table below:

Function	**Description**

`func setseed(val d: -> bool, n: nat) -> (-> bool)`
`func setseed(val d: -> nat, n: nat) -> (-> nat)`
`func setseed(val d: -> int, n: nat) -> (-> int)`
`func setseed(val d: -> real, n: nat) -> (-> real)`
	Set the seed of distribution `d`

`func reset(val d: -> bool) -> (-> bool)`
`func reset(val d: -> nat) -> (-> nat)`
`func reset(val d: -> int) -> (-> int)`
`func reset(val d: -> real) -> (-> real)`
	Reset distribution `d`

The `setseed()` function allows you to set the seed of a specific distribution to a specific value. This gives you very precise control over the seeds used within the distributions. The `reset()` function uninitializes a distribution. It resets the distribution (that is, sets it to the empty distribution, see Section A.1). It also resets the seeds used internally.

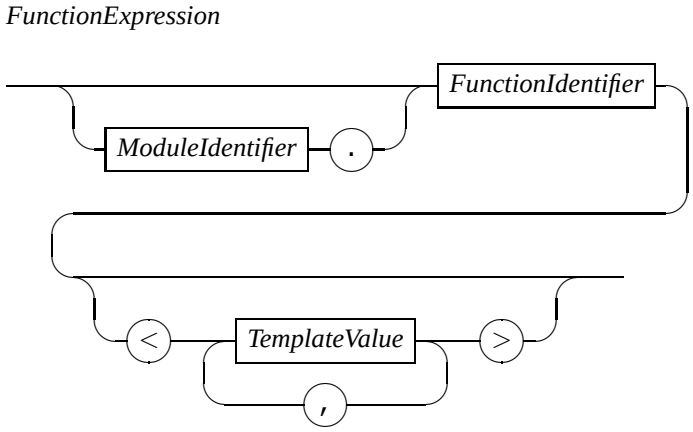

Figure 4.18: Raildiagram of *FunctionExpression*.

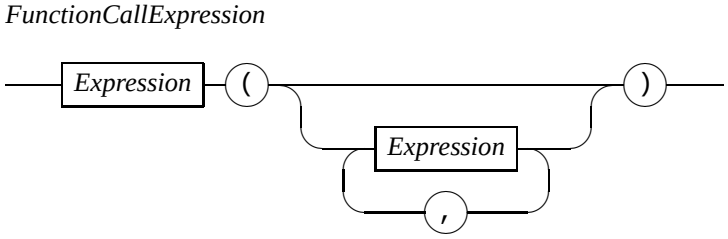

Figure 4.19: Raildiagram of *FunctionCallExpression*.

4.16 Functions

A function can be declared[page 80] or defined[page 79] 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 35] or `insert`[page 35]) or send a function over a channel.

Before discussing function calls, first the *FunctionExpression* block that represents a function (as data) is explained. Its diagram is shown in Figure 4.18. A *FunctionExpression* refers to a function with the *FunctionIdentifier*[page 79] block, optionally prefixed with a module name to allow use of functions defined in other modules. If the function has explicit template definitions, they must be given a concrete value by listing the values between triangular brackets.

The most common use of a *FunctionExpression* is to call the function it represents. The syntax of such a call is shown in the *FunctionCallExpression* diagram in Figure 4.19. A function call starts with a *FunctionExpression*[page 43] 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, formal parameter[page 83] in a function

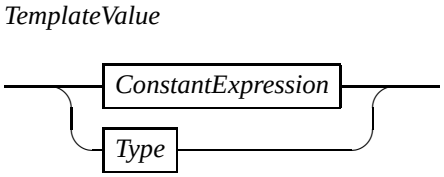

Figure 4.20: Raildiagram of *TemplateValue*.

call, or the receiving end of a communication[page 60]). 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 Template values

A *TemplateValue* block, as shown in Figure 4.20, gives a concrete value to a template parameter of implicit template definitions[page 87] or explicit template definitions[page 87] with functions and processes. Since a template values is 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 62].

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

□

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

4.18 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**	**Chi**
$1 + 2 * 3$	'$(1 + 2) * 3$' or '$1 + (2 * 3)$'	$2 * 3$
$10 - 2 - 3$	'$(10 - 2) - 3$' or '$10 - (2 - 3)$'	$10 - 2$
$\{1, 2, 3\} \backslash \{1, 2\} \backslash \{1\}$	'$(\{1, 2, 3\} \backslash \{1, 2\}) \backslash \{1\}$' or '$\{1, 2, 3\} \backslash (\{1, 2\} \backslash \{1\})$'	$\{1, 2, 3\} \backslash \{1, 2\}$
$\{1, 2, 3\} \backslash \{1, 2\} / \backslash \{1\}$	'$(\{1, 2, 3\} \backslash \{1, 2\}) / \backslash \{1\}$' or '$\{1, 2, 3\} \backslash (\{1, 2\} / \backslash \{1\})$'	$\{1, 2\} / \backslash \{1\}$

The third column shows which computation is done first by the Chi language.

□

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
2	left	$e_0(e_1, e_2, \ldots, e_n)$	Function call[page 43]
3	left	$e_1.e_2$	Projection
4	left	`dot` 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 `/\` 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 `\/` e_2, e_1 `\` 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	Implication operator
13	left	e_1 `&` e_2, e_1 `and` e_2	Conjunction operators
14	left	e_1 `or` e_2	Disjunction operator

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 $(\neg 5) < 2$, which has no meaning (the 'not' operator at level 10 is applied first)
`5 = 3 = false`	$(5 = 3) = \text{false}$	Compare the comparison result of $(5 = 3)$ with 'false' (both operators at level 11, and left binding order)

□

4.19 Addressable expressions

In the fold statement[page 64], expression folding[page 19], the assignment statement, and the receive statement[page 60] 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* (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 35] or record tuple [page 36]).

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 *VariableEx-*

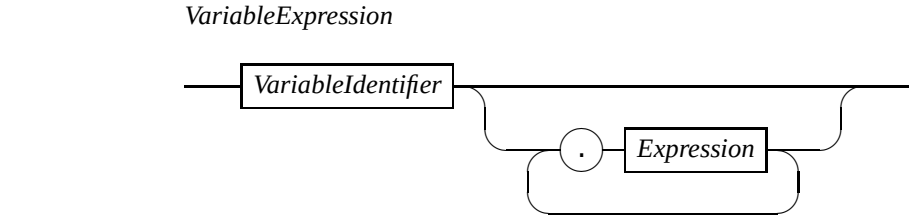

Figure 4.21: Raildiagram of *VariableExpression*.

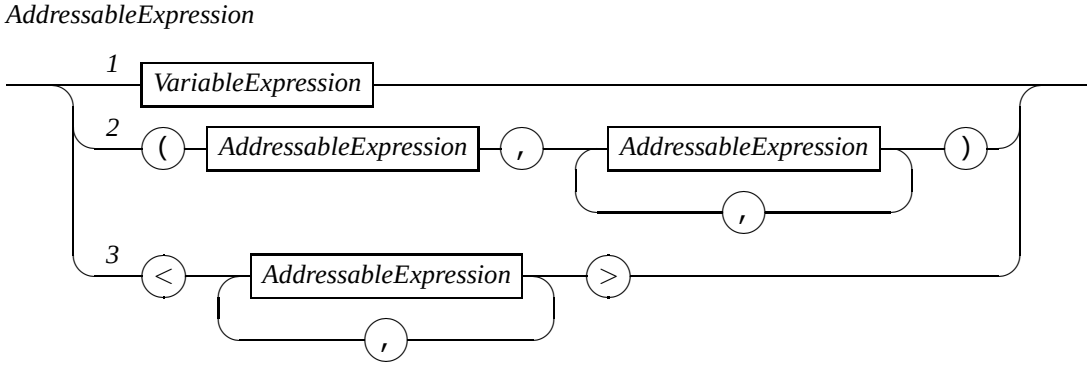

Figure 4.22: Raildiagram of *AddressableExpression*.

pression[page 46] (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.

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`

□

4.20 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.*

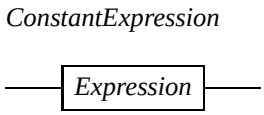

Figure 4.23: Raildiagram of *ConstantExpression*.

Chapter 5

Local declarations and definitions

Local declarations is a comma-seperated list of *LocalVariables*[page 49], *LocalChannels*[page 50], *LocalLabels*[page 51], and *ModeDefinition*[page 52] blocks. Variable declarations are explained below, channel definitions are explained in Section 5.2, declaration of labels can be found in 5.3, and mode definitions are explained in Section 5.4.

5.1 Local variables

Local variables introduce new variables inside a scope statement (Diagram *ScopeStatement*[page 62]) or inside a variable scope statement (Diagram *AdvancedScopeStatement*[page 70], Track 1).

Their syntax is defined by the *LocalVariables* diagram in Figure 5.2. It starts with the `var` keywords, followed by a comma-seperated list of variable declarations. One declaration consists of a comma-seperated list of variable identifiers, introduced with the *VariableIdentifier* block. After the identifiers comes a colon, followed by the type description. The type description consists of a dynamic part (one of the `disc`, `cont`, or `alg` keywords), and a static type part, expressed by the *Type*[page 11] block. If the dynamic type part is empty, it is defined by the type identifier in the *Type* block. Optionally, new variables can be initialized to a value by adding an equal sign and an *Expression*[page 17] block. The static type of the expression has to match with the type

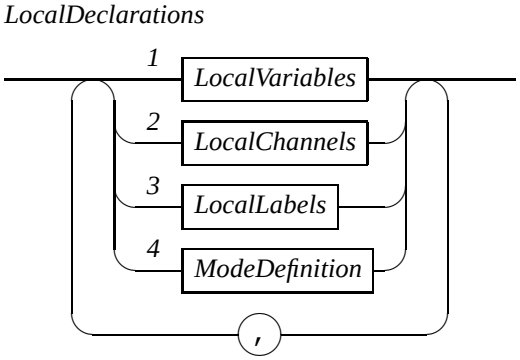

Figure 5.1: Raildiagram of *LocalDeclarations*.

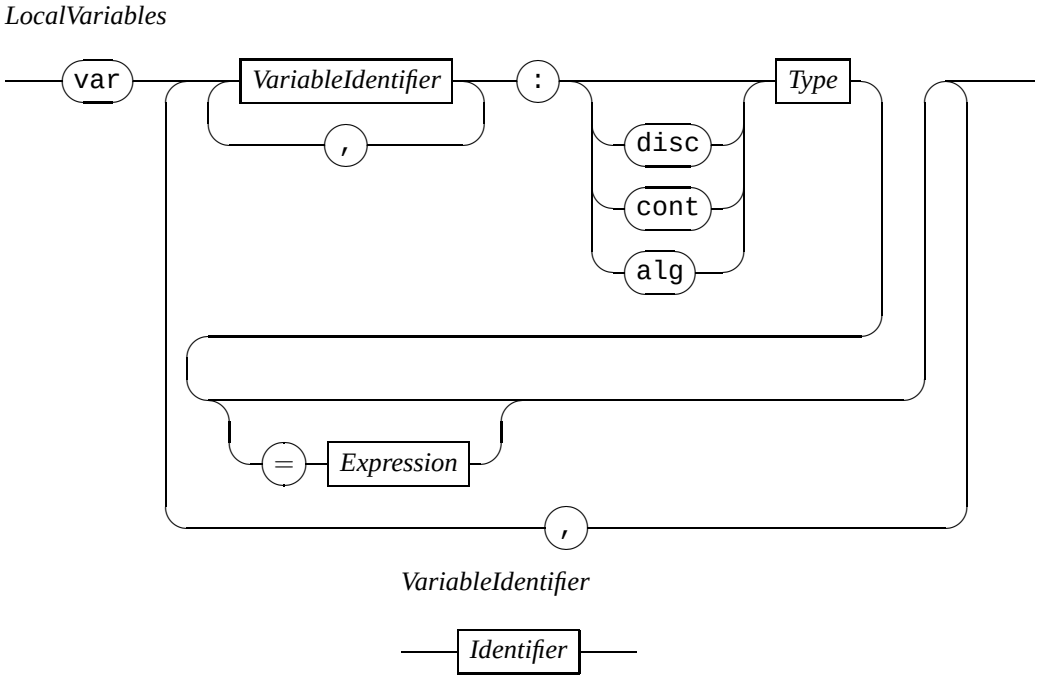

Figure 5.2: Raildiagram of *LocalVariables*.

of the list of variable identifiers. For one new variable identifier, it is equal to the static type expressed in the *Type*[page 11] block, for more identifiers, it has to be a tuple with fields of that type, and an length equal to the number of new variables in the list.

Examples

As an example of initialization, '`var x:  disc nat = 1`' creates a new discrete variable x with initial value 1, '`var x,y:  disc nat = (1, 2)`' creates new variables x and y, with initial values 1 for variable x, and initial value 2 for variable y.

□

5.2 Local channels

With the *LocalChannels* diagram in Figure 5.3, new (local) communication channels are introduced. After the `chan` keyword, a comma-seperated list of channel declarations is given. A declaration consists of a number of *ChannelIdentifier* blocks, a colon, and the structure and static type of the declared channels.

The meaning of a channel declaration is that each identifier listed is a new set of channels of the stated structure, allowing data of the stated static type to be transferred over the channel. The optional question and exclamation mark is purely for consistency with other uses of channel declarations.

The structure of the channel identifier is either a single channel (by using Track 1), or it is a bundle, (nested) vector of channels, where the size of each vector is given by the *NatExpres-*

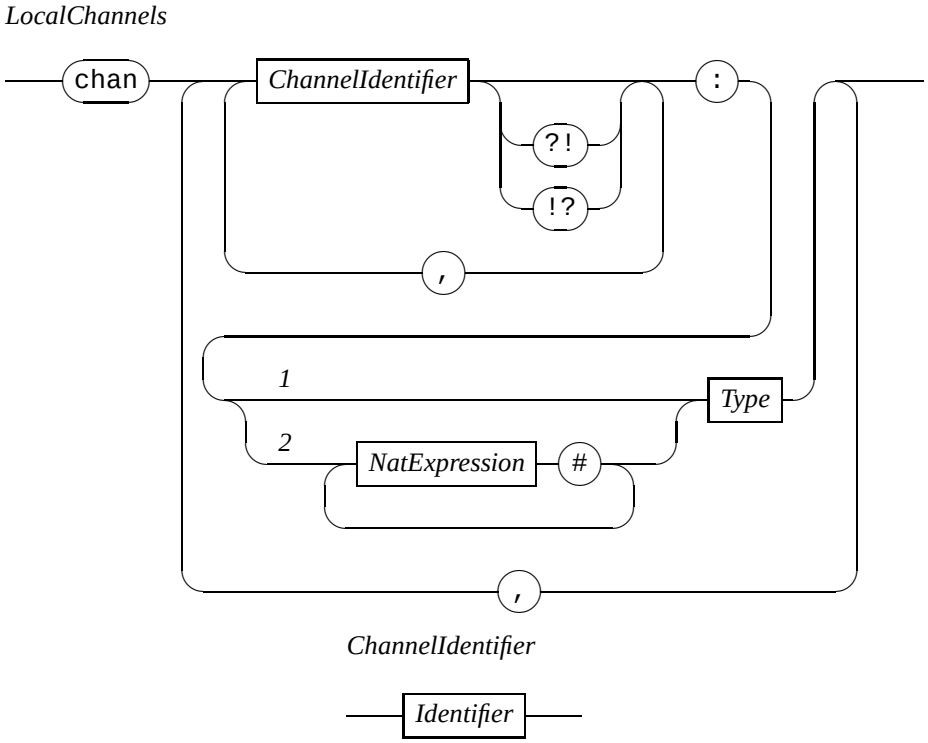

Figure 5.3: Raildiagram of *LocalChannels*.

sion[page 24] block (which must evaluate to a constant) at Track 2.

Examples

Declaration	Description
`chan c: nat`	A single channel named c that transports natural numbers.
`chan d: void`	A single channel name d that transports no data, it only synchronizes execution of send and receive statements.
`chan b: 2#real`	A bundle (vector) of two channels, $b.0$ and $b.1$, both transporting real numbers
`chan w: 2#3#{bool}`	A bundle of channels called w which is a vector of length 2 of vectors of length 3 of channels transporting sets of booleans.

□

5.3 Local labels

Label declarations introduce new labels that can be used to decorate action predicate statements, and to synchronize execution of such statements. Labels are declared using the syntax of the *LocalLabels* diagram in Figure 5.4. As you can see, a declaration of new labels starts with the `act` keyword, followed by a comma-seperated list of *LabelIdentifier*[page 56] blocks.

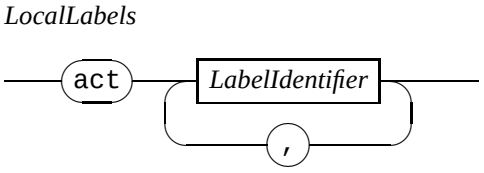

Figure 5.4: Raildiagram of *LocalLabels*.

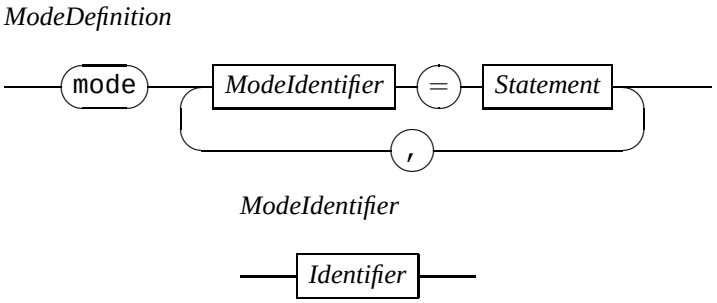

Figure 5.5: Raildiagram of *ModeDefinition*.

5.4 Mode definitions

A mode definition gives a name to a piece of code. They are used to program state machines or automata in Chi. The syntax of mode definitions is shown in Figure 5.5. The *ModeDefinition* diagram 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 as final statement (that is, tail-recursion only), 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 as final statement.

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 6

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 6.1. Just like normal expressions, you can group statements together by surrounding them with a pair of round brackets, as shown at Track 1. The commonly used statements are represented by the *BasicStatement*[page 55] 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 6.9 and 6.10. Finally, more advanced statements of the language are available using the *AdvancedStatement* block at Track 5, explained in Section 6.11.

6.1 Basic statements

The basic statements of Chi are the statements often used in modeling a system. 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 6.2 shows the kinds of basic statement available. The action statements described by the *ActionStatement*[page 56] block at Track 1 are used to emit a label. They are described in Section 6.1.1. The *AssignmentStatement*[page 58] block at Track 2 defines the statement for assigning values to variables. It is further

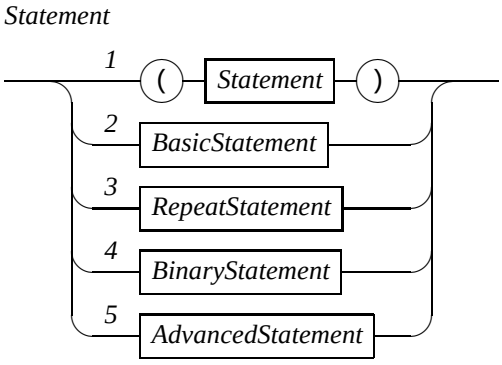

Figure 6.1: Raildiagram of *Statement*.

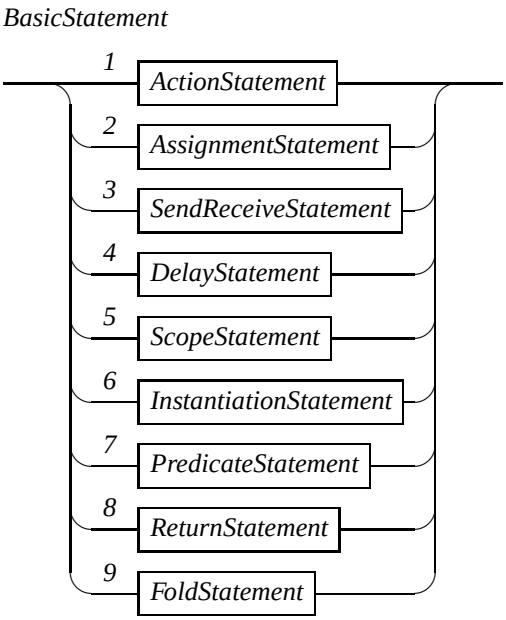

Figure 6.2: Raildiagram of *BasicStatement*.

explained in Section 6.1.3. Statements for communicating between processes are defined in the *SendReceiveStatement*[page 60] block at Track 3, further explained in Section 6.2. Waiting for some time can be specified by the delay statement described by the *DelayStatement*[page 61] block at Track 4 explained in Section 6.3. With the scope statement, new variables, channels, and mode definitions can be introduced. It is described the the *ScopeStatement*[page 62] block at Track 5 and details can be found in Section 6.4. Instantiation of mode definitions or process definitions is done with syntax described by the *Instantiation*[page 62] block used at Track 6. Explanation of the instantiation statements can be found in Section 6.5. Invariants or equations can be introduced by means of using the *PredicateStatement* block at Track 7. The statement is explained in more detail in Section 6.6. The *ReturnStatement*[page 64] block at Track 8 defines the `ret` statement, used exclusively in functions. Details can be found in Section 6.7. Finally, the *FoldStatement*[page 64] block at Track 9 explained in Section 6.8 shows how to fold several statements into one.

6.1.1 Action statement

Action statements emit a label when executed. Their syntax is shown in the *ActionStatement* diagram in Figure 6.3. Emitting a label can be done using the action statements shown at Track 1. The `skip` statement at Track 2 emits a τ. The more general form of the action statement is shown at Track 3 and consists of a label (defined by the *LabelIdentifier* diagram) and one or two tilde characters (one tilde character means that the action statement can delay, two tilde characters means it cannot). An action statement with the label `tau` emits τ when executed (due to this translation, the `skip` of the second track can also be specified as `tau ~~` by using the third track). In front of the `skip` or the general action statement a guard can be added by means of the *BooleanExpression*[page 22] block and the arrow. The guard must evaluate to true before

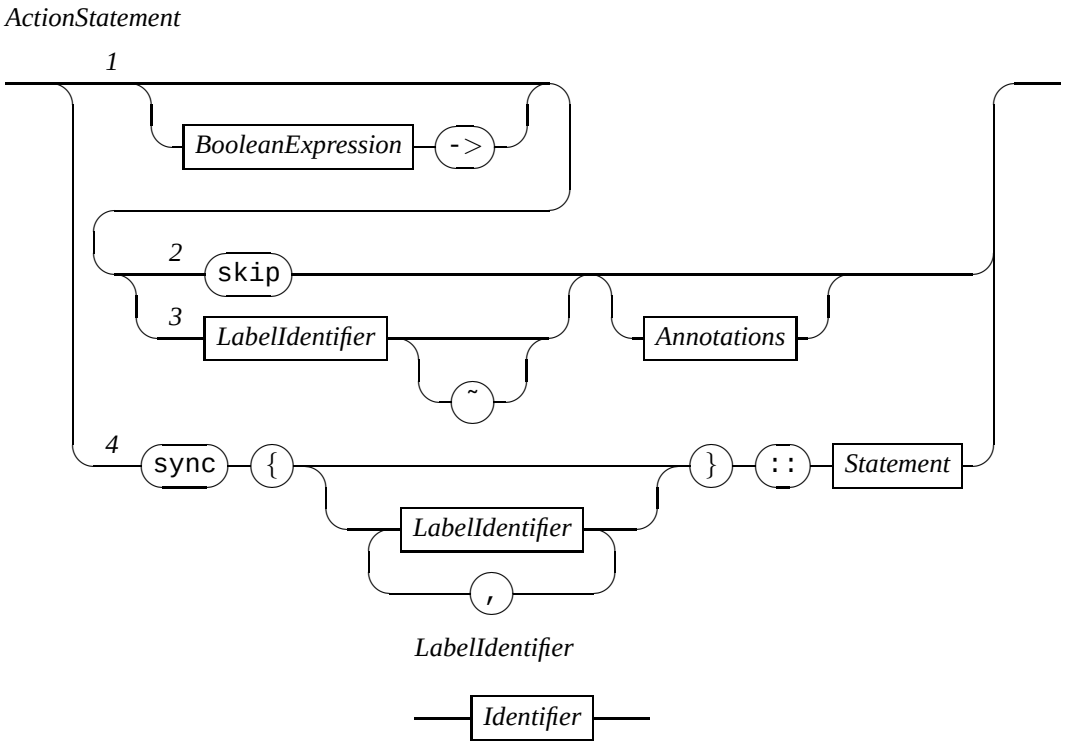

Figure 6.3: Raildiagram of *ActionStatement*.

the statement can be executed. Finally, after the action statement has executed, additional side effects can be performed by using the *Annotations*[page 57] block.

With Track 4, a set of synchronizing labels can be defined for a statement. An action statement which is part of the latter statement with a label listed in the set after the `sync` keyword can only be executed together with other action statements that also synchronize on the same label. The set of labels may not include `tau`.

6.1.2 Annotations

Annotations are used to attach side effects to a statement without changing the meaning of the model (the latter may happen when statements are added). There are two kinds of side effects available in Chi, namely generation of output to `stdout` (such as the terminal), and termination of the simulation. The syntax of the annotations is shown in the *Annotations* diagram in Figure 6.4. The annotation with Track 1 in it causes the value of the expressions to be output to `stdout` when the statement it is attached to has been executed. The annotation with Track 2 in it causes the execution to be terminated after the execution of the statement it is attached to is done. A few examples to illustrate the use of annotations are shown below.

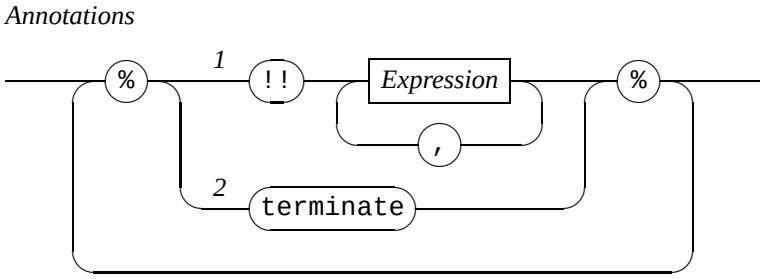

Figure 6.4: Raildiagram of *Annotations*.

Examples

Statement	Description
`c?x % !! x %`	After receiving a value for `x` over channel `c`, the received value is printed to the terminal
`delay 10 % !! "bye" % % terminate %`	After 10 time units, the delay statement ends. It outputs 'bye', and the simulation (as a whole) terminates

□

6.1.3 Assignment statement

Assignment statements assign values to variables. The most commonly used form of the assignment statement is shown in the *AssignmentStatement* diagram in Figure 6.5, the other assignment statement, the action predicate statement, is shown in Section 6.11 in the *AdvancedActionStatement*. An assignment statement consists of three parts. The first optional part is a guard, expressed by the *BooleanExpression*[page 22] block and the arrow symbol. The second part is also optional and consists of a *LabelIdentifier*[page 56] block followed by zero or one tilde characters and a colon. The third and last part is obligatory, and consists of the actual assignment. It takes the form of a 'normal' assignment statement (expressed with a *Assignment* block), or an action predicate statement (expressed with the *ActionPredicate*[page 59] block, explained in Section 6.1.4), optionally concluded with annotations.

The meaning of the assignment statement is that values expressed by the list of *Expression*[page 17] blocks in the *Assignment* diagram are assigned to the variables expressed by the list *AddressableExpression*[page 46] blocks in the same diagram. If the guard part is not empty, the assignment can only take place when the value of the boolean expression of the *BooleanExpression*[page 22] block is true. The result of the execution of an assignment statement is a transition in the program state labeled with τ. By using the second part of the statement, you can change the label being emitted in the transition. Also in the second part, you can can specify the delay behavior of the statement. One tilde character means that the statement is delayable, two tilde characters means that the statement cannot delay. If you only want to specify the delay behavior of the statement without changing the label of the transition, use label `tau`. The system will replace that label with the τ label.

Finally, after the assignments have been performed, the annotations are executed if the *Annotations*[page 57] block is used in the statement.

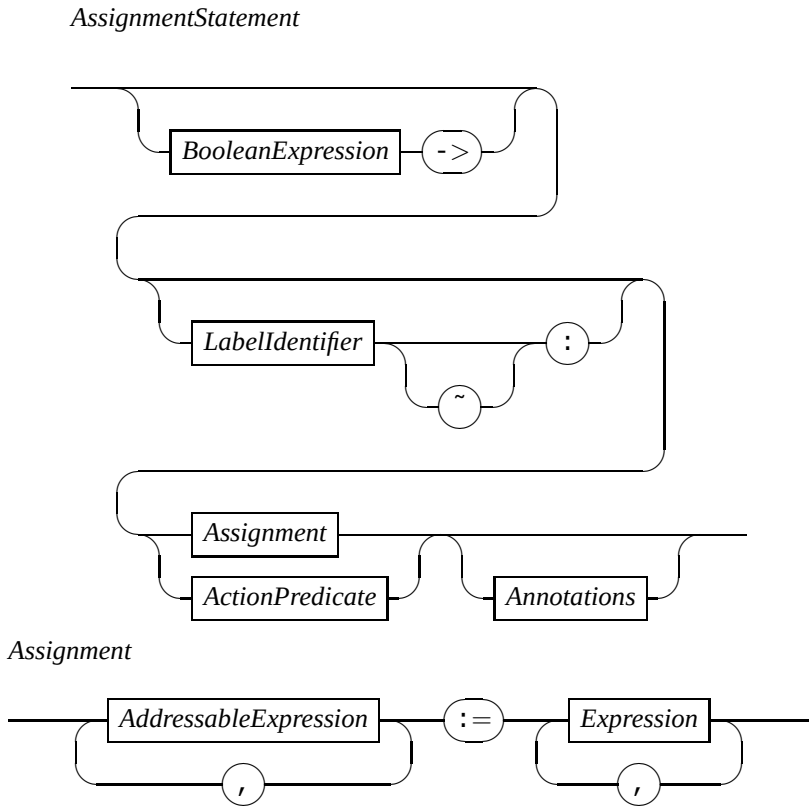

Figure 6.5: Raildiagram of *AssignmentStatement*.

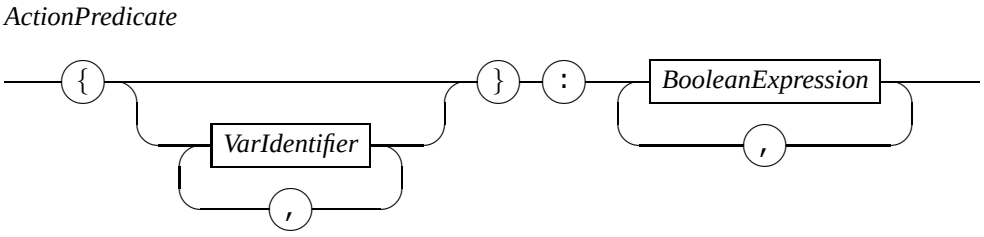

Figure 6.6: Raildiagram of *ActionPredicate*.

6.1.4 Action predicate

The action predicate defined in the *ActionPredicate* diagram in Figure 6.6 is a generalisation of an assignment. It consists of two parts, a set of variable identifiers, and a boolean expression. The semantics of the action predicate is that a value for the variables has to be found such that the boolean expression holds. The value of the variables listed in the set may be changed to find a solution.

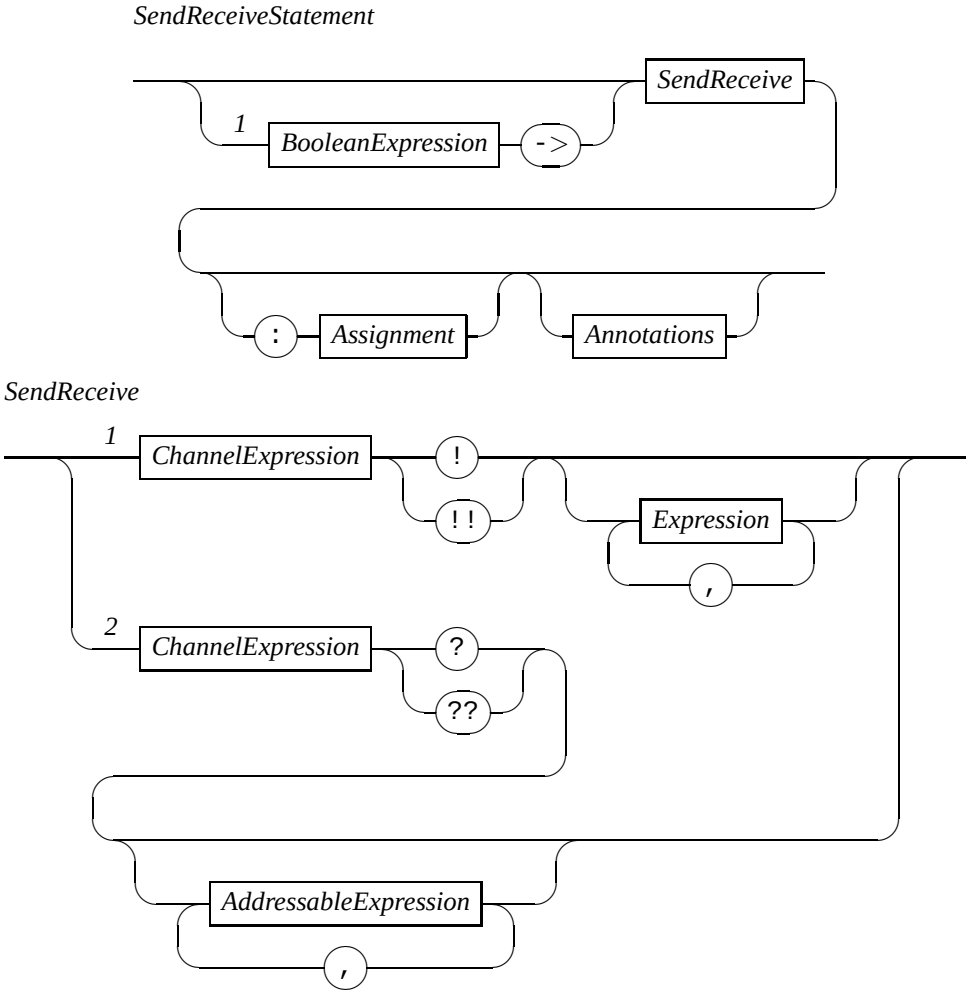

Figure 6.7: Raildiagram of *SendReceiveStatement*.

6.2 Communication statements

With communication statements, you can exchange information between two different processes across a channel. The syntax of the send and receive statement are shown in the *SendReceiveStatement* diagram in Figure 6.7. 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. Exclamation marks (at Track 1 of then *SendReceive* diagram) are used for expressing that information should be sent, question marks (at Track 2 of the same diagram) means receipt of information.

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.

Due to the synchronous communication between the sender and the receiver, it may happen

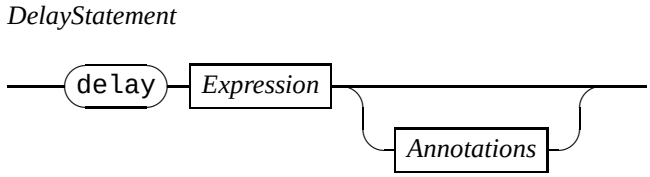

Figure 6.8: Raildiagram of *DelayStatement*.

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 statement and with double question marks for the receiving statement. The expressions of the *ChannelExpression* blocks must evaluate to a channel. The data exchanged in the communication specified by the *Expression*[page 17] and *AddressableExpression*[page 46] blocks must match the data part of the channel declaration. When the data part of the channel declaration is `void`, it means that no actual data is transferred, only the send and the receive statements are synchronized (executed together in a single step). In that case, the expressions of the send and the receive statements should be omitted.

A send or a receive statement can be performed conditionally by placing a guard in front by means of the *BooleanExpression*[page 22] block and the arrow. The send or receive can only take place when the guard hods. Finally, after the communication has taken place, an assignment may be performed to update the internal state. At the receiving side, assignments to variables just received are generally not recommended. They are allowed, but the same value must be assigned in order for the receive statement to be executed.

Finally, side effects can be added after execution of the statement by means of the *Annotations*[page 57] block.

6.3 Delay statement

The *delay statement* is used to state that the process should wait a while before proceeding. Typically, this is used to simulate some internal behavior that is not modeled further and takes time to perform. An example is a manufacturing step in a machine.

The delay statement is defined by the *DelayStatement* diagram in Figure 6.8. It consists of the keyword `delay`, followed by an expression that defines how long to wait in time-units. The expression has to evaluate to a non-negative numeric value, and is computed just before the first delay step. After a delay step has been taken, the expression has no influence of the length of the delay any more.

After the statement terminates with a τ action, the side effects of the annotations are performed if specified.

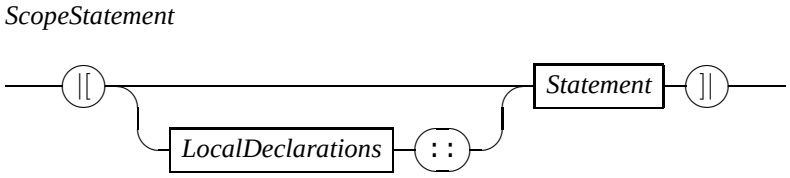

Figure 6.9: Raildiagram of *ScopeStatement*.

Examples

```
model M() =
|[ var x: cont real = 2.0
:: ( dot 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.

□

6.4 Scope statement

With the scope statement, you can introduce new local variables, channels, and mode definitions. The definition of the scope statement is shown in the *ScopeStatement* diagram in Figure 6.9. The syntax of the new variables, channels, labels, and mode definitions is described by the *LocalDeclarations* block, and the code that should be executed is available through the *Statement*[pages 55, 81] block. As you can see, the *LocalDeclarations* block is optional. This is done to allow for writing an additional scope around statements (in much the same way as you can write brackets around statements).

The scope statement is a syntactical extension. It is translated to a nested variable scope, channel scope, action scope, and mode scope statement, explained in Section 6.11.

6.5 Instantiation statements

In Chi you can define processes and modes[page 52]. The syntax for instantiating them is shown in the *Instantiation* diagram in Figure 6.10. At Track 1, a process definition or declaration is instantiated. The *ProcessIdentifier*[page 81] must match the name of the process definition or declaration. If a *ModeIdentifier*[page 52] is also specified, the process definition or declaration must exist in the named module. Otherwise, it must exist in the current module. The list of *TemplateValue*[page 44] blocks must match with the explicit template definitions of the used process definition or declaration. Omssision of this list means that the used definition or declaration should have zero explicit template definitions. The list of expressions (the *Expression*[page 17]

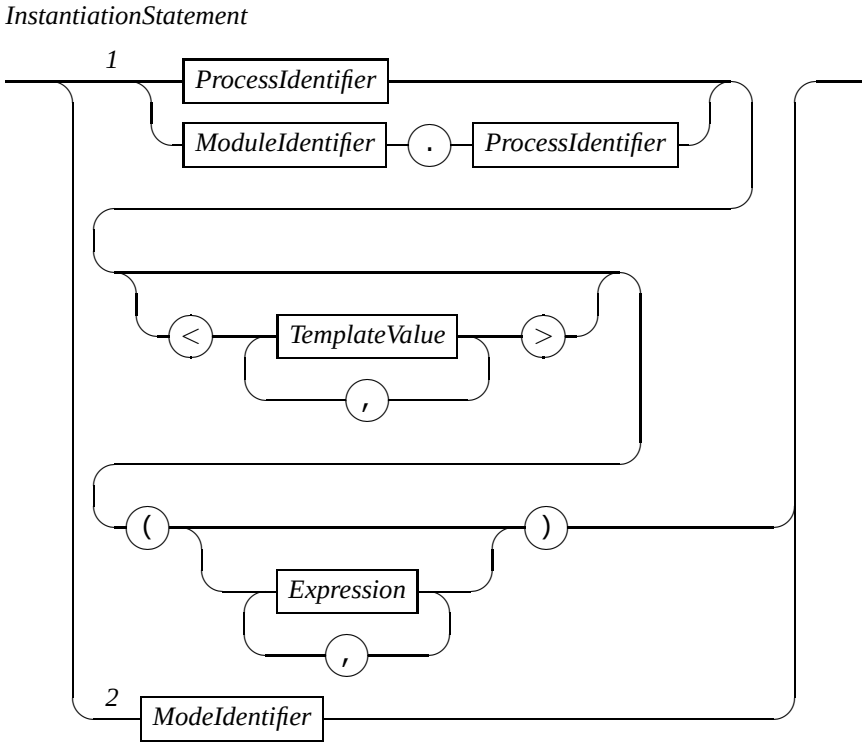

Figure 6.10: Raildiagram of *Instantiation*.

blocks) are the actual arguments of the instantiated definition. The number of arguments must match the number of formal parameters, also their dynamic and static type parts 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 dynamic type part of the formal parameter. Reference parameters (parameters with dynamic type part `cont`, `alg`, `chan`, or `disc`) refer to the variable stated in the actual expression, value parameters[page 84] (parameters with dynamic type part `val`) are read-only in the instantiated proces.

At Track 2, a mode definition is instantiated.[page 52] The *Identifier*[page 8] block must match with the identifier of the mode definition being instantiated. Since mode definitions have no parameters, there is no expression list.

6.6 Predicate statement

At the heart of continuous-time systems and combined discrete-event and continuous-time systems are the equations, which describe the relation between discrete, continuous, and algebraic variables. In Chi, equations are available through the predicate statement, described by the *PredicateStatement* diagram in Figure 6.11. The statement starts with the keyword `inv` (invariant), followed by a boolean condition. In Chi, the condition always holds, up to the point that a statement is not executed if their execution would imply breaking an invariant condition.

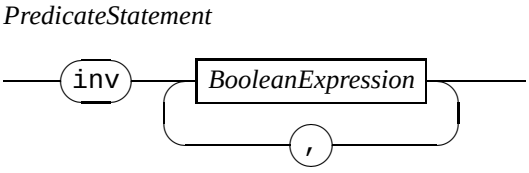

Figure 6.11: Raildiagram of *PredicateStatement*.

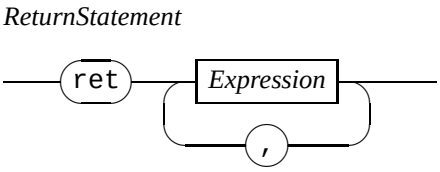

Figure 6.12: Raildiagram of *ReturnStatement*.

Examples

```
model M() =
|[ var x: cont real = 2
:: inv dot x = cos(time)
]|
```

After initializing the continuous variable `x` to the value 2.0, a predicate statement defines the trajectory for x' to be $\cos$ time which results in a trajectory of $2 + \sin$ time for variable x.
□

6.7 Return statement

The return statement shown in the *ReturnStatement* diagram in Figure 6.12 is used in the statement part of function definitions[page 79]. 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.

6.8 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 integral numeric 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 6.13. It consists of three arguments between a pair of round brackets. The first argument is the binary statement operator[page 66] that is to be used *between* different instantiations of the statement,

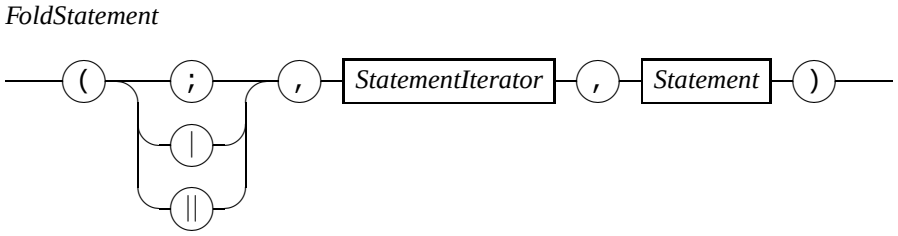

Figure 6.13: Raildiagram of *FoldStatement*.

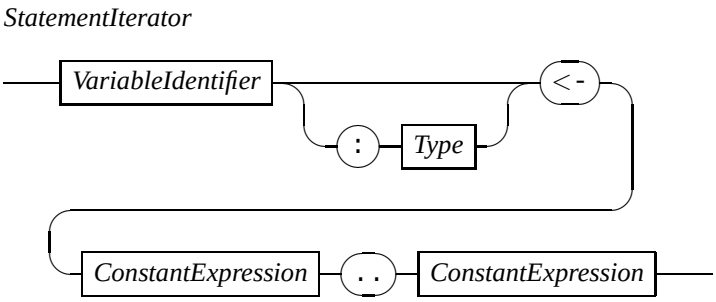

Figure 6.14: Raildiagram of *StatIterator*.

the second argument is the *StatIterator*[page 65] 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*[pages 55, 81] 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`.

□

medskip The *StatIterator* diagram in Figure 6.14 defines the range used for unfolding the fold statement[page 64]. 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.*

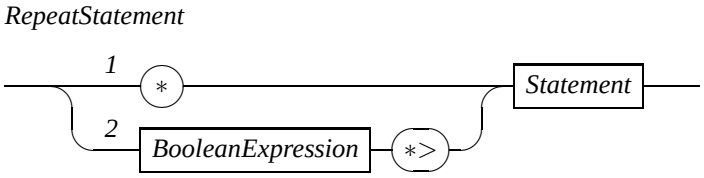

Figure 6.15: Raildiagram of *RepeatStatement*.

6.9 Repeat statements

There are two unary statement operators, shown in the *RepeatStatement* diagram in Figure 6.15. Both are used to define a repetition. (In Section 6.11 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 *Expression*[page 17] 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.

□

6.10 Binary statements

There are three binary operators on statements, as shown in Diagram *BinaryStatement* at Figure 6.16. 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.

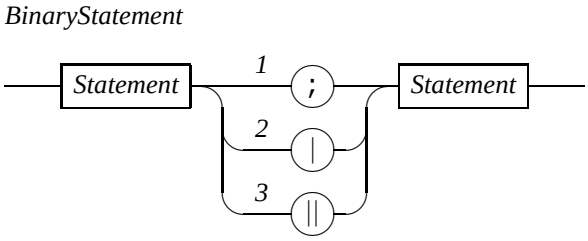

Figure 6.16: Raildiagram of *BinaryStatement*.

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 61] 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`.
□

6.11 Advanced statements

Most of the advanced statements are derived directly from statements in the formal semantics and have little practical value for modelers. The statements are shown in Figure 6.17. At Track 1 you can see the `tcp` (time can progress) statement. It begins with a `tcp` keyword followed by a condition in the form of a boolean expression. The meaning of the statement is that delays can occur (time can progress) only if the condition holds. Track 2 shows the urgent action statement. It takes a set of channel identifiers and a set of label identifiers. If the *Statement*[pages 55, 81] block allows a communication over a channel listed in the set channel identifiers, a delay step is not possible. In the same way, actions that can take place with a label identifier from the second set also prevent occurence of a delay step. Track 3 encapsulates channels and actions. It blocks seperate send and receive steps for channels in the first (channel identifier) set, enforcing that only communication can take place on the listed channels. Also, it prevents actions from occurring if their label is listed in the second set. At Track 4, the initialization statement is shown. In the *BooleanExpression*[page 22] an initialization predicate is defined. The *Statement*[pages 55, 81] following the double bigger-than signs can only be executed when a value for the variables can be found such that the initialization predicate holds. At Track 5 the *AdvancedSendReceiveStatement*[page 68] is introduced. You can find an explaination of the statement in Section 6.11.1. At Track 6, several scope statements are added to the language via the *AdvancedScopeStatement*[page 70]. These scopes are further explained in Section 6.11.2. Finally, at Track 7 you can find the deadlock statement δ, and at Track 8 the inconsistent state is introduced.

6.11.1 Advanced send and receive statements

There are two track in the *AdvancedSendReceiveStatement* diagram in Figure 6.18. The first track is a normal send/receive statement, except that the communication is followed by an action predicate instead of an optional assignment. At the second track you can see the combined send-receive statement. Its syntax is the same as a normal send or receive statement, except that in the *SendReceiveAssignment* diagram in Figure 6.19 there is a combined send and receive token `!?` after the *ChannelExpression* diagram. In addition, it is followed by an assignment. The statement is normally never entered directly, it is intended to act as a replacement for a combined send and receive statement that may appear as a result of algebraic rewriting of a specification.

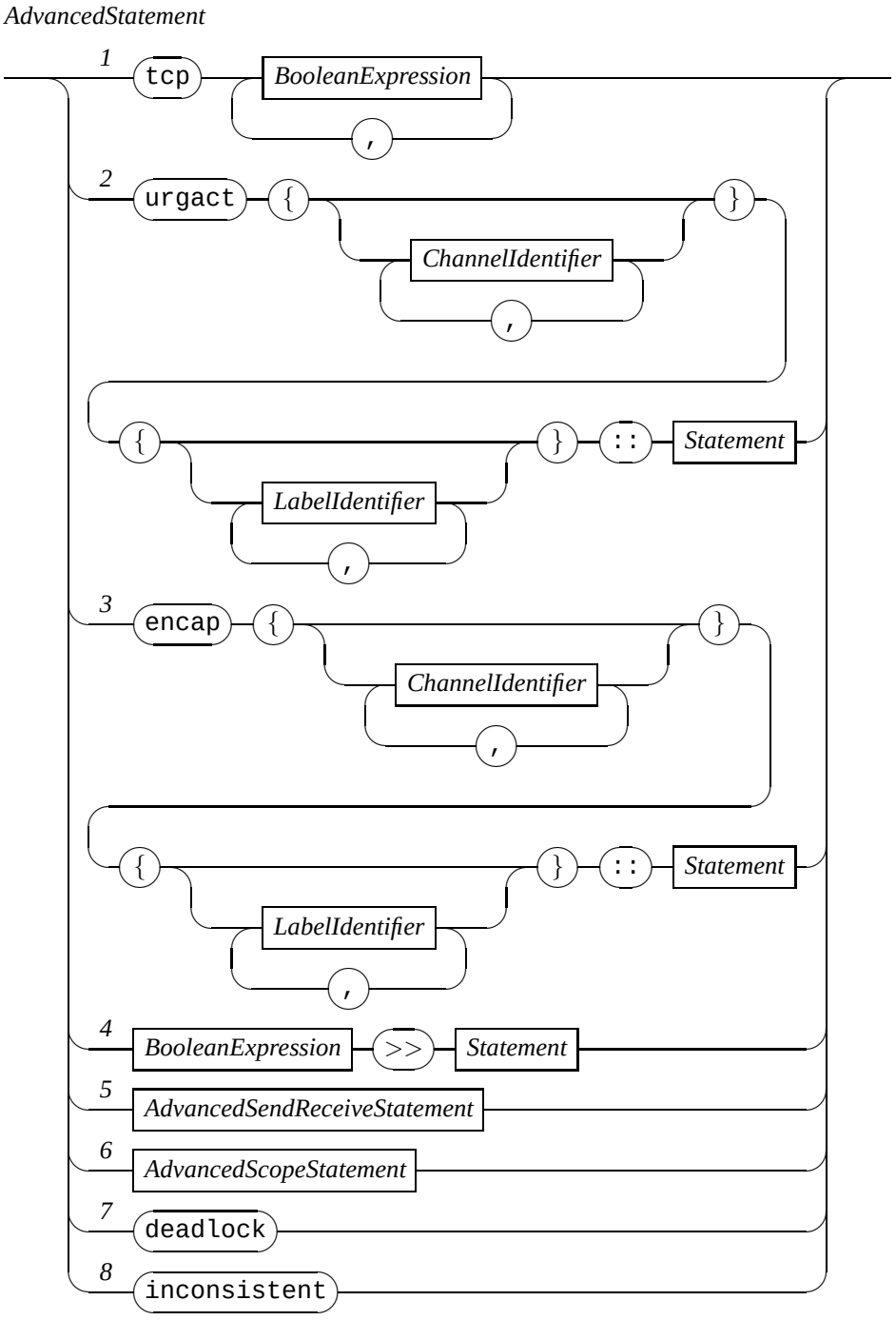

Figure 6.17: Raildiagram of *AdvancedStatement*.

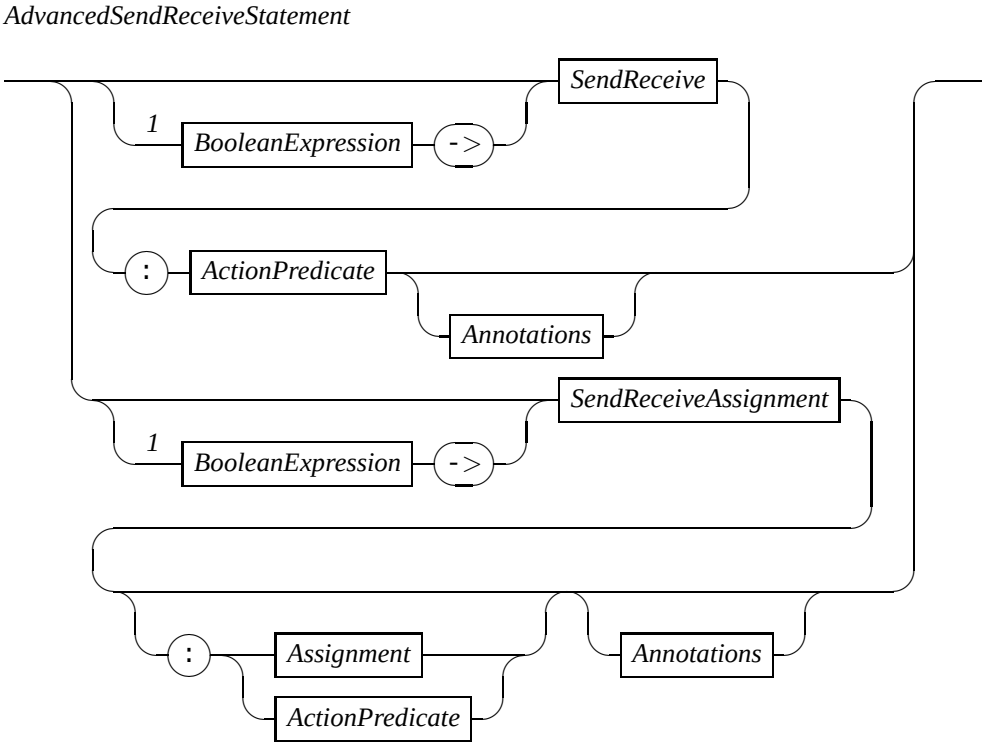

Figure 6.18: Raildiagram of *AdvancedSendReceiveStatement*.

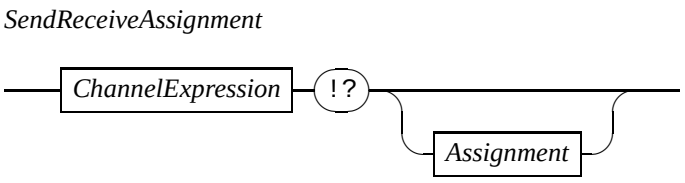

Figure 6.19: Raildiagram of *SendReceiveAssignment*.

6.11.2 Advanced scope statements

The modeling scope statement defined in the *ScopeStatement*[page 62] block is a syntactic extension. It is rewritten to three nested primitive scope statements shown in the *AdvancedScopeStatement* in figure 6.20. At Track 1 you can see the variable scope statement. Its purpose is to make new variables available in the *Statement*[pages 55, 81] block. At Track 2, the channel scope statement does the same, except with new channels. New labels are introduced with an action scope statement, as shown at Track 3. Finally, at Track 4 the mode definition scope statement creates new mode variables to be used in its *Statement*[pages 55, 81] block.

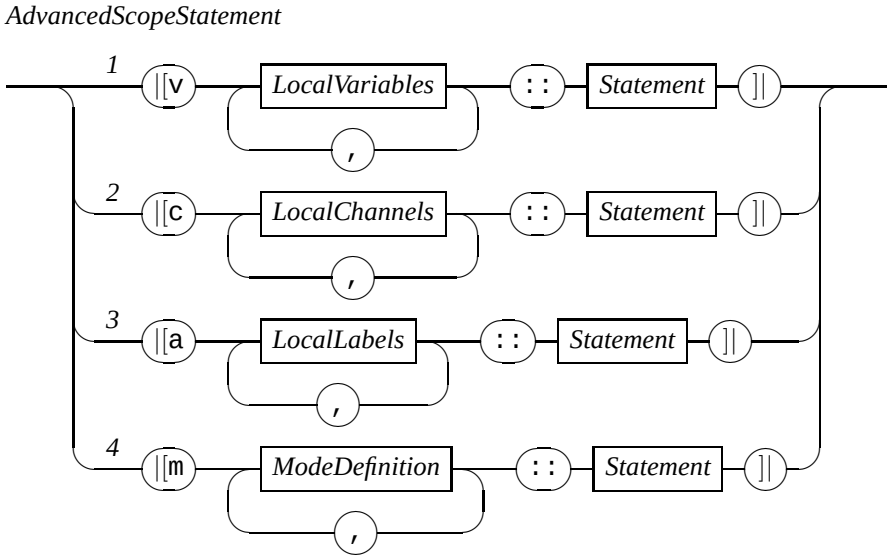

Figure 6.20: Raildiagram of *AdvancedScopeStatement*.

6.12 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. *Repeat statement operators*: Operators of the loop, and the while statement (shown in the *RepeatStatement*[page 66] 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 (repeat) operators. It is shown in the *BinaryStatement*[page 66] 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 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 composition operator*: This operator has lower priority than the sequential composition operator, so the alternative composition operator can be used to make a choice between two sequences of statements without using brackets.

4. *Parallel composition operator*: The parallel composition operator has the lowest priority.

Chapter 7

Global declarations and definitions

A Chi module is defined as a (non-empty) sequence of global definitions and declarations as shown in the *ChiModule* diagram in Figure 7.1. Enumeration definitions are further explained in Section 7.1, and more details about constant definitions are in Section 7.2. What forms of type definitions exist is shown in Section 7.3 and how to import other Chi modules in explained in Section 7.4. You can find the details of how to create a function in Section 7.5, while the contents of processes are explained in Section 7.6. Finally, details of models are shown in Section 7.7.

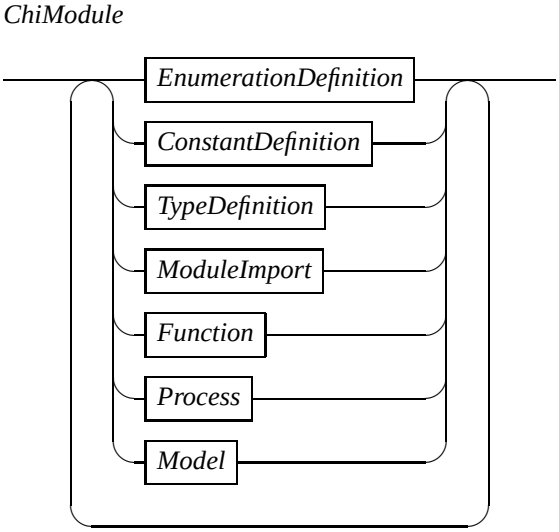

Figure 7.1: Raildiagram of *ChiModule*.

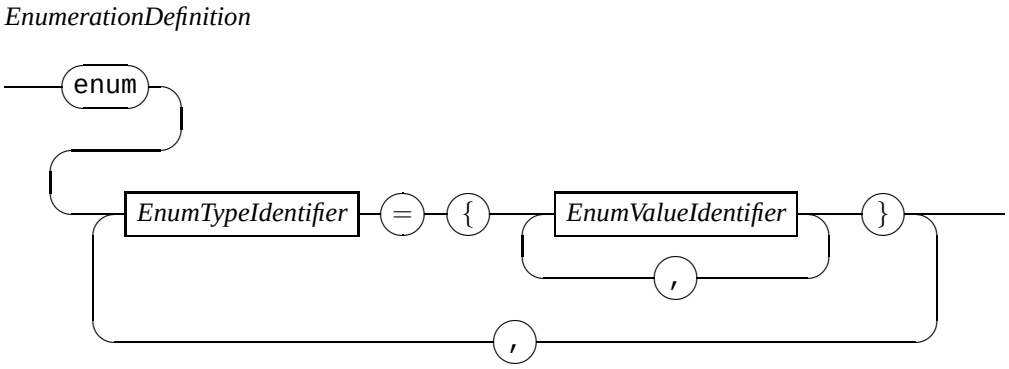

Figure 7.2: Raildiagram of *EnumerationDefinition*.

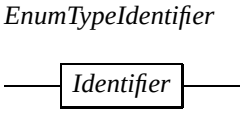

Figure 7.3: Raildiagram of *EnumTypeIdentifier*.

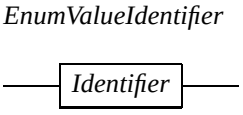

Figure 7.4: Raildiagram of *EnumValueIdentifier*.

7.1 Enumeration definition

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 defined in Figure 7.3. Between the curly brackets, the set of enumeration values of the enumeration type is listed as a sequence of *EnumValueIdentifier* blocks. The latter block is defined in Figure 7.4. Each enumeration value must be unique in its enumeration type.

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.*

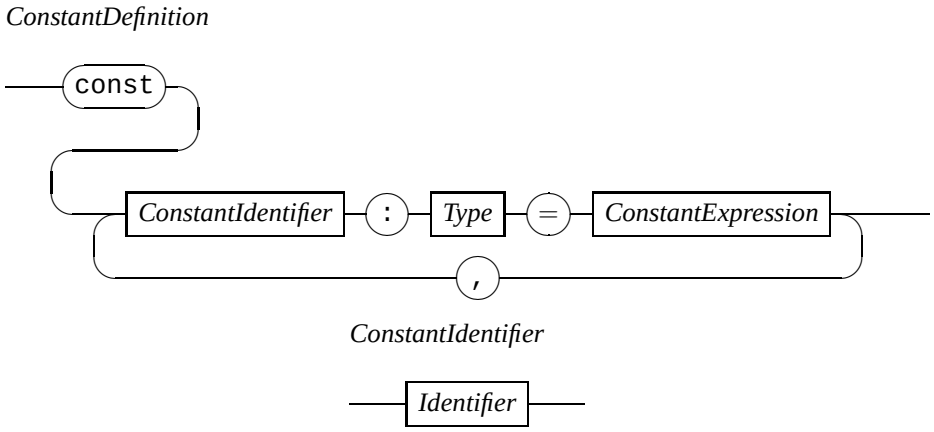

Figure 7.5: Raildiagram of *ConstantDefinition*.

7.2 Constant definition

Constant definitions are used to give names to constant values. The syntax is shown in Figure 7.5. 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 47] 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.*

7.3 Type definition

Type definitions are used to introduce names for types, making the Chi specification more readable. The syntax of a type definition is shown in Figure 7.6. The name of the new type is

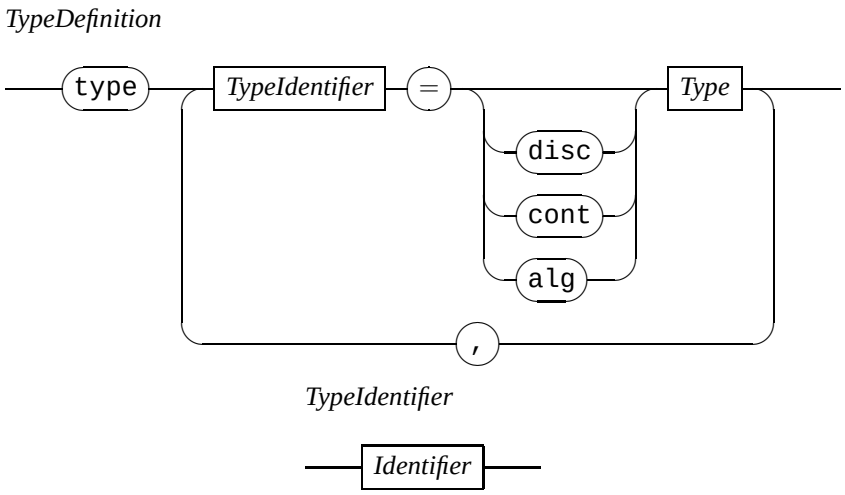

Figure 7.6: Raildiagram of *TypeDefinition*.

defined by the *TypeIdentifier* block, an *Identifier*[page 8] that refers to a type. The associated type consists either of a combined dynamic and static type part (by prefixing the static part with one of the keywords `disc`, `cont`, or `alg`), or a static type only. It is allowed to use names of type definitions in the static type part of another type definition only if the type associated with the name has no dynamic part.

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

Examples

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

The type definition attaches the static type `nat` to the type identifier 'lot'. The type identifier 'batch' has dynamic type `disc` and static type `[nat]`.

□

Examples

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

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

□

7.4 Module import

Module imports or import statements allow access to definitions written in other files. Its syntax is shown in Figure 7.7. Import has two forms. At Track 1, only the module is imported, meaning its name becomes available for use. At Track 2, both the module and its definitions are imported.

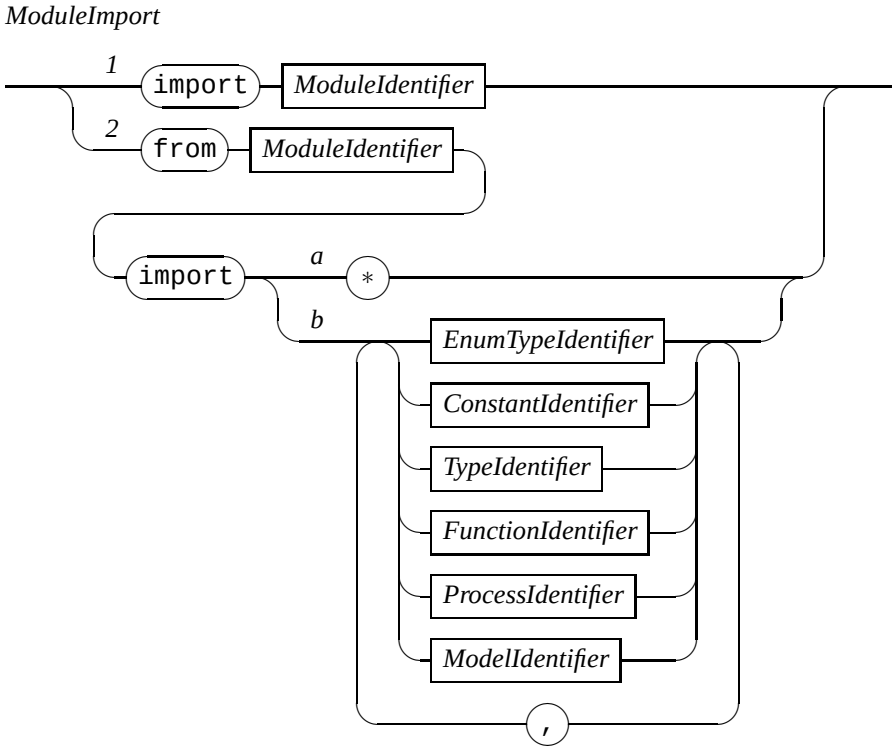

Figure 7.7: Raildiagram of *ModuleImport*.

In the latter case you can 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 name.

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.

Function

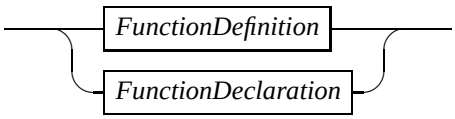

Figure 7.8: Raildiagram of *Function*.

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 to its definitions 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

The syntax of a function is shown in the *Function* diagram in Figure 7.8. It is either a function definition[page 79] or a function declaration[page 80]. The former gives the type signature[page 79] of the function and also the algorithm to compute the result in Chi. A function declaration on the other hand only give the type signature. The algorithm is specified elsewhere, possibly in another language.

With the type signature of a function, a connection is made between the name of the function and its computation. Once this connection is made, the computation can be performed 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 43] or the more common phrase *function call*.[page 43] In Chi, the computations defined by a function behave as proper mathematical functions, that is, they are not influenced from the outside, cost no time, and do not have side effects.

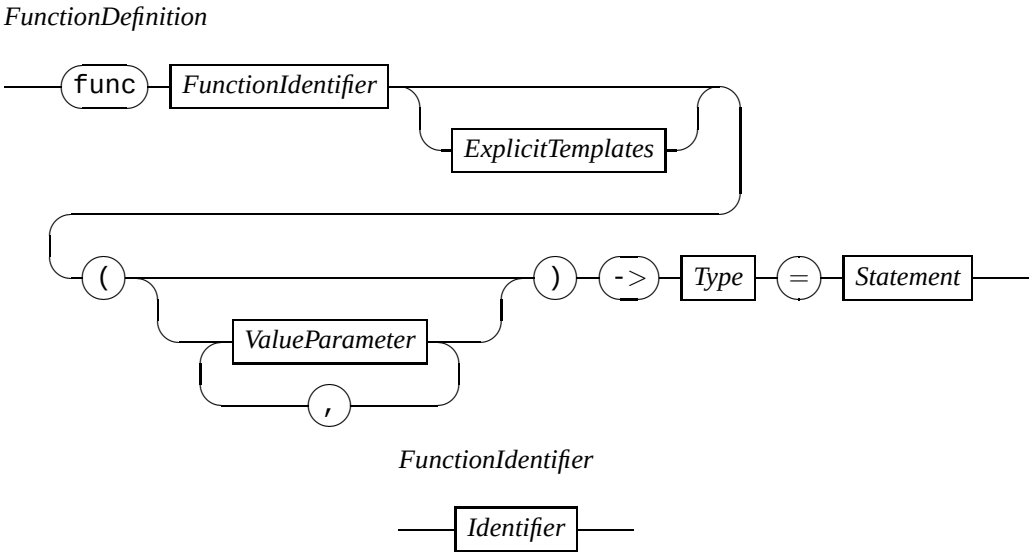

Figure 7.9: Raildiagram of *FunctionDefinition*.

7.5.1 Function definition

A function definition is the usual way of defining a computation in Chi. Its syntax is shown in the *FunctionDefinition* diagram in Figure 7.9. The *FunctionIdentifier* block defines the name (which is the same as an *Identifier*[page 8]) of the computation. The optional *ExplicitTemplates*[page 87] block defines the statically decided function arguments (explained in more detail in Section 7.14), the declarations in the *ValueParameter*[page 84] blocks 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 combination of the identifier, the templates, the value parameters, and the return type is called the type signature of the function.

The *Statement*[pages 55, 81] block 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 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 64] to deliver the computed value to its caller.

Examples

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

□

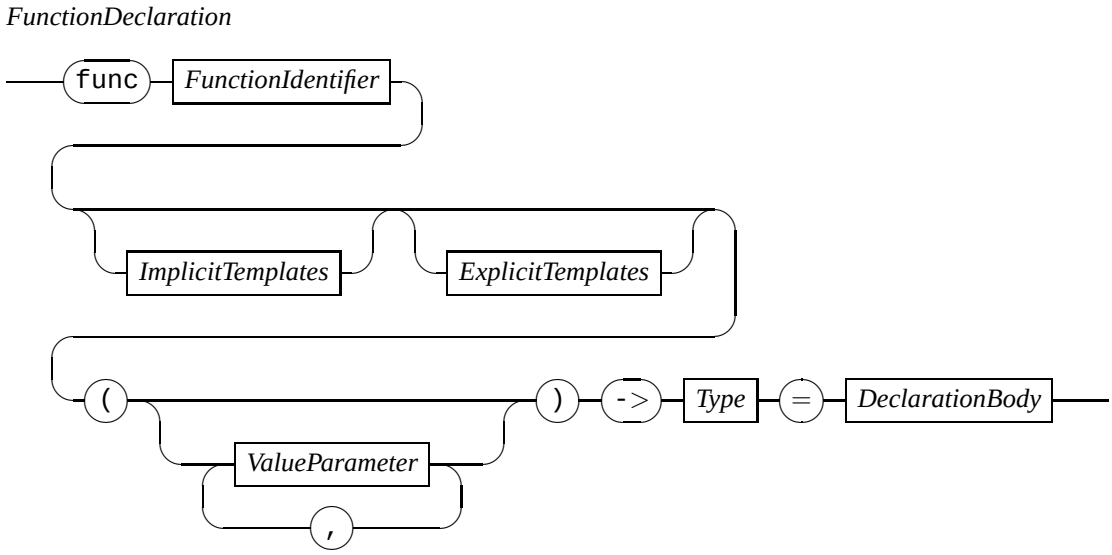

Figure 7.10: Raildiagram of *FunctionDeclaration*.

7.5.2 Function declaration

A *FunctionDeclaration* block shown in Figure 7.10 is used to declare a function (that is, state the type signature of the function and giving the location of its implementation). Like the function definition above, a function declaration describes a computation without side effects. The *FunctionIdentifier*[page 79] states the name of the function being declared. The optional *ImplicitTemplates*[page 87] and *ExplicitTemplates*[page 87] are the statically decided (during the static semantics phase) parameters of the function. Both forms of template definitions are explained in more detail in Section 7.14.

The *ValueParameter*[page 84] blocks are discrete call-by-value arguments filled in at the time of the function application (that is, their value is computed 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. The syntax of a process is shown in the *Process* diagram in Figure 7.11. As you can see, a process is either a

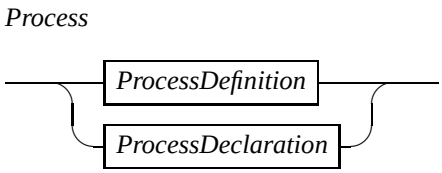

Figure 7.11: Raildiagram of *Process*.

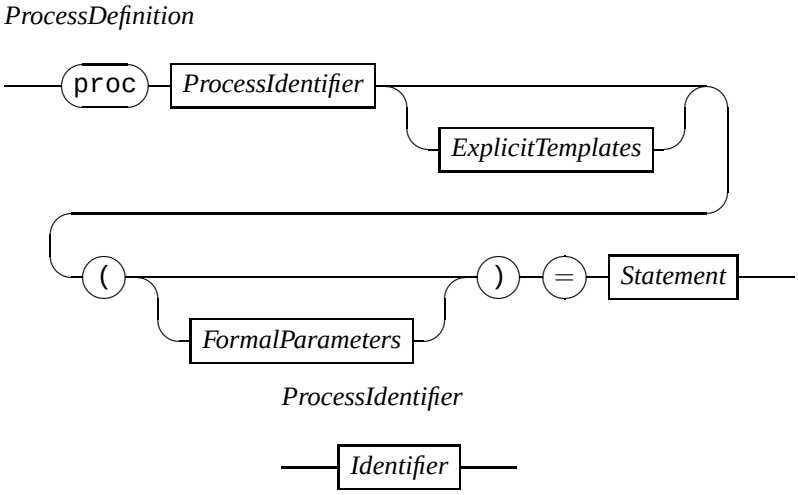

Figure 7.12: Raildiagram of *ProcessDefinition*.

process definition (by means of the *ProcessDefinition*[page 81] block) or it is a process declaration (by using the *ProcessDeclaration*[page 82] block). A *process definition* has an explicit description of its behavior (in its statement). A *process declaration* has no explicit behavioral description. Instead it refers to an external description. The process definition is used to describe behavior in the Chi language, the process declaration is used to state that some behavior exists with a certain name elsewhere.

The sections below explain the process definition and the process declaration syntax. Instantiation of a process is commonly known as *process instantiation*,[page 62] and is explained in Section 6.5.

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 7.12 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 87] and the *FormalParameters*[page 83] are used to parameterize the behavior. The template parameters are constructed statically during static semantics checking and are explained in Section 7.14, the formal parameters are exchanged during process instantiation and are explained in Section 7.8. The behavior of the process is described explicitly in the *Statement* block.

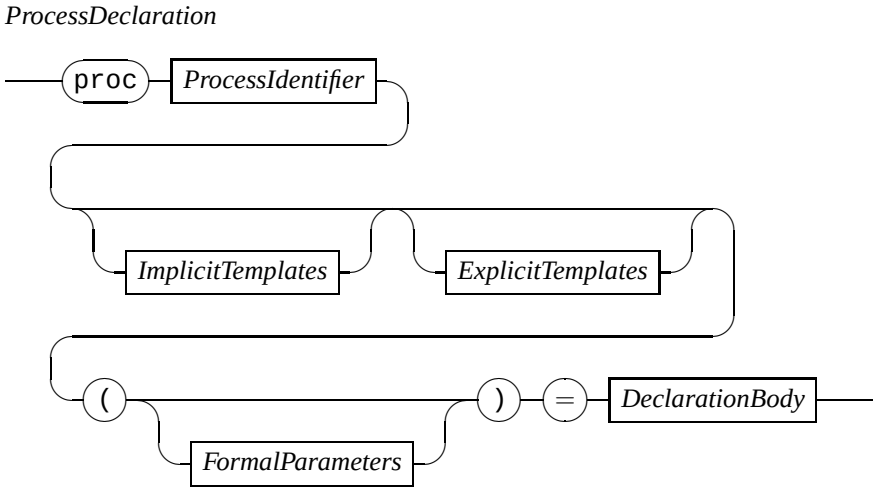

Figure 7.13: Raildiagram of *ProcessDeclaration*.

7.6.2 Process declaration

A process declaration is very similar to a process definition, except that there is no body. It just states the type signature of the process and states where the contents of the process body is defined. The syntax of a process declaration is shown in the *ProcessDeclaration* diagram in Figure 7.13. Like the process definition, the *ProcessIdentifier*[page 81] block states the name of the process being declared. The optional *ImplicitTemplates*[page 87] and *ExplicitTemplates*[page 87] are the statically decided (during the static semantics phase) parameters of the process and are explained in Section 7.14. The parameters in the *FormalParameters*[page 83] are (as in the process definition) arguments filled in at the time of process instantiation. Their syntax is explained in Section 7.8. The implementation of the process is specified in the final *DeclarationBody*[page 86] block. The contents of this block is described in Section 7.13.

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

7.7 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 *Model* diagram in Figure 7.14. 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 *FormalParameters*[page 83] block. If the block is used, the values of the formal parameters are filled at startup of the experiment.

The actual behavior of a model can be defined in two ways. Track 1 shows the most commonly used way of specifying a model, namely as a cooked model. With such a model you get all the 'normal' behavior of Chi such as an initialized continuous (read-only) variable `time`, actions take precedence above delaying, communications are not delayable, and seperate send and receives are not possible.

In the case you want to specify such behavior yourself, you can use Track 2, a 'raw' model.

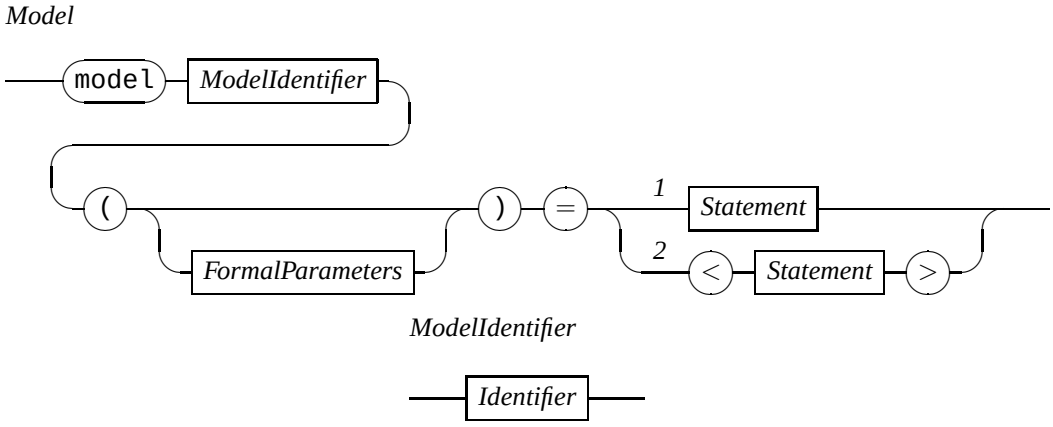

Figure 7.14: Raildiagram of *Model*.

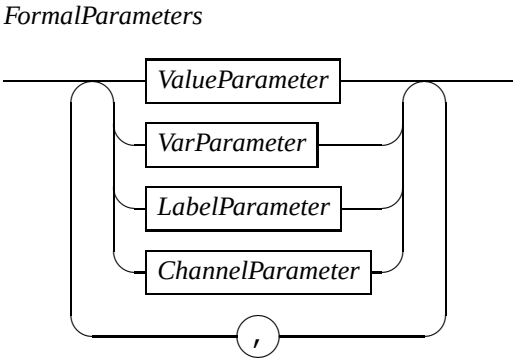

Figure 7.15: Raildiagram of *FormalParameters*.

With such a model, you must specify *everything* yourself.

WARNING: *Raw models are currently not supported.*

7.8 Formal parameters

The syntax of formal parameters is defined in the *FormalParameters* diagram in Figure 7.15. It consists of one or more parameter blocks seperated from each other by a comma. A parameter block is either a value parameter (by means of the *ValueParameter*[page 84] block), a variable parameter (through the *VarParameter*[page 84] block), a label parameter (by means of the *LabelParameter*[page 84] block), or a channel parameters (by using the *ChannelParameter*[page 85] block). Value parameters are explained in Section 7.9, details about variable parameters can be found in Section 7.10, details about label parameters are in Section 7.11, and channel parameters are explained in Section 7.12.

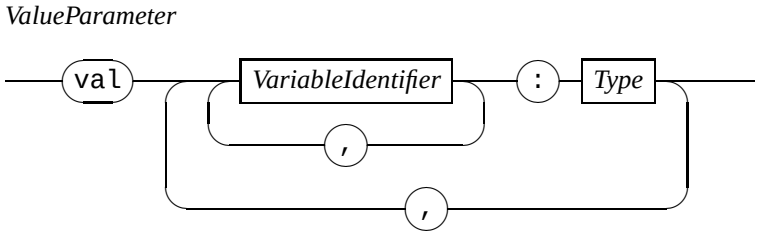

Figure 7.16: Raildiagram of *ValueParameter*.

Examples

As an example of formal parameters, consider the process definition below

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

The formal parameters of process `P` introduce a channel `c`, a continuous variable `w`, and a discrete variable `k`.

□

7.9 Value parameter

Value parameters introduce one or more formal value parameters in a function, process, or model. The syntax is defined in the *ValueParameter* diagram in Figure 7.16. Declaration of formal value parameters always starts with the keyword `val`, followed by a comma-seperated list of declarations. Each declaration consists of a list of variable identifiers, a colon, and finally the static type of the variables.

Each identifier represents a constant value of the stated static type which may be used (but not modified) in the body of the function, process, or model.

7.10 Variable parameter

With a variable parameter, defined in the *VarParameter* diagram in Figure 7.17, you can get access to variables outside the process definition. At process instantiation, the static type of the parameter has to match exactly (without widening) with the actual parameter. For parameters with `disc` and *cont* dynamic type, the dynamic type also has to match exactly. Variable parameters with a `alg` dynamic type may be coupled to actual parameters with any dynamic type.

7.11 Label parameter

With a label parameter, defined in the *LabelParameter* diagram in Figure 7.18 diagram, you can make labels defined outside the process or model available for use. At instantiation, the formal

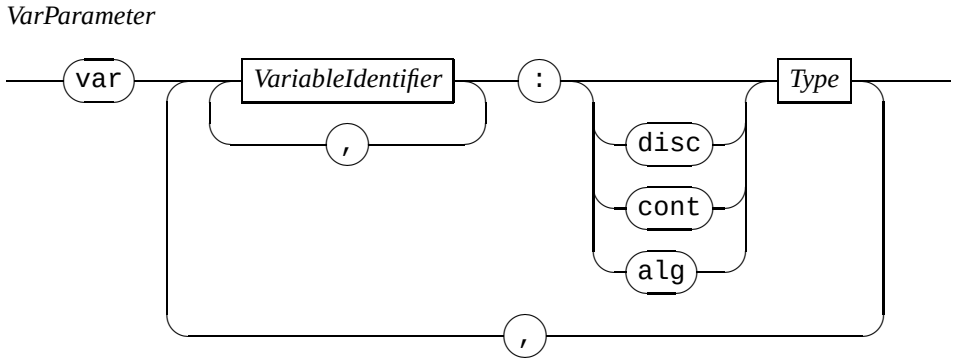

Figure 7.17: Raildiagram of *VarParameter*.

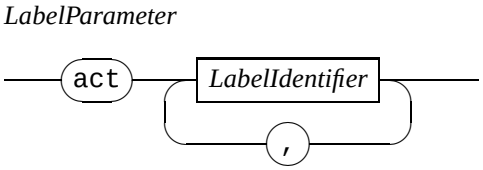

Figure 7.18: Raildiagram of *LabelParameter*.

labels listed in the *LabelParameter*[page 84] block of the process or model header are replaced by the actual labels.

7.12 Channel parameter

With channel parameters, you declare channels that can be used to synchronously exchange data between different processes. The syntax of a channel parameter is shown in the *ChannelParameter* block in Figure 7.19. Channel declarations are used in formal parameters.[page 83] 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).

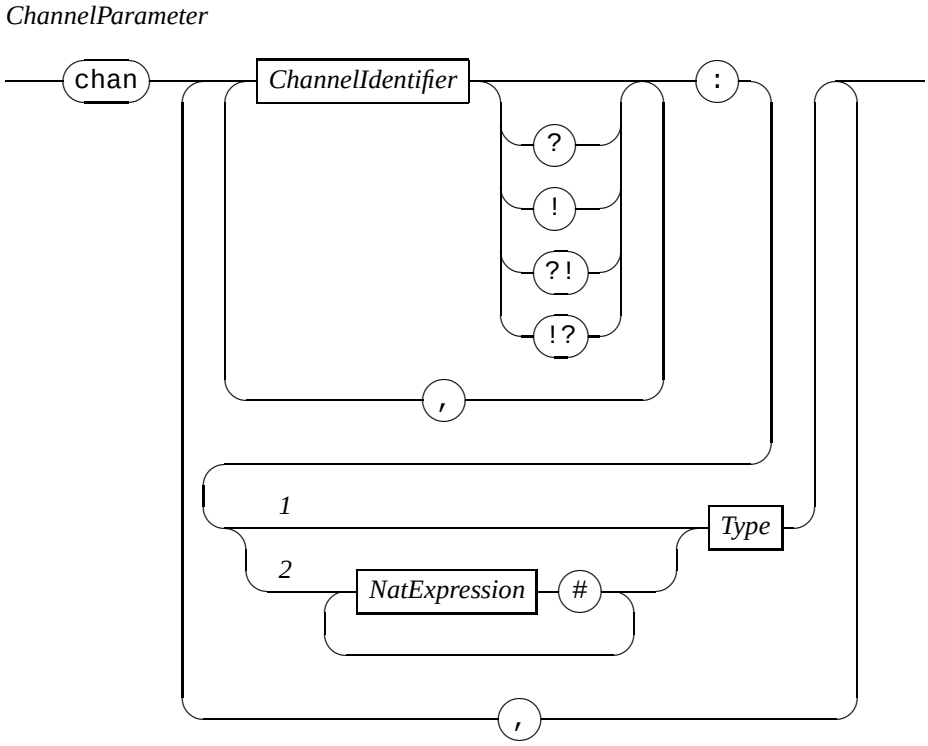

Figure 7.19: Raildiagram of *ChannelParameter*.

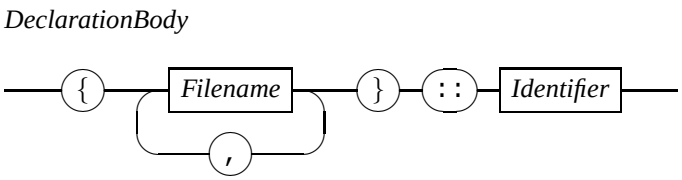

Figure 7.20: Raildiagram of *DeclarationBody*.

7.13 Declaration body

The body of a function or process declaration is described by the *DeclarationBody* diagram in Figure 7.20. 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] blocks 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 Chi 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.

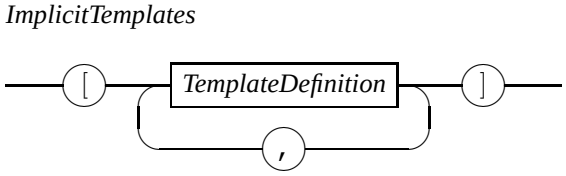

Figure 7.21: Raildiagram of *ImplicitTemplates*.

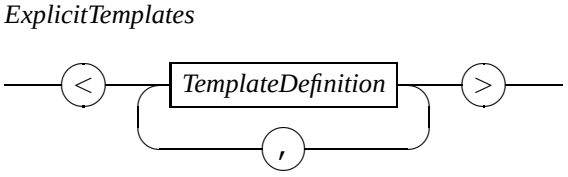

Figure 7.22: Raildiagram of *ExplicitTemplates*.

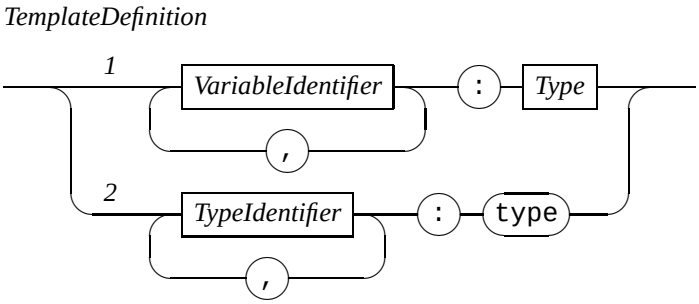

Figure 7.23: Raildiagram of *TemplateDefinition*.

7.14 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 7.21, the latter are defined in an *ExplicitTemplates* block shown in Figure 7.22. 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 7.23. 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.*
WARNING: *The implementation may add further restrictions to the set of allowed types in template definitions.*

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.

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 Empty distributions

The empty distributions are used internally as a default value for uninitialized values with a distribution type as their static type. You cannot draw a sample from an empty distribution. For completeness, the available empty distributions are listed below.

Function	Description
`none() -> (-> bool)`	Empty boolean distribution
`none() -> (-> nat)`	Empty natrual number distribution
`none() -> (-> int)`	Empty integer distribution
`none() -> (-> real)`	Empty real number distribution

A.2 Constant distributions

Function	Description
Constant distribution returning the specified value	
`constant(b: bool) -> (-> bool)`	Law & Kelton: - Mean (μ): b Variance (σ^2): 0 Range: $\{\text{true}, \text{false}\}$
Constant distribution returning the specified value	
`constant(n: nat) -> (-> nat)`	Law & Kelton: - Mean (μ): n Variance (σ^2): 0 Range: $[0, \infty)$
Constant distribution returning the specified value	
`constant(i: int) -> (-> int)`	Law & Kelton: - Mean (μ): i Variance (σ^2): 0 Range: $(-\infty, \infty)$
Constant distribution returning the specified value	
`constant(r: real) -> (-> real)`	Law & Kelton: - Mean (μ): r Variance (σ^2): 0 Range: $(-\infty, \infty)$

A.3 Discrete distributions

Function	Description
Bernoulli distribution with chance $p \in [0, 1]$ for true	
`bernoulli(p: real) -> (-> bool)`	Law & Kelton: Bernoulli(p) Mean (μ): p Variance (σ^2): $p \cdot (1 - p)$ Range: $\{\text{true}, \text{false}\}$
Bernoulli distribution with chance $p \in [0, 1]$ for 1	
`bernoulli(p: real) -> (-> nat)`	Law & Kelton: Bernoulli(p) Mean (μ): p Variance (σ^2): $p \cdot (1 - p)$ 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 (μ): $t \cdot p$ Variance (σ^2): $t \cdot p \cdot (1 - p)$ 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 (μ): $(1 - p)/p$ Variance (σ^2): $(1 - p)/p^2$ Range: $\{0, 1, 2, \ldots\}$

Discrete distributions, continued

Function	Description
Poisson distribution with rate $r¿0$	
`poisson(r: real) -> (-> nat)`	Law & Kelton: P(r) Mean (μ): r Variance (σ^2): r Range: $\{0, 1, 2, \ldots\}$
Discrete uniform distribution with $a < b$	
`uniform(a, b: nat) -> (-> nat)`	Law & Kelton: DU(a, b-1) Mean (μ): $(a + b - 1)/2$ Variance (σ^2): $((b - a)^2 - 1)/12$ 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 (μ): $(a + b - 1)/2$ Variance (σ^2): $((b - a)^2 - 1)/12$ Range: $\{a, a + 1, a + 2, \ldots, b - 1\}$

A.4 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 (μ): $a/(a + b)$ Variance (σ^2): $a \cdot b/((a + b)^2 \cdot (a + b + 1))$ 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 (μ): $m \cdot b$ Variance (σ^2): $m \cdot b^2$ Range: $[0, \infty)$
Negative exponential distribution with scale parameter $b > 0$	
`exponential(b: real) -> (-> real)`	Law & Kelton: expo(b) Mean (μ): b Variance (σ^2): 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 (μ): $a \cdot b$ Variance (σ^2): $a \cdot b^2$ Range: $[0, \infty)$
Lognormal distribution with location parameter a and scale parameter $b > 0$	
`lognormal(a, b: real) -> (-> real)`	Law & Kelton: LN(a, b) Mean (μ): $e^{a+b/2}$ Variance (σ^2): $e^{2 \cdot a+b} \cdot (e^b - 1)$ Range: $[0, \infty)$

Continuous distributions, continued

Function	Description
Normal distribution with location parameter a and scale parameter $b > 0$	
`normal(a, b: real) -> (-> real)`	Law & Kelton: N(a, b) Mean (μ): a Variance (σ^2): 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 (μ): $(a + b + c)/3$ Variance (σ^2): $(a^2 + b^2 + c^2 - a \cdot b - a \cdot c - b \cdot c)/18$ Range: $[a, c]$
Random distribution, mainly intended for developers	
`random() -> (-> real)`	Law & Kelton: U(0, 1) Mean (μ): $1/2$ Variance (σ^2): $1/12$ Range: $[0, 1)$
Continuous uniform distribution from a to b, with $a < b$	
`uniform(a, b: real) -> (-> real)`	Law & Kelton: U(a, b) Mean (μ): $(a + b)/2$ Variance (σ^2): $(b - a)^2/12$ 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 (μ): $b/a \cdot \Gamma(1/a)$ Variance (σ^2): $(b^2)/a \cdot (2 \cdot \Gamma(2/a) - (\Gamma(1/a)^2)/a)$ Range: $[0, \infty)$

In the `weibull` function, $\Gamma(z)$ is the gamma function, defined as $\Gamma(z) = \int_0^\infty t^{(z-1)} \cdot e^{-t} \mathrm{d}t$ 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.