Document version: 2.0-1
Status: final
Date: 2023-06-16
Document id: 0760c000

TeSSLa Language Specification Version 2.0

Contents

1 Introduction

The TeSSLa language is a stream-based specification language suitable for defining the expected behaviour of a cyber-physical system. Its main features are that

- it supports the notion of different streams defining behaviour of parts of the underlying system by formulating functions from time points to values,
- each event comes with a timestamp from a single global clock and
- it is not bound to a fixed rate for sampling events but supports different rates, turning TeSSLa into an asynchronous specification language suitable for describing signal behaviour in arbitrary precision.

This document is the TeSSLa language specification. Its aim is to define the TeSSLa language in a precise way, both in terms of syntax and semantics. This document is not meant as a tutorial for neither the TeSSLa language nor for supporting tools, but it may be used to verify TeSSLa tools w.r.t. language conformance.

1.1 Copyright Notice

1.2 Document and Language Versioning

To refer to the right version of the TeSSLa languages, we use versioning schemas both for the TeSSLa language itself as well as for this specification document.

The TeSSLa language uses versioning with two numbers *major.minor*, for example *1.3*.

Incrementing these versions identifies the amount of change. *Major* version increment identifies a fundamental change in the language. This includes backwards incompatible changes, i.e. changes that render previously valid specifications invalid.

Minor version increment identifies new extensions to the language that do not impact compatibility, although previously unspecified behaviour may still be specified and additional diagnostics may warn against potential errors in previously warning-free programs.

The *document version* identifies the version of this document and consists of the language version, a status and a single number that is incremented with each draft or updated document, for example *1.3-draft-5*.

The status distinguishes between *draft*, *review* and *final*. If the status is *final*, status is skipped in the document version.

For example a document version of *1.3-draft-5* is the fifth draft for the language specification version 1.3.

The title page also contains a *document id* which changes whenever the content of the document changes, by creating a hash of the source files.

1.3 Structure of the Specification

This specification defines two languages: The TeSSLa language and the TeSSLa Core language, referenced as TeSSLa Core or Core.

While the TeSSLa language is intended to be a high-level specification language, the TeSSLa Core language is the common intermediate language used by different TeSSLa interpreters and engines. TeSSLa Core itself is not a subset of TeSSLa but a simple standalone language, although it has a high number of similarities with the TeSSLa language. A TeSSLa specification can be translated into a semantically equivalent specification in TeSSLa Core. This document covers how this translation is performed and defines the semantics of TeSSLa Core. The semantics of TeSSLa is then given by its translation into TeSSLa Core. Any translation of TeSSLa preserving this semantics, even without explicit translation to TeSSLa Core, is considered TeSSLa Language Specification conformant.

TeSSLa Core can be seen as more high-level than a typical intermediate language of programming languages, because it is still defined in terms of streams and not in terms

of processor operations. Intuitively a specification in TeSSLa Core is a graph of basic stream transforming operations, which still allows to perform very powerful optimizations targeted for specific interpreters or engines, e.g. those utilizing FPGAs or other specialized hardware designed for stream processing. Hence a TeSSLa specification can first be compiled to TeSSLa Core by a backend-independent compiler and afterwards be further translated or interpreted by different backends.

A basic concept behind TeSSLa is that so called extern references to constants, types and functions may be used, whose behaviour/translation has to be built into the specific compiler backend or defined in another language. This makes them the basic building blocks of the TeSSLa language: All basic functions and stream operations have to be defined by externs. This enables the execution of the same specification on highly diverse platforms from a PC to an FPGA.

The common lexical syntax of TeSSLa and TeSSLa Core is given in chapter 2. It defines how the byte stream is translated into a token stream. Subsequent chapters define the syntax using these token sequences.

The syntax and semantics of TeSSLa Core are given in chapter 3. This chapter defines the type system, how input and output streams are declared and how internal streams and value expressions are bind to names. Further the definition and application of functions is described.

Then chapter 4 covers the syntax and semantics of the TeSSLa language:

The top level structure of a TeSSLa specification is defined in section 4.1. This includes definitions for modules, streams, functions, types and annotations, which are described in the following sections.

In section 4.13 expressions on streams and values are defined. Also the constant evaluation and macro expansion which is crucial during compilation is described there.

TeSSLa is a statically typed language with parametric polymorphism and type inference. The type system of TeSSLa is equal to the one in TeSSLa Core, though in TeSSLa Core all types have to be given explicitly. The type inference rules needed for compilation are defined in section 4.15.

A definition of the mandatory operations for stream processing can be found in chapter 6. Likewise there are some basic functions, types and constants which have to be supported by every backend. These mandatory elements and their semantics are listed in chapter 5.

A TeSSLa specification may also refer to libraries, where some common functions are pre-defined. There is an official standard library and extension libraries available on https://www.tessla.io/.

A full documentation of the official libraries is not part of this document but can be found on https://www.tessla.io/.

1.4 List of Contributors

- Gunnar Bergmann
- Thiemo Bucciarelli
- Manuel Caldeira
- Lukas Convent
- Normann Decker
- Sebastian Hungerecker
- Hannes Kallwies
- Martin Leucker
- César Sánchez
- Torben Scheffel
- Malte Schmitz
- Volker Stolz
- Daniel Thoma

1.5 Resources

Additional resources about TeSSLa can be found on the web site https://www.tessla.io/. This includes

- an implementation of a TeSSLa compiler (including compilation to TeSSLa Core),
- an introduction to TeSSLa,
- an online implementation,
- additional tools for TeSSLa,
- news and additional information, as well as
- the documentation of the official libraries.

1.6 Notation

Specification code is written in typewriter font `40+2` or

```
def x: Int = 40 + 2
```

Syntactical definitions are also written in typewriter font.

Placeholders for token sequences or values are typeset in a cursive, mathematical notation: e_1.

2 Lexical syntax

This chapter defines the lexical syntax of the TeSSLa and TeSSLa Core languages. The lexical syntax is a regular language which defines the syntax rules for transforming a stream of bytes into tokens.

The following chapters define the higher-level syntax of TeSSLa Core and TeSSLa on a stream of tokens, where the tokens may be interspersed with whitespace characters (ASCII code point 32 and code point 9), but not by line breaks.

2.1 Unicode

Specifications are written with the Unicode Basic Multilingual Plane (BMP) character set encoded as UTF-8.

2.2 Notation

This specification uses an EBNF-like notation for defining the syntax of the TeSSLa language.

Each rule is in the form

`NonTerm` `::=` *pattern*

where *pattern* defines a pattern, which the nonterminal `NonTerm` can expand to.

Lexical tokens are typed in upper case letters.

The following table sums up the production rules used in this document:

pat`?`	Optional
`(`*pat*`)`	Grouping
`-`*pat*	All words not matching *pat*
pat`+`	One or more repetitions of *pat*
pat`*`	Zero or more repetitions of *pat* *pat*`*` is equivalent to *pat*`+?`
pat`,+`	One or more repetitions of *pat*, separated by comma *pat*`,+` is equivalent to *pat* `(',' ` *pat*`)*`

pat`,*`	Zero or more repetitions of *pat*, separated by comma *pat*`,*` is equivalent to `(`*pat*`),+?`
pat_1 `|` pat_2	Alternatives
‘*term*’	Terminal – Sequence of characters
`[`*begin*`-`*end*`]`	Matches one character between code points *begin* and *end*
pat_1 `/` pat_2	Difference. All words that match pat_1, but do not match pat_2
`newline`	newline char (ASCII code 10, escape sequence \n) It may be optionally preceded by ASCII code 13 (\r).
`eof`	End of file
`any`	Any character

All patterns use the largest matching rule: At each point the longest possible substring matching a production rule is chosen.

2.3 Lexical Structure

2.3.1 Comments

```
COMMENT             ::= ‘#’ (-newline)* (newline | eof)
```

Comments start with the character ‘#’ and extend to the first end of line.

2.3.2 Identifiers

```
LETTER              ::= [a-z] | [A-Z] | ‘_’ | ‘$’
DECIMAL_DIGIT       ::= [0-9]
LETTER_OR_DIGIT     ::= LETTER | DECIMAL_DIGIT
IDENTIFIER_SEQ      ::= LETTER LETTER_OR_DIGIT*

ID                  ::= IDENTIFIER_SEQ / KEYWORD
LOCAL_ANNOTATION    ::= ‘@’ ID
GLOBAL_ANNOTATION   ::= ‘@@’ ID
```

Identifiers are sequences of alphabetic ASCII characters, the underscore ‘_’, the Dollar sign ‘$’ and the decimal digits, which additionally

- do not start with a digit and
- are not a keyword.

The dollar sign may only be used for identifiers in TeSSLa Core.

Upper and lower case is significant.

For example `tessla` and `TeSSLa` are distinct identifiers.

`x1` and `_1` are valid identifiers, but `1x`, `1_` or `1` are not.

2.4 Keywords

```
KEYWORD             ::= 'in'       | 'out'      | 'as'
                      | 'def'      | 'type'     | 'extern'
                      | 'lazy'     | 'expand'   | 'strict'
                      | 'if'       | 'then'     | 'else'
                      | 'module'   | 'import'   | 'include'
                      | 'static'   | 'liftable' | '__root__'
```

Keywords appear similar to identifiers, but they are reserved for specific use cases.

2.5 Literals

```
PRIM_LITERAL        ::= INT | FLOAT
```

Literals are the source code representation of a value of a built-in type.

Note that Boolean literals are defined as mandatory externs (see chapter 5)

2.5.1 Integer Literals

```
INT                 ::= DECIMAL_DIGIT+ | '0x' HEX_DIGIT+
HEX_DIGIT           ::= [0-9] | [a-f] | [A-F]
```

An integer literal is a sequence of digits with an optional prefix.

A decimal integer literal contains only digits between 0 and 9.

A hexadecimal integer literal contains digits between 0 and 9 and between `a` and `f` or `A` and `F`.

2.5.2 Float Literals

```
FLOAT               ::= DECIMAL_DIGIT+ '.' DECIMAL_DIGIT+
                      | DECIMAL_DIGIT+ ('.' DECIMAL_DIGIT+)?
                        'e' ('+'|'-') DECIMAL_DIGIT+
```

There are two notations for float literals. The first one is a sequence of digits with a dot. The second one uses an additional exponent, separated with an 'e' character.

2.5.3 String literals

```
STRING_LITERAL     ::= STRING_OPEN STRING_CLOSE

STRING_OPEN        ::= '"' STRING_ELEMENT*
STRING_CONTINUE    ::= STRING_ELEMENT*
STRING_CLOSE       ::= '"'

STRING_ELEMENT     ::= CHAR_ESCAPE_SEQ
                     | (any - ('\' | '$' | '"'))

STRING_ELEMENT_FMT ::= STRING_ELEMENT - '%'
                     | '%%' | '%n'
STRING_OPEN_FMT    ::= 'f"' STRING_ELEMENT_FMT*
STRING_CONTINUE_FMT ::= STRING_ELEMENT_FMT*
STRING_CLOSE_FMT   ::= '"'

CHAR_ESCAPE_SEQ    ::= '\n' | '\r' | '\t' | '\$' | '\\' | '\"'
```

`STRING_ELMENT` is the basic building block for strings. A string element is either a plain character or an escaped sequence, determined by `CHAR_ESCAPE_SEQ`. The characters `\`, `$` and `"` can only occur as part of an escape sequence.

The `STRING_LITERAL` rule encodes string literals only. Unlike Core the TeSSLa language supports string interpolation and string formatting syntax which are more general than string literals.

The string formatting syntax is not part of the lexical grammar, because it allows the inclusion of expression into strings, which are no longer regular. Instead it is defined in section 4.13.3. Therefore only building-blocks are defined in this chapter.

`STRING_OPEN` begins a string, `STRING_CLOSE` terminates it and `STRING_CONTINUE` defines a sequence between interspersed formatting elements.

For the formatting syntax similar constructs are defined. `STRING_OPEN_FMT`, `STRING_CONTINUE_FMT` and `STRING_CLOSE_FMT` define a formatted string. The rule `STRING_ELEMENT_FMT` defines an element for the formatting syntax. It excludes the `%` character except for the sequences `%%` and `%n`.

2.6 Operators

```
INFIX_OPERATOR     ::= '<<'  | '>>'  | '>=' | '<=' | '<'  | '>'
                     | '>=.' | '<=.' | '>.' | '<.' |'!='  | '=='
                     | '&'   | '|'   | '^'  | '+'  | '-'  | '*'
                     | '/'   | '%'   | '+.' | '-.' | '*.' | '/.'
```

```
                      | '&&'  | '||'
UNARY_OPERATOR     ::= '~'   | '-'   | '-.' | '!'
```

The given sequences of characters form *operators*. Operators are split into infix and unary operators.

2.7 Time Unit

```
TIME_UNIT          ::= 'fs' | 'ps'  | 'ns' | 'µs' | 'us' | 'ms'
                     | 's'  | 'min' | 'h'  | 'd'
```

A time unit can be used to define time spans.

2.8 End of Statement

```
EOS                ::= ';' | newline | eof
```

A statement ends either at an semicolon, at a new line indicated by the `\n` (ASCII value 0x0A), or at the end of a file.

2.9 Spaces

In a TeSSLa specification an arbitrary number of spaces can be inserted between tokens.

2.10 Newlines

In some places EOS tokens are necessary, in other places newlines can be used for improving readability.

Any newline that is preceded by a backslash is ignored and does not constitute an EOS token.

Additional newlines can be used in the following positions without creating an EOS token:

- Directly after an unary operator
- Directly before, and after an infix operator
- Directly after an opening brace, bracket or parenthesis
- Directly before a closing brace, bracket or parenthesis
- Directly after a comma

- Directly after the keywords `in`, `out`, `def`, `liftable`, `type`, `module`, `as`, `if`
- Directly after the `:` of a type annotation
- Directly before, and after the `.` of a member access
- Directly before, and after the keywords `then`, `else`
- Directly after the `=>` of a lambda expression or function type
- Directly after the `=` of a definition or a member definition
- Directly after an annotation
- As part of an empty line that only consists of whitespaces

3 TeSSLa Core

The TeSSLa Core language is intended to be the common intermediate language used by different TeSSLa interpreters and engines. A TeSSLa compiler translates a specification given in TeSSLa into a semantically equivalent specification in TeSSLa Core.

TeSSLa Core primarily differs from TeSSLa in that

- no functions over streams (macros) exist,
- all types are explicit,
- expressions are not nested, i.e. arguments in expressions are exclusively variables or constants, and
- all definitions contain unique identifiers (this includes parameter identifiers and type parameters).

This chapter defines the syntax and semantics of the Core language itself. Semantics of event streams and operations on events or values are defined later in chapter 5 and chapter 6.

Syntactical production rules that are specific to the Core language are prefixed with a lowercase `c`, as in `cSpecification` instead of `Specification`.

3.1 Specification Structure

```
cSpecification     ::= (cStatement EOS)*
cStatement        ::= cOutputStream     | cInputStream
                    | cDefinition       | cExternDef
                    | cGlobalAnnotation
```

A specification consists of a sequence of statements.

Each statement is either a definition, an extern definition, an annotation definition, an input stream or an output stream declaration.

Definitions bind expressions (over constants, variables or input streams) to names. Stream definitions can be marked as output streams and printed subsequently by the backend if the input events they depend on are available.

```
UID               ::= ID
```

`UID` is a unique identifier, that can only be defined once in the specification.

In the following there will be a definition of the basic concepts values, types, expressions and annotations and afterwards a detailed introduction to the statements named above.

3.2 Values

Backends for TeSSLa Core internally have to support the following types in order to evaluate a specification:

- Booleans, Integers, Floats, Strings
- Streams of values
- *function representations*
- *record and tuple data structures*

Details on the characteristics of these types can be found further below.

Information on streams can be found in chapter 6 where also the stream operations are defined.

During expression evaluation errors may occur. There are the following distinct error kinds in TeSSLa:

Static error is an error that is raised during specification translation. A message with details is included in the implementation's output.

Panic immediately aborts evaluation and writes a message to the output.

Error value $\diamondsuit$ is an error raised by evaluation of a value expression.

Error event † is an error raised by evaluation of a stream expression. For details see chapter 6.

Error values and error events are similar to exceptions in other languages. They propagate the error and may contain additional information, but may be handled silently. In this case $\diamondsuit$ and † are treated as usual values by the backend. There are no mandatory operators in TeSSLa to retrieve detailed information from an error value but can be supported as extern operators by some backends.

For the backend it is possible to support further value types (e.g. sets, lists…). They are called extern types.

3.3 Types

```
cTypeArgumentList  ::= '[' cType,* ']'
cTypeParamList     ::= '[' ID,* ']'
cType              ::= ID cTypeArgumentList?
                     | cFunctionType
                     | cRecordType
                     | cTupleType

cFunctionType      ::= cTypeParamList '(' (('strict' | 'lazy')? cType),*
                       ')' '=>' cType
cRecordType        ::= '{' (ID ':' cType),* '}'
cTupleType         ::= '()' | '(' cType ',' cType,+ ')'
```

Types are either identifiers refering to type parameters, identifiers of extern types followed by their type arguments, function types, record types or tuple types.

Extern types are such types which are natively supported by the backend. Mandatory extern types are `Bool`, `Int`, `Float`, `String`, `Option[T]` and `Events[T]` for streams (see chapter 5 and chapter 6), but backends may also support futher extern types which can then be used in the specification. Extern types are solely identified by their name. While in the TeSSLa language extern types have to be explicitly defined, in Core they can be used without a further definition.

Function types consist of a list of type arguments (unique identifiers given in square brackets), parameter types with an evaluation strategy (strict or lazy) and a return type. Unlike in TeSSLa, arguments must have an explicit evaluation strategy in TeSSLa Core.

Records are data structures which contain named fields of arbitrary type. Record types consist of a list of identifiers (field names) and their types. The field names have to be distinct.

Tuples are data structures which contain unnamed (positional) fields of arbitrary type. Tuple types consist of the fields' types in their positional order.

In TeSSLa and TeSSLa Core tuples are equal to records with field names `_1`, `_2`, …

There are no tuples with one element. Tuples with no elements are used as unit values.

We call types, which are not based on a stream type in the following *value types*.

A TeSSLa Core specification must be sound according to the type checking rules defined in section 4.15. The types of certain expressions are also defined there.

3.4 Type Equality

Two types are distinct, unless one of the following rules applies

Extern Types

Two extern types $A[T_1, \ldots, T_n]$ and $B[T'_1, \ldots, T'_n]$ are equal if

- the names are equal and
- the type arguments are equal

Function Types

If the number of type arguments or the number of arguments is different, then the types are distinct.

Otherwise type A has the form

`[`$T_{A,1}, \ldots, T_{A,n}$`](`$P_{A,1}, \ldots, P_{A,m}$`) =>` R_A

and type B has the form

`[`$T_{B,1}, \ldots, T_{B,n}$`](`$P_{B,1}, \ldots, P_{B,m}$`) =>` R_B.

Let type B' be B with renamed type variables

$$B' = B[T_{B,1}/T_{A,1}][T_{B,2}/T_{A,2}]...[T_{B,m}/T_{A,m}].$$

A and B are equal if for every $1 \leq i \leq m$ $P_{A,i}$ and $P_{B',i}$ are equal and if R_A and $R_{B'}$ are equal.

$P_{A,i}$ and $P_{B',i}$ are equal if their type and their evaluation strategy (strict or lazy) are equal.

However a TeSSLa Core specification is still type correct if two function types which have to be equal differ exclusively in their evaluation strategy. In this case the backend has to perform a conversion.

Record Types

Two record types are equal if all their fields match.

More precisely, two record types A and B are equal if

- for every field in A there is a field in B with same name and same type

  and

- for every field in B there is a field in A with same name and same type.

Tuple Types

Tuple types are equal to the record type with the same field types and names `_1`, `_2`… in the tuple's order.

3.5 Expressions

```
cConstant          ::= PRIM_LITERAL   | STRING_LITERAL | 'true'
                     | 'false'        | '()'

cExpressionArg     ::= cConstant      | UID            | cExpArgTypeApp

cExpression        ::= cExternRef     | cConstant      | cFunctionDef
                     | cRecordDef     | cFunctionCall  | cRecordAccess
                     | cTupleDef      | cExpTypeApp
```

Expressions are either

- references to externs, whose behavior is resolved by the implementation,
- constants,
- function, record or tuple definitions
- function calls and record accesses, or
- type applications of them.

ExpressionArgs can be used as arguments in other expressions. These can be constants, identifiers or type applications of them.

The value to which certain expressions evaluate are given in the following.

3.5.1 Constants

Constants are integer, float and (basic) string literals, as well as the externs true and false and the unit value.

Evaluation

The value of a constant is the common interpretation of its textual representation. The unit value is equal to the value of the empty tuple expression.

3.5.2 Identifiers

Valid identifiers are either defined in the current specification or parameters.

Evaluation

If an identifier is defined in the current specification, an expression is tied to this identifier. The value of this expression is the value of the identifier.

If an identifier is a parameter, the value is dependent on the specific application of the function. See the corresponding sections of this chapter for details on how the exact value can be obtained in this case.

3.5.3 Extern References

```
cExternRef          ::= extern(STRING_LITERAL)
```

The reference to an extern value is represented by the keyword extern, followed by the name of the extern in brackets as string literal.

Extern references are the basic building blocks of a TeSSLa specification. They provide values and functions which are implemented or translated in a backend-specific way.

Externs are necessary to define the mandatory functions, which have to be supported by every implementation, listed in chapter 6 and chapter 5.

Evaluation

The value of the extern reference is determined by the implementation dependent on the name. This value can either be built into the implementation or dynamically loaded during runtime, e.g. from a library.

3.5.4 Function Definitions

```
cFunctionDef       ::= cTypeParamList? cFunctionParams '=>' cFunctionBody
cFunctionParams    ::= '(' (UID ':' ('strict' | 'lazy') cType),* ')'
cFunctionBody      ::= '{' (cDefinition EOS)* cExpressionArg '}'
```

A function definition consists of a list of type parameters (unique identifiers), a parameter list (unique identifiers with name and evaluation strategy, `strict` or `lazy`, and a type) and a body of definitions with a result expression (ExpressionArg) at the end. A skipped type parameter list equals an empty one.

Evaluation

The value of such a definition is a *function representation*. This *function representation* maps a list of values for `strict` parameters and expressions for `lazy` parameters to the result value in the following way:

If the argument value of any strict parameter is $\diamondsuit$, the result value is also $\diamondsuit$.

Otherwise the result value is the value of the result expression.

For the evaluation of the result expression it may be necessary to evalute parameter identifiers or identifiers which are defined in the body, or a surrounding scope.

- Parameter identifiers evaluate to the value of the corresponding argument. If the parameter is `lazy`, the expression which is passed as argument is evaluated.
- Identifiers defined in the body are evaluated the same way as identifiers defined directly in in the specification. However they may reference parameter identifiers and locally defined identifiers.

3.5.5 Call Expression

```
cFunctionCall      ::= cExpressionArg '(' cExpressionArg,* ')'
```

A call expression is an ExpressionArg, representing the function which is called, followed by a list of ExpressionArgs for the arguments.

The number of arguments must be equal to the number of parameters of the *function representation* the called expression evaluates to.

Evaluation

The expression in front of the first bracket can be evaluated to a *function representation*. For strict parameters of the *function representation*, the corresponding argument expressions are evaluated and passed as values, for lazy parameters the expressions themselfs are passed to this *function representation*.

The value onto which the *function representation* maps the argument values resp. expressions is the value of the function application.

3.5.6 Record Definitions

```
cRecordDef          ::= '{' cFieldDef,* '}'
cFieldDef           ::= ID '=' cExpressionArg
```

A record definition is a list of identifiers and corresponding expressions (ExpressionArgs) inside curly brackets.

Evaluation

The value of a record definition is a *record data structure*. For each identifier it stores the value of its corresponding expression.

3.5.7 Record Access

```
cRecordAccess       ::= cExpressionArg '.' ID
```

A record access is an ExpressionArg, representing the record which is accessed, followed by a dot and the name of the field which is accessed.

Evaluation

The ExpressionArg can be evaluated to a *record data structure*. The value of the record access is the value to which this data structure maps the identifier.

3.5.8 Tuple Definitions

```
cTupleDef           ::= '()'
                      | '(' cExpressionArg ',' cExpressionArg,+ ')'
```

A tuple definition is a list of ExpressionArgs inside parens. Tuples with one element do not exist, however empty tuples exist and are equal to the unit value.

Evaluation

The value of a tuple definition is a *record data structure*. It stores the value of the subexpressions to the names `_1`, `_2`, … in the given order. Therfore the fields of a tuple can be accessed via record accesses.

3.5.9 Type Application

```
cExpTypeApp        ::= cExpression ‘[’ cType,* ’]’
cExpArgTypeApp     ::= cExpressionArg ‘[’ cType,* ’]’
```

A type application is a `cExpression` or `cExpressionArg` followed by a list of type arguments. It can be used to explicitly name the types, which are used for the type parameters of the sub-expression (left to right). The number of type arguments equals the numer of type parameters of the sub-expression.

Evaluation

A type application has no effect on the evaluation and is just used to give type hints to the backend. Hence the value of a type application is the value of its sub-expression.

3.6 Annotation Usage

```
cGlobalAnnotation  ::= GLOBAL_ANNOTATION
                       ( ‘(’ (cExpression),* ‘)’ )
cLocalAnnotation   ::= LOCAL_ANNOTATION
                       ( ‘(’ (cExpression),* ‘)’ )
```

An annotation is an annotation sequence followed by an argument list. A global annotation is an identifier preceded by two `@` signs, while a local annotation starts with a single `@`.

Annotations can be used to pass additional information to the backend and connected tools. The semantics of an annotation is backend/tool specific. For each one there may be individual annotations.

A global annotation can be used anywhere in the specification and provides information concerning the whole specification. A local annotation can be used at input and output stream definitions and contains information related to this stream exclusively.

Annotations may get expressions or identifiers as parameters. These parameters may be of any type, they must match with the signature of the annotation definition of the given name.

3.7 Input Streams

```
cInputStream        ::= cLocalAnnotation* 'in' UID ':' cType
```

`in` *name*`: Events[T]` defines an input stream of the specification, where `T` is the type and *name* is the name of the stream.

name is part of the public interface of the specification. It is the name that is used for mapping input data to the respective streams.

Only valid instances of the given datatype `T` are allowed as input. This excludes $\diamondsuit$ and †.

The type used in the `in` statement must be `Events[T]` where `T` is a type not based on a stream or function type.

3.8 Output Streams

```
cOutputStream       ::= cLocalAnnotation* 'out' ID
```

`out` *s* defines an existent stream as output stream.

Given the events from the input streams, TeSSLa stream *s* is evaluated step-by-step and the events from *s* are written to an output trace.

An annotation `@$name(`*name*`)` where *name* is a string, is used to associate the output stream with a name.

`out` panics when a received event is either † or if its value contains or is $\diamondsuit$.

s has to be of type `Events[T]` with `T` a type not based on a stream.

3.9 Definitions

```
cDefinition         ::= 'def' UID ':' cType '=' cExpression
```

`def` *id*`:` *T* `=` *e* binds an expression to name *id*. This name must be unique. No other definition or parameter must contain such a name. The type of *e* must be *T*.

Cyclic definitions of type `Events[`*T*`]` for some *T* are generally not allowed. We call a definition cyclic if it binds an expression to a name such that the expression depends on that particular name. The first argument of the built-in event operators `last` (see section 6.6) and `delay` (see section 6.7) are not considered as dependencies regarding this definition because they are resolving the cyclic dependency by resolving the argument to a previous/future value of a stream. Other extern functions taking a `lazy Events[`*T*`]`

will be treated the same, their implementation has to handle the resolution of a potential cyclic definition in some way.

The compiler may support compile-time resolution of finite cycles (see section 4.14)

The order in which the definitions appear is not of relevance. An identifier can be used before it is defined.

4 TeSSLa

This chapter defines the TeSSLa language and how a TeSSLa specification can be translated to a semantically equivalent TeSSLa Core specification, which can then be evaluated according to the rules in chapter 3. By this translation the semantics of TeSSLa is defined. Note that any compilation perserving this semantics is valid according to this language specification, i.e. the translation into TeSSLa Core is not mandatory.

4.1 Specification Structure

```
Specification      ::= (Include EOS)* (Statement EOS)*
Include            ::= 'include' STRING_LITERAL
Statement          ::= ModuleDef        | VariableDef   | TypeDef
                     | FunctionDef      | AnnotationDef | ExternDef
                     | OutputStream     | InputStream   | Import
                     | GlobalAnnotation
```

A specification is a sequence of statements, where a statement is either a global annotation, or a definition of a module, variable, type, function, annotation or extern. It can also define an input stream, mark a stream as an output or import definitions from a module's namespace. Before any statement, a sequence of includes can be specified, which define additional TeSSLa source files to be included.

Translation into Core

At first, all included files are resolved and their contents appended to the specification. Therefore the string after the `include` keyword has to match a file path relative to the location of the translated TeSSLa file. Includes with absolute pathes are not supported. Implementations are also allowed to include some TeSSLa files (e.g. a standard library) implicitly without an explicit include statement in the specification.

The resulting TeSSLa specification is then flattened. The tree of expressions is traversed and every nested expression e, except for literals, is replaced with a new statement

`def` $name$`:` t `=` e

where $name$ is a fresh, unique identifier and t is the type of e.

Afterwards every statement in the specification is iteratively translated according to the following rules until it is directly contained in the Core language.

4.2 Types

```
TypePath           ::= ID ('.' ID)*
Type               ::= TypePath cTypeArgumentList?
                     | cFunctionType
                     | cRecordType
                     | cTupleType
TypeArgumentList   ::= '[' Type,* ']'
TypeParamList      ::= cTypeParamList
```

Types in TeSSLa are almost identical to types in TeSSLa Core. Identifiers may also refer to local type definitions, or type definitions within local modules. Additionally, the following restriction applies:

- Function types are not allowed to be parameterized when used as return type or parameter type, e.g. `[T](T) => T` is a valid function type, but not allowed as return type or parameter type.

Information which types certain expressions have, when a TeSSLa specification is type correct and how types can be inferred can be found in section section 4.15.

4.3 Annotation Definitions

```
AnnotationDef      ::= 'def' (GLOBAL_ANNOTATION | LOCAL_ANNOTATION)
                       cFunctionParams?
```

Annotation definitions provide a signature for an annotation. Global annotations (starting with `@@`) and local ones (starting with `@`) are defined in the same way.

Translation into Core

Annotation definitions are solely used for type-checking their usages in section 4.15. As such, they are not needed in Core and therefore discarded during the translation.

4.4 Annotation Usage

```
LocalAnnotation    ::= cLocalAnnotation
GlobalAnnotation   ::= cGlobalAnnotation
```

Annotation usages syntactically and semantically don't differ between TeSSLa and TeSSLa Core.

Global annotations can be used anywhere in the specification, local annotations only at in- and output streams.

Translation into Core

Annotation usages exists in Core and can directly be translated.

4.5 Input Streams

```
InputStream        ::= cInputStream
```

Input definitions are fully equal to Core. The name of an input stream has to be unique in the outermost scope.

Translation into Core

Input streams can be directly translated into Core.

4.6 Output Streams

```
OutputStream       ::= LocalAnnotation* 'out' Expression ('as' ID)?
                     | 'out' '*'
```

Marks a stream as an output, with an optional identifier as name for this output stream. Though output stream definitions already exist in Core, TeSSLa enables an advanced syntax for it.

In Core the expression after `out` is required to be of type `Events[T]` with a type `T` that does not contain stream or function types. Constant expressions and certain structures with streams contained therein are allowed in Tessla, and are implicitly split up and converted to simple streams (see section 4.15.2).

With `out *` every input stream, and every defined stream is made an output.

Translation into Core

An output stream of the form `out` e `as` s is translated as:

```
def s': t := e
@$name("s")
out s'
```

where t is the inferred type of expression e.

If the identifier is omitted, the output stream is annotated with `@$name("`*name*`")` instead, where *name* is a string representation of expression e.

`out *` is replaced by `out s` for every globally defined identifier `s` with type `Events[T]` including input streams, but not imported module members.

4.7 Modules

```
ModuleDef          ::= 'module' ID '{' (ModuleStatement EOS)* '}'
ModuleStatement    ::= ModuleDef | VariableDef | TypeDef | FunctionDef
                     | ExternDef | Import | AnnotationDef
```

A module creates a namespace, which can be used to avoid name collision.

It may contain submodules, and module imports, variable, extern, type, function and annotation definitions, but no input or output stream definitions.

Translation into Core

In Core modules are represented as records, where all imported variables and types are resolved. This means that a module

```
module mod {
    def x_1 = e_1
    ...
    def x_n = e_n
}
```

is equivalent to a record

```
def mod: { x_1 : T_1, ..., x_n : T_n } = {
    x_1 = e_1,
    ...,
    x_n = e_n
}
```

where $T_1, \ldots, T_n$ are the inferred types of $e_1, \ldots, e_n$, and is then translated as such.

4.8 Imports

```
Import              ::= 'import' (ID '.')* ID
```

Imports allow to import the definitions, types and annotations of a module into the current scope, such that its definitions can be used without using a member access.

Translation into Core

Imports are resolved by replacing each usage of an identifier of the imported module with a member access. During type checking, all type identifiers pointing to local, or imported type definitions are recursively resolved, such that Core only contains type identifiers pointing to type parameters. Imported annotation definitions are only used in type checking and then discarded, so they have no core representation.

As an example, given a module `A` which contains a definition `foo`, and a type `Bar`

```
import A
in x: Events[Bar]
out foo
```

will be translated to

```
in x: Events[A.Bar]
out A.foo
```

If the current scope contains a definition or type with the same name, that definition has priority over the imported one. Import of two modules containing a definition or type with the same name is not allowed.

4.9 Function Definition

```
FunctionDef         ::= 'liftable'? FunctionSignature '=' Expression

FunctionSignature   ::= 'def' ID TypeParamList? ParamList? (':' Type)?

ParamList           ::= '(' (ID ':' ('strict' | 'expand' | 'lazy')?
                        Type),* ')'
```

A function definition consists of a signature with an identifier, type parameters and parameters, and a result expression.

If the type parameter list is not empty, the parameter list can be skipped. It equals an empty parameter list then.

Constant definitions with type arguments can be achieved this way:

`def` *name*`[T_1] = ...`

A function definition can be `liftable`, which allows an implicit conversion from a function over values to a function over streams. This is explained in more detail in section 4.13.8.1.

If the evaluation strategy of a parameter is not explicitly given it is inferred in the following way:

- If the type is a stream type (`Events[T]`), set the strategy to `lazy`,
- if the type is a plain type parameter (`T`), set the strategy to `expand`,
- otherwise set the strategy to `strict`.

Parameters with evaluation strategy `expand` are treated specially in constant evaluation (see section 4.14).

Translation into Core

The generic form of a function definition

`def` $name$`[`$T_1, \ldots, T_m$`](`$p_1 : m_1 P_1, \ldots, p_n : m_n P_n$`):` R `=` res

is translated into a variable definition with a lambda expression.

`def` $name$`:` `[`$T_1, \ldots, T_m$`](`$m'_1 P_1, \ldots, m'_n P_n$`)` `=>` R `=`
`[`$T_1, \ldots, T_m$`](`$p_1 : m'_1 P_1, \ldots, p_n : m'_n P_n$`)` `=>` res

A missing parameter or type parameter list is equivalent to an empty one.

The evaluation strategy of a parameter m'_i in Core is converted to `strict` if m_i was `expand`, otherwise it is kept the same as in Tessla.

4.10 Extern Definitions

```
FullParamList      ::= '(' (ID ':' ('strict' | 'expand' | 'lazy')?
                       Type),* ')'
ExternDef          ::= 'liftable'? 'def' ID
                       TypeParamList? FullParamList? ':' Type
                       '=' cExternRef
```

As in Core, TeSSLa specifications may reference functions or constants built into the backend. An extern definition binds a reference to such an external value or function to an identifier and provides its signature.

If the evaluation strategies for a parameter is not specified, it will be inferred (see section 4.9).

Translation into Core

An extern definition

`def` $name$`[`$T_1, \ldots, T_m$`](`$p_1 : m_1 P_1, \ldots, p_n : m_n P_n$`):` R `= extern(`s`)`

is translated into a definition

`def` $name$`: [`$T_1, \ldots, T_m$`](`$m_1 P_1, \ldots, m_n P_n$`) => ` R `= extern(`s`)`

which is then in Core.

A skipped type parameter or parameter list equals an empty one.

The evaluation strategy of a parameter m_i' in Core is converted to `strict` if m_i was `expand`, otherwise it is kept the same as in Tessla.

If the extern definition in TeSSLa contains neither a parameter nor type parameter list, no translation is performed:

`def` $name$`:` T `= extern(`s`)`

is preserved in TeSSLa Core.

4.11 Variable Definition

```
VariableDef        ::= 'def' ID (':' Type)? '=' Expression
```

As in Core a variable definition binds an expression to a name.

Translation into Core

Variable definitions in TeSSLa are equal to those in Core and do not need to be translated. However the expressions which are assigned to the variable are translated.

4.12 Type Definition

```
TypeDef            ::= 'type' ID TypeParamList? '=' (Type | cExternRef)
```

`TypeDef` defines an alias for a type or introduces a new extern type. Extern types are identified by the string which is used in the `cExternRef`.

Translation into Core

Type definitions are only required for type checking (see section 4.15) and are therefore discarded during translation.

4.13 Expressions

```
Expression          ::= PRIM_LITERAL
                      | INT TIME_UNIT
                      | InterpolatedString
                      | UnaryOperator Expression
                      | Expression InfixOperator Expression
                      | IfThenElse
                      | Variable
                      | '(' Expression ')'
                      | BlockExpression
                      | CallExpr
                      | RecordCreation
                      | TupleCreation
                      | MemberAccess
                      | LambdaExpression
```

Expressions are also translated into Core and can be evaluated by the rules defined in chapter 3.

4.13.1 Literals

Literals define basic instances of the mandatory external types Int and Float. They already exist in Core.

Unlike in Core, Strings are not literals in TeSSLa but handled as InterpolatedString expressions instead (see section 4.13.3). The `Bool` values `true` and `false`, and the `Option` values `Some` and `None` are also not literals, but instead extern constants (see section 5.1, section 5.6).

Translation into Core

Literals are preserved during the translation to Core.

4.13.2 Time Units

An integer literal can be followed by a time unit. These time units refer the to common units.

`fs`	femtoseconds
`ps`	picoseconds
`ns`	nanoseconds
`µs`	microseconds
`us`	microseconds
`ms`	milliseconds
`s`	seconds
`min`	minutes
`h`	hours
`d`	days

Translation into Core

Time units are converted to a numeric literal in relation to a given resolution, called the *base time*.

> Example: If the *base time* is `20 ns`, then the value `2 µs` is converted to the literal `100`.

The *base time* is passed to the compiler during translation.

4.13.3 String Interpolation

```
String             ::= InterpolatedString | FormattedString
InterpolatedString ::= STRING_OPEN
                       (ExprInString STRING_CONTINUE)*
                       STRING_CLOSE
FormattedString    ::= STRING_OPEN_FMT
                       (ExprInString FormatCmd? STRING_CONTINUE_FMT)*
                       STRING_CLOSE_FMT
ExprInString       ::= '$' ID
                     | '${' Expression '}'
FormatCmd          ::= '%' FmtFlags? FmtWidth? ('.'FmtPrecision)? FmtType
```

String interpolation allows the inclusion of expressions into a string. These expressions are evaluated and the result is inserted into the string at the given position. There are two interpolation operations, one for identifiers `$`*foo* where *foo* is a single identifier and one for arbitrary expressions `${`*expr*`}`.

String interpolation does not allow additional spaces between the tokens.

4.13.3.1 Formatted Strings

A formatted string allows customized formatting.

The optional `FmtFlags` is a set of characters that modifies the output format.

The optional `FmtWidth` is a non-negative decimal integer, that specifies the minimal number of characters written to the output.

The behaviour of flags and precision depends on the `FmtType`.

4.13.3.2 Format Type

```
FmtType            ::= 's' | 'S' | 'd' | 'o' | 'x' | 'X'
                     | 'f' | 'e' | 'g' | 'G'
```

FmtType	allowed types	Behaviour
s, S	all	Converts the type to string using `toString`
d	Int	Format as a decimal integer
o	Int	Format as an octal integer
x, X	Int	Format as a hexadecimal integer
f	Float	Format as decimal number
e	Float	Format with scientific notation
g, G	Float	Scientific notation or decimal depending on precision and rounding

Besides these operators `%%` and `%n` can occur everywhere in the string, not just after an interpolated expression. `%%` produces a single `%` and `%n` produces a platform-specific line separator.

4.13.3.3 Flags

```
FmtFlags           ::= '-' | '#' | '+' | ' ' | '0'
```

Flag	Behaviour
-	Left justify within the given field of width `FmtWidth`.
+	Precede the output with a `+`, unless the value is negative.
␣	If no sign is written, write a space (0x20) instead.
#	Alternative form: - Precede with `0` for `o`, `0x` for `x` and `0X` for `X`.

Flag	Behaviour
	- Always include a decimal point for `f` and `e`. - Not supported for `s`, `S`, `g`, `G`.
`0`	Left-pad the number with `0` if a padding is specified.

4.13.3.4 Width

```
FmtWidth            ::= INT
```

Width is the minimum number of characters that is written. If fewer are written, then the formatted string is padded with either zeros, `0` if the flag `0` is given or with spaces (0x20) otherwise. The result is right-justified, unless the flag `-` is given.

4.13.3.5 Precision

```
FmtPrecision        ::= INT
```

For floating-point format flags, the precision specifies the number of decimal places written to the output.

Otherwise the precision specifies the maximum number of characters written until it is truncated. If precision is less than width, then the output is truncated to the number of characters specified by precision.

Precision is not applicable to the integer format flags `d`, `o`, `x` and `X`.

Translation into Core

Every expression s has an equivalent expression s' where each `$`p is replaced by `${`p`}`. Additionally, every `${`p`}` without a formatting string can be extended to `${`p`}%s`.

Now s' has the form "$content_0$`${`$expr_1$`}`$\%format_1 content_1 ...$`${`$expr_n$`}`$\%format_n content_n$".

Produce

- `def` e_i`:String = String.format(`$format_i$`,` $expr_i$`)` for all $1 \leq i \leq n$
- `def` s_0`:String =` $content_0$
- `def` s_i`:String = String.concat(String.concat(`s_{i-1}`,` e_i`),` $content_i$`)` for all $1 \leq i \leq n$

where all e_i and s_i are fresh identifiers.

The resulting String s_i then represents the formatted result of the initial expression s.

`String.concat` and `String.format` are mandatory built-in string operations. For details see section 5.5.

4.13.4 Operators

There are unary prefix, and binary infix operators. A unary operator precedes the expression, e.g. `-x`, whereas an infix operator is written between two expressions, e.g. `y-x`.

All operators are semantically equivalent to certain functions, that mandatorily have to be supported by the implementation. For details on these functions see chapter 5. The following tables define the name of the functions and the type signatures for each operator.

4.13.4.1 Infix Operators

Binary operator application of the form e_1 *op* e_2 where *op* is the operation and e_1 and e_2 are expressions is semantically equivalent to an expression m`(`e_1`, `e_2`)` where m is a function from module `Operators`.

operator symbol (*op*)	function name (m)	type signature
`&&`	`and`	`(Bool, Bool) => Bool`
`||`	`or`	`(Bool, Bool) => Bool`
`==`	`eq`	`[T](T, T) => Bool`
`!=`	`neq`	`[T](T, T) => Bool`
`>`	`gt`	`(Int, Int) => Bool`
`<`	`lt`	`(Int, Int) => Bool`
`>=`	`geq`	`(Int, Int) => Bool`
`<=`	`leq`	`(Int, Int) => Bool`
`>.`	`fgt`	`(Float, Float) => Bool`
`<.`	`flt`	`(Float, Float) => Bool`
`>=.`	`fgeq`	`(Float, Float) => Bool`
`<=.`	`fleq`	`(Float, Float) => Bool`
`+`	`add`	`(Int,Int) => Int`
`-`	`sub`	`(Int,Int) => Int`
`*`	`mul`	`(Int,Int) => Int`
`/`	`div`	`(Int,Int) => Int`
`%`	`mod`	`(Int,Int) => Int`

operator symbol (op)	function name (m)	type signature
`&`	`bitand`	`(Int,Int) => Int`
`|`	`bitor`	`(Int,Int) => Int`
`^`	`bitxor`	`(Int,Int) => Int`
`<<`	`leftshift`	`(Int,Int) => Int`
`>>`	`rightshift`	`(Int,Int) => Int`
`+.`	`fadd`	`(Float,Float) => Float`
`-.`	`fsub`	`(Float,Float) => Float`
`*.`	`fmul`	`(Float,Float) => Float`
`/.`	`fdiv`	`(Float,Float) => Float`

4.13.4.2 Unary Operators

Unary operator application of the form $op\ e$ where op is the operation and e an expression is semantically equivalent to an expression $m(e)$ where m is a function from module `Operators`.

operator symbol (op)	function name (m)	type signature
`!`	`not`	`(Bool) => Bool`
`-`	`negate`	`(Int) => Int`
`~`	`bitflip`	`(Int) => Int`
`-.`	`fnegate`	`(Float) => Float`

4.13.4.3 If then else

```
IfThenElse          ::= (‘static’)? ‘if’ Expression ‘then’ Expression
                        ‘else’ Expression
```

If-then-else is a ternary operation of the form `if` e_1 `then` e_2 `else` e_3 where e_i are expressions, and is semantically equivalent to a function application of `Operators.ite(`e_1`,` e_2`,` e_3`)`. With the `static` keyword, the e_2`,` e_3 expressions and the result is of type `Events[T]`, this is equivalent to the function application of `Operators.staticite(`e_1`,` e_2`,` e_3`)`.

4.13.4.4 Precedence and Associativity

The precedence of the infix operators is as follows (lowest to highest):

1. `IfThenElse`

2. `=>` (lambda function)
3. `||`
4. `&&`
5. `==`, `<`, `>`, `<=`, `>=`, `!=`, `>.`, `<.`, `<=.`, `>=.`
6. `|`, `^`
7. `&`
8. `<<`, `>>`
9. `+`, `-`, `+.`, `-.`
10. `*`, `/`, `%`, `*.`, `/.`
11. `!`, `-`, `~`, `-.` (unary)
12. `()` (function call)

    For example an expression of the form `a+b*c` parses as `a+(b*c)`.

The operators 2 to 9 (inclusive) are left-associative.

For example `a+b+c` parses as `(a+b)+c`.

The unary operators have a higher precedence and are right-associative.

For example `--x` ≡ `-(-x)`

The anonymous function creation is right-associative.

For example `(x: Int) => (y: Int) => y+x` ≡ `(x: Int) => ((y: Int) => (y+x))`

The function call is left-associative.

For example `f()()` ≡ `(f())()`

Translation into Core

Through application of the previously defined rules, operators are translated to function applications, which exist in Core.

4.13.5 Variable/Parameter Access

```
Variable          ::= ID
```

A variable is a non-qualified identifier within a given scope.

A scope is a region in which a declared item can be accessed by its name.

Within a scope all defined names have to be unique.

Scopes can be nested. Within the inner scope each name refers to the item that was declared within that scope or is a parameter belonging to this scope. Otherwise it refers to the item that is referred to by the name in the outer scope.

Scopes are created by

- modules,
- block expressions,
- functions (for the parameters) and
- lambda expressions (for the parameters).

Translation into Core

Non-qualified identifiers are looked up in their current scope and then replaced with fully qualified identifiers.

For block expressions this is explained in section 4.13.7.

In functions and lambda expressions the parameters and variables are replaced by unique identifiers.

For modules, the non-qualified identifier is replaced with the according member access.

4.13.6 Grouping

Expressions can be grouped by wrapping them into parenthesis.

The expression then has the form (e).

Translation into Core

In flat form grouping is implicitly given hence no explicit translation is performed.

4.13.7 Block Expressions

```
BlockExpression    ::= ‘{’ (VariableDef | FunctionDef EOS)*
                       Expression ‘}’
```

Variables and functions can be defined within the scope of a block expression and used to derive a result expression (expression at the end of the block). The block expression evaluates to the value of that result expression.

Translation into Core

The names of all variables defined within the block are with fresh unique names. All references to these definitions within the definition expressions is renamed to refer to the new name. These new definitions are inserted into the surrounding scope, which is valid because the new names are not in conflict. The block expression is replaced with the result expression, where all references to the block definitions are renamed to refer to their new names.

4.13.8 Call Expressions

```
CallExpr           ::= Expression TypeArgumentList?
                       (‘(’ FunctionArgList ‘)’)?
FunctionArgList    ::= PositionalArgument,*
                     | NamedArgument,*
                     | PositionalArgument,+ ‘,’ NamedArgument,+
NamedArgument      ::= ID ‘=’ Expression
PositionalArgument ::= Expression
```

Call expressions apply a function to a list of arguments. Arguments can be given in the positional order of the called function's parameters or in any order identified by the parameter names. It is also possible, to partially give the first `k` arguments by position, and then the rest in arbitrary order by the remaining names. Giving a parameter both by position and by name is not allowed. A call expression is valid if every parameter gets exactly one expression assigned.

A missing type argument list equals an empty one.

The argument list can be skipped if the type argument list is not empty (type application). It equals an empty argument list then.

Translation into Core

The call expression is translated into a call expression by resolving the use of named arguments and replacing them by according positional arguments.

If the called expression has type arguments, type checking will try to infer the type arguments from the given argument types, and wrap the called expression with the inferred type application. This inference may not succeed, in which case the type arguments have to be specified.

If an expression is a function that takes no arguments, an application will be inferred where necessary.

4.13.8.1 liftable

Functions can automatically be signal-lifted (see section 6.9), if the definition is prefixed with the keyword `liftable`

This means such functions can be applied to value types but also to stream types by implicitly converting them on their call site if needed:

A call of a function declared as

`liftable def` f`[`$T_1, ..., T_m$`](`x_1`:` m_1 P_1`, ... ,` x_N`:` m_N P_N`):` R `=` ...

can implicitly be converted to a call to f_{lift} defined as

`def` f_{lift}`[`$T_1, ..., T_m$`](`x_1`: Events[`P_1`], ... ,` x_N`: Events[`P_N`]): Events[`R`]`
`= slift`$_N$`(`x_1`, ...,` x_N`,` f`)`

if it is called with streams as arguments. `slift`$_N$ is a function mandatorily provided by the implementation (see chapter 6).

The lift is ambiguous and should fail if the function return type is a plain type parameter which at call site can either be inferred to as some `Events[T]` or `T`. In this case either the type application or the lift has to be specified explicitly. Likewise a function is not signal-lifted if this would lead to a type `Events[T]` where `T` is based on a stream type itself.

4.13.9 Record and Tuple Creation

```
RecordCreation     ::= '{ (ID '=' Expression),* '}'
TupleCreation      ::= '(' ')' | '(' Expression ',' Expression,+ ')'
```

Records are created with the syntax `{` id_1 `=` $e_1, ..., id_n$ `=` e_n`}`, where id_1 to id_n are mutually distinct. The result is a record with the members id_1 to id_n, where each member id_i evaluates to value of the expression e_i.

Tuples are created with the syntax `(`$e_1, e_2, ... e_n$`)`, and are evaluated to a record with field names `_1`, `_2`, … `_n`.

Translation into Core

Record and tuple constructors exist in Core (in a flattened version) as well and can be directly translated.

4.13.10 Member Access

```
MemberAccess       ::= Expression ‘.’ ID | ‘__root__’ ‘.’ ID
```

In the same way as in Core fields of tuples and records can be accessed by adding `.fieldName` to an expression that evaluates to a tuple/record, where `fieldName` is the name of the desired field.

The pseudo member access of `__root__` allows navigation to the outermost scope. Note that `__root__` is a keyword which cannot be used as an identifier for variables, therefore no naming conflicts can occur.

Translation into Core

Member accesses exists in Core and can be directly translated.

4.13.10.1 `__root__` member access

A member access on `__root__` performs name lookup on the root scope: The name resolution does not take place in the current scope, instead the access is handled as if it was performed in the global scope. This can be used to access a global definition from within a scope (see section 4.13.5) where that definition is shadowed by a local definition of the same name.

> For example `__root__.x` refers to a definition `x` from the global scope.

`__root__` member accesses are replaced with accesses to the unique identifiers of the referenced scope's member.

4.13.11 Lambda Expression

```
LambdaExpression   ::= TypeParamList?
                       LambdaParamList
                       (‘:’ Type)? ‘=>’ Expression

LambdaParamList    ::= ‘(‘ (ID (‘:‘ (‘strict‘ | ‘expand‘ | ‘lazy‘)?
                       Type)? ),* ‘)’
```

A lambda expression defines a function, unlike a function its parameter types do not have to be specified. In that case they have to be inferred (see section 4.15).

It introduces a new scope that contains all parameters and type parameters.

No type parameter list is equivalent to an empty list.

If not explicitly given, the evaluation strategies (`strict`, `expand`, `lazy`) are inferred just like they are for function definitions (see section 4.9)

Therefore a lambda expression has the most general form

`[`$T_1, \ldots, T_m$`](`$p_1 : m_1 P_1, \ldots, p_n : m_n P_n$`):R =>` *expr*

Translation into Core

Lambda expressions exist (as Function Definitions) in Core and can be directly translated.

The evaluation strategy of a parameter m_i' in Core is converted to `strict` if m_i was `expand`, otherwise it is kept the same as in Tessla.

If the expression after the `=>` is a block expression it is not translated but preserved as block expression. However all statements/expressions in the block expression are translated according to the usual rules.

4.14 Macro Expansion and Constant Evaluation

During compilation from TeSSLa to TeSSLa Core, constant expressions should be evaluated to constants and reified where possible. This includes evaluation of expressions and also expansion of function and lambda calls. This is done by substituting the call by its result expression with all parameters replaced. The expression can then be further evaluated except from the sub-expressions which were passed as lazy parameters.

A function or lambda expression which has type `Events[T]` for some parameter or as return type is called a macro. If such a macro cannot be removed by the constant evaluation, the resulting Core specification may contain an infinite loop over streams. That means, that a finite graph of stream operators in the resulting Core specification is not guaranteed. Backends can, but do not have to support resolving loops of an infinite graph at runtime. The compiler must emit a warning for any macro not removed.

The macro expansion must treat arguments with strategy `expand` as lazy.

4.15 Type Inference and Type Checking

The type inference is guaranteed to be successful, if the types of the following identifiers are known statically:

- Parameters in functions and lambda expressions,
- input streams,
- identifiers defined by externs, and

- recursively defined identifiers.

Type checking is allowed to try to infer a type for all of the above mentioned identifiers as a convenience feature for the user. There are cases where missing types for some of these identifiers can be inferred unambiguously.

For other identifiers the implementation has to be able to infer the correct type. This is done in the following way:

At first all type aliases defined by

`type` a `=` T

where T is a type, are expanded by replacing every occurrence of a with T. If a is defined inside (nested) modules, all accesses via type paths ($m_1.\ldots.m_n.a$) leading to a are also replaced by T.

The types of variables with unknown type are then determined step by step until the types of all variables are fully known. Therefore the types of the expressions which are bind to these variables are evaluated.

If, at any point, a required type is not equivalent to the inferred type, the type checking fails.

4.15.1 Expression types

Expressions are first translated into Core according to the rules of this chapter. Afterwards their type can be determined in the following way:

- An integer literal has type `Int`.
- A float literal has type `Float`.
- A string literal has type `String`.
- An identifier (variable/parameter access) has the type of the variable/parameter.
- A record creation expression `{`m_1 `=` $x_1, \ldots, m_n$ `=` x_n`}` has the record type `{`m_1`:`$T_1, \ldots, m_n$`:`T_n`}` where T_i is the type of x_i.
- A tuple creation `(`$x_1, \ldots, x_n$`)` where x_i has the type T_i has the type `{_1:` T_1`, ..., _n:` T_n`}`.
- A member access has the member's type in the base expresion's record type.
- A function call op`(`$x_1, \ldots, x_n$`)` has the function's return type.
- A type application has the type of its subexpression with the type parameters resolved from left to right.

- A function declaration of the form `[`$T_1, \ldots, T_m$`](`p_1`:` $m_1 P_1, \ldots, p_n$`:` $m_n P_n$`):` R `=>` ... has the type `[`$T_1, \ldots, T_m$`](`$m_1 P_1, \ldots, m_n P_n$`)` `=>` R.

  If functions have no evaluation strategy m_i given it is inferred in the following way:

  - If the parameter type is a stream type (`Events[T]`), it is denoted as `lazy`
  - if the parameter type is a plain type parameter `T`, it is denoted as `expand`
  - otherwise as `strict`.

  If a function declaration has the expected type except from the evaluation strategies it is still type correct since implicit conversion is possible in Core.

4.15.2 Implicit Conversion

If required, the type inference also performs implicit conversions. The conversion of `liftable` functions is already explained in section 4.13.8.1. More implicit conversions are described here.

4.15.2.1 Constants

Constants can be automatically converted to streams. Every constant c of value type T can be lifted to a stream of type `Events[`T`]` that has a single event with value c at timestamp 0 and no events otherwise.

A lifted constant c is equivalent to an expression c' where c' is defined by

`def` c'`: Events[`T`] = default(`c`, nil[`T`])`

where `default` and `nil` are functions mandatorily provided by the implementation (see chapter 6).

4.15.2.2 Member access

Member access is implicitly convertible to an operation over streams, Member access $s.m$ where m is the member and s has type `Events[`T`]` can be automatically converted to:

$\texttt{slift}_1$`(`s`, (x:` T`) => x.`m`)`

where $\texttt{slift}_1$ is a function mandatorily provided by the implementation (see chapter 6).

4.15.2.3 Constant Functions

To make generic constants such as `None[`T`]` nicer to use, type checking infers an application on expressions of type `[`T_1`,...,`T_n`]() =>` R if types $T_1, \ldots, T_n$ can be inferred.

5 Mandatory Operations and Constants on Values

There is a set of types, operators and constants which must be available in every implementation.

Depending on the backend they may be partly defined by TeSSLa expressions or by references to externs, which are implemented in the particular backend. Basic definitions are contained in the official standard library, which can be included during compilation. Alternatively the implementations are allowed to include an individual library instead, where these operations are defined.

Note that operators in TeSSLa are translated to calls of these mandatory functions described in this chapter (see section 4.13.4), which is why they have to exist.

5.1 Bool

`Bool` is an extern type. It has only two possible values `true` and `false`.

```
def true: Bool
def false: Bool
```

`true` and `false` are constants of type `Bool` with the constraint that

`true` $\neq$ `false`

All following functions using `Bool` must be defined in the module `Operators` and hence are accessible by `Operators.opName`.

5.1.1 If-Then-Else

```
liftable
def ite[T](condition: strict Bool, ifTrue: lazy T, ifFalse: lazy T): T
```

`ite` (if then else) is a conditional expression.

`ite` strictly evaluates the first argument `condition`.

Afterwards `ite` evaluates to

- `ifTrue` if `condition` evaluates to `true`
- `ifFalse` if `condition` evaluates to `false`
- ♢ if `condition` evaluates to ♢

5.1.2 Static-If-Then-Else

```
def staticite[T](condition: strict Bool, ifTrue: lazy Events[T], ifFalse:
lazy Events[T]): Events[T]
```

`staticite` (static if then else) is a conditional expression.

`staticite` strictly evaluates the first argument `condition`.

Afterwards `staticite` evaluates to

- `ifTrue` if `condition` evaluates to `true`
- `ifFalse` if `condition` evaluates to `false`
- ♢ if `condition` evaluates to ♢

5.1.3 Not

```
liftable
def not(arg: strict Bool): Bool
```

The *logical not* negates the value of the parameter.

`not(true)` ≡ `false`
`not(false)` ≡ `true`
`not(`♢`)` ≡ ♢

5.1.4 And

```
liftable
def and(lhs: strict Bool, rhs: lazy Bool): Bool
```

`and` is the *logical and* of two logical expressions.

The following equality holds:

`and(lhs, rhs)` ≡ `ite(lhs, rhs, false)`

5.1.5 Or

```
liftable
def or (lhs: strict Bool, rhs: lazy Bool): Bool
```

`or` is the *logical or* of two logical expressions.

The following equality holds:

`or(lhs, rhs)` $\equiv$ `ite(lhs, true, rhs)`

5.2 Comparison

```
liftable
def eq[T](strict lhs: T, strict rhs: T): Bool

liftable
def neq[T](strict lhs: T, strict rhs: T): Bool
```

`eq` tests two values for equality, `neq` for inequality.

Implementations should uphold the equalities `eq`(x, y) $\equiv$ `not(neq`(x, y)`)` and `eq`(x, x) $\equiv$ *true*.

However there are types which usually do not uphold these equalities for all values. For some floating point values `eq`(x, x) may evaluate to false.

5.3 Integer

`Int` is the integer type. The internal representation is backend specific but must at least be able to represent values from -2^{32} to $2^{32} - 1$. The behavior of operations which leads to under- or overflows of the resulting integers is undefined.

An integer value can be created with the use of integer literals.

All following functions for type `Int` must be defined in the module `Operators` and so are accessible by `Operators.opName`.

5.3.1 Integer Addition and Subtraction

```
liftable
def add(lhs: strict Int, rhs: strict Int): Int
liftable
def sub(lhs: strict Int, rhs: strict Int): Int
```

If either `lhs` or `rhs` is $\diamondsuit$, then the result is $\diamondsuit$. Otherwise the result is an operator specific value.

Addition with `add` evaluates to `lhs`+`rhs`. Subtraction with `sub` evaluates to `lhs`−`rhs`.

5.3.2 Additive Inverse

```
liftable
def negate(arg: strict Int): Int
```

If `arg` $\diamondsuit$, then the result is $\diamondsuit$. Application of `negate` evaluates to `-arg`.

5.3.3 Integer Multiplication

```
liftable
def mul(lhs: strict Int, rhs: strict Int): Int
```

If either `lhs` or `rhs` is $\diamondsuit$, then the result is $\diamondsuit$. Otherwise multiplication evaluates to `lhs`·`rhs`

5.3.4 Integer Division

```
liftable
def div(lhs: strict Int, rhs: strict Int): Int
liftable
def mod(lhs: strict Int, rhs: strict Int): Int
```

If either `lhs` or `rhs` is $\diamondsuit$ or if `rhs` is 0, then the result is $\diamondsuit$.

Otherwise the operators evaluate to a specific value.

The division operator `div` evaluates to `lhs`/`rhs`. If `lhs` is no multiple of `rhs`, then any fractional parts are truncated to zero.

`mod` evaluates to the remainder of the quotient `l`/`r`.

The equality

`add(mul(div(lhs,rhs), rhs), mod(lhs,rhs))` $\equiv$ `lhs`

holds, unless the operations produce ♢.

5.3.5 Bitwise Operations

```
liftable
def bitand(lhs: strict Int, rhs: strict Int): Int
liftable
def bitor(lhs: strict Int, rhs: strict Int): Int
liftable
def bitxor(lhs: strict Int, rhs: strict Int): Int
```

The operator performs a bitwise operation. The result is ♢ if either `lhs` or `rhs` is ♢. Otherwise the operation performs the logical operation on each pair of matching bits, which is *and* for `bitand`, *or* for `bitor` and *exclusive or* for `bitxor`.

5.3.6 Bit Flip

```
liftable
def bitflip(arg: strict Int): Int
```

The operator flips each bit in `arg`. The result is ♢ if `arg` is ♢.

5.3.7 Bit Shifts

```
liftable
def leftshift(lhs: strict Int, rhs: strict Int): Int
liftable
def rightshift(lhs: strict Int, rhs: strict Int): Int
```

The operator shifts each bit of the value in `lhs` by the value in `rhs`. If either `lhs` or `rhs` are ♢, then the result is ♢.

5.3.8 Integer Comparison

```
liftable
def gt(lhs: strict Int, rhs: strict Int): Bool

liftable
def geq(lhs: strict Int, rhs: strict Int): Bool

liftable
def lt(lhs: strict Int, rhs: strict Int): Bool
```

```
liftable
def leq(lhs: strict Int, rhs: strict Int): Bool
```

Compares integers. `gt` returns whether `lhs` is larger than `rhs`, `geq` whether `lhs` is larger or equal to `rhs`, `lt` whether `lhs` is less than `rhs` and `leq` whether `lhs` is less or equal than `rhs`.

If either `lhs` or `rhs` is $\diamondsuit$, then the result is $\diamondsuit$.

The following equalities hold:

$\texttt{gt}(x, y) \equiv \texttt{lt}(y, x), \texttt{geq}(x, y) \equiv \texttt{leq}(y, x)$

5.4 Float

`Float` is the floating point type.

A float value can be created with the use of float literals.

Details of the operations are left unspecified here. Implementations must comply to IEEE 754-1985.

All following functions must be defined in the module `Operators` and so are accessable by `Operators.opName`.

5.4.1 Float Addition

```
liftable
def fadd(lhs: strict Float, rhs: strict Float): Float
liftable
def fsub(lhs: strict Float, rhs: strict Float): Float
```

If either `lhs` or `rhs` is $\diamondsuit$, then the result is $\diamondsuit$. Otherwise the result is an operator specific value.

Addition with `fadd` evaluates to `lhs`+`rhs`. Subtraction with `fsub` evaluates to `lhs`−`rhs`.

5.4.2 Additive Inverse

```
liftable
def fnegate(arg: strict Float): Float
```

If `arg` is $\diamondsuit$, then the result is $\diamondsuit$. Application of `fnegate` evaluates to `-arg`.

5.4.3 Float Multiplication

```
liftable
def fmul(lhs: strict Float, rhs: strict Float): Float
```

If either `lhs` or `rhs` is ♢, then the result is ♢. Otherwise multiplication evaluates to `lhs`·`rhs`.

5.4.4 Float Division

```
liftable
def fdiv(lhs: strict Float, rhs: strict Float): Float
```

If either `lhs` or `rhs` is ♢ or if `rhs` is 0, then the result is ♢. Otherwise division evaluates to `lhs`/`rhs`.

5.4.5 Float Comparison

```
liftable
def fgt(lhs: strict Float, rhs: strict Float): Bool

liftable
def fgeq(lhs: strict Float, rhs: strict Float): Bool

liftable
def flt(lhs: strict Float, rhs: strict Float): Bool

liftable
def fleq(lhs: strict Float, rhs: strict Float): Bool
```

Compares floating point numbers. `fgt` returns whether `lhs` is larger than `rhs`, `fgeq` whether `lhs` is larger or equal to `rhs`, `flt` whether `lhs` is less than `rhs` and `fleq` whether `lhs` is less or equal than `rhs`.

If either `lhs` or `rhs` is ♢, then the result is ♢.

5.5 String

`String` is the type for 8 bit character strings of arbitrary length.

A string value can be created with the use of string literals.

All following functions must be defined in the module `String` and so are accessible by `String.opName`.

5.5.1 Conversion into String

```
liftable
def toString[T](s: strict T): String
```

If `s` is ♢, then the result is ♢. Otherwise the result is a string representation of the value of `s`. The details of this representation are implementation specific.

5.5.2 String Concatenation

```
liftable
def concat(lhs: strict String, rhs: strict String): String
```

If either `lhs` or `rhs` is ♢, then the result is ♢. Otherwise `concat` returns a new string which concatenates `lhs` and `rhs`.

5.5.3 String Formatting

```
liftable
def format[T](formatString: strict String, value: strict T): String
```

If either `formatString` or `value` is ♢, then the result is ♢. Otherwise the resulting string is `formatString`, where the % character and the following formatting instructions are replaced by a string representation of value, which is formatted according to the rules described in section 4.13.3.1.

5.6 Option

The type `Option[T]` is either a value of `Some(v)` where `v` is a value of type `T`, a captured error `Some(♢)` or it is the value `None`.

All following functions must be defined in the module `Option` and are thus accessible by `Option.opName`.

5.6.1 None

```
def None[T]: Option[T]
```

`None` is a generic constant of type `Option[T]`.

5.6.2 Some

```
liftable
def Some[T](value: lazy T): Option[T]
```

`Some(v)` is a function that constructs a value of variant `Some(v)`.

`Some` creates an object

- `Some(`v'`)` if `v` evaluates to v'
- `Some(`$\diamondsuit$`)` if `v` evaluates to $\diamondsuit$

The evaluation to `Some(`$\diamondsuit$`)` is possible because of the lazyness of the parameter `value`.

5.6.3 isNone

```
liftable
def isNone[T](opt: strict Option[T]): Bool
```

`isNone` tests if an option is a variant of `None`.

`isNone` returns

- `true` if opt is a variant of `None`
- `false` if opt is a variant of `Some`
- $\diamondsuit$ if opt is $\diamondsuit$

5.6.4 isSome

```
liftable
def isSome[T](opt: strict Option[T]): Bool
```

`isSome` tests if a option is a variant of `Some`.

`isSome` returns

- `true` if opt is a variant of `Some`
- `false` if opt is a variant of `None`
- $\diamondsuit$ if opt is $\diamondsuit$

The following equality holds: `isSome(t)` $\equiv$ `not(isNone(t))` for all `t`.

5.6.5 getSome

```
liftable
def getSome[T](opt: strict Option[T]): T
```

`getSome` extracts the value `v` from the variant `Some(v)`.

`getSome` returns

- `v` if opt is a variant of `Some(v)`
- ♢ if opt is a variant of `Some(`♢`)`
- ♢ if opt is a variant of `None`
- ♢ if opt is ♢

6 Mandatory Operations for Streams

The operations and types mentioned in this chapter have to be mandatorily supported by every TeSSLa compiler.

They may be partly defined by TeSSLa expressions or by externs which are supported by the backend. Basic definitions are contained in the official standard library, which can be included during compilation. Alternatively the implementations are allowed to include an individual library instead, where these operations are defined.

`Events[T]` is the type of streams over values of type `T`. There may be no streams based on streams, i.e. `T` must be a value type.

The *time domain* is a numeric. It includes the common operations on numbers. Time itself is never $\diamondsuit$, but some operations may return $\diamondsuit$. In this document this type is referred to as `TIME`, its corresponding value range as `Time`. `Time` must contain an element 0 as the first timestamp. Implementations may use `INT`, `FLOAT` or some other built-in type for the realization.

The data domain of a stream is $\mathcal{T} = \mathtt{T} \cup \{\dagger\}$. $\dagger$ is an error event. Semantically $\dagger$ denotes that a previous operation could not determine whether an event exists or not.

Note that the error value $\diamondsuit$ is still part of the data type `T`. Unlike $\dagger$ the value $\diamondsuit$ indicates a valid event with an erroneous datum.

For describing the semantics of the stream operations the following paragraphs define a notation for event streams. This notation is not part of the TeSSLa language.

A *finite event stream* over a data domain $\mathcal{T}$ is a function $f : \mathtt{Time} \to \mathcal{T}_\perp$ depicting from `Time` to $\mathcal{T}_\perp$ where $\mathcal{T}_\perp := \mathcal{T} \cup \{\perp\}$ and $f(t) \neq \perp$ holds only for a finite number of timestamps t.

We use $ticks(s)$ for the set $\{t \in \mathtt{Time} \mid s(t) \in \mathtt{T}\}$ of timestamps where s has events.

In the following we use the $[\![\cdot]\!]$ operator to express the finite event stream value of a stream expression. By the subsequent evaluation according to these rules the output streams of a TeSSLa specification can be determined dependent on the input streams.

TeSSLa operators are monotone, continuous and future-independent. This makes it possible to subsequently calculate the output events during receiving input events up to the timestamp until which the input streams are known.

In this document a set of basic stream functions is defined which is sufficient to describe any monotonous, continuous and future-independent transformation from input to output streams. For some backends it may be desirable to restrict these functions by skipping the delay operator. However this causes that no events with timestamps other than input timestamps can occur which makes TeSSLa less expressive.

The mentioned functions are defined top-level, i.e. not in a special module. For every function the semantics is described in the mentioned stream notation or as TeSSLa code.

6.1 Nil

```
def nil[T]: Events[T]
```

The `nil` stream is a stream, that never contains any events.

$$\forall_{t \geq 0} : [\![\texttt{nil}]\!](t) = \bot$$

6.2 Unit

```
def unit: Events[()]
```

The `unit` stream is a stream, that has a single unit event at timestamp zero and no events afterwards.

$$[\![\texttt{unit}]\!](0) = ()$$
$$\forall_{t > 0} : [\![\texttt{unit}]\!](t) = \bot$$

6.3 Default

```
def default[T](stream: strict Events[T], value: strict T):
           Events[T]
```

`default` initializes a stream with `value`.

For timestamp 0 the output stream contains

- the event of `stream` at 0 if it has one
- † if `stream` has an † at 0
- `value` otherwise

For other timestamps the output stream is equal to `stream`.

$$[\![\texttt{default}(\texttt{s}, v)]\!](0) = \begin{cases} [\![\texttt{s}]\!](0) & \text{if } [\![\texttt{s}]\!](0) \neq \bot \\ v & \text{otherwise} \end{cases}$$
$$\forall_{t>0} : [\![\texttt{default}(\texttt{s}, v)]\!](t) = [\![\texttt{s}]\!](t)$$

6.4 Time

```
def time[T](stream: strict Events[T]): Events[TIME]
```

The `time` operator carries an event for each event on the base stream with the value of the according timestamp.

$$\forall_{t>0}[\![\texttt{time}(\texttt{s})]\!](t) = \begin{cases} \dagger & \text{if } [\![\texttt{s}]\!](t) = \dagger \\ t & \text{if } t \in \mathit{ticks}([\![\texttt{s}]\!]) \\ \bot & \text{otherwise} \end{cases}$$

`time` ignores the values on the stream. It returns the current time even if the value is $\diamondsuit$.

6.5 Lift

```
def lift_N[T_1, T_2, ... T_{N+1}](stream_1: strict Events[T_1],
                                  stream_2: strict Events[T_2], ...
                                  stream_N: strict Events[T_N],
                                  f: (lazy Option[T_1], lazy Option[T_2], ...
                                      lazy Option[T_N])
                                     => Option[T_{N+1}]):
            Events[T_{N+1}]

def lift = lift_2
```

$\texttt{lift}_N$ lifts an N-ary function f on values to a function on streams by applying f to the stream values for every timestamp.

$$[\![\texttt{lift}_N(\texttt{s}_1, \ldots, \texttt{s}_N, \texttt{f})]\!] = z$$

where

$$z(t) = \begin{cases} \bot & \text{if } values(t) = \{\bot\} \\ \dagger & \text{if } values(t) = \{\bot, \dagger\} \text{ or } values(t) = \{\dagger\} \\ unwrap(f(s_1(t), \ldots, s_N(t))) & \text{otherwise} \end{cases}$$

$$values(t) = \{[\![\mathtt{s}_1]\!](t)\} \cup \ldots \cup \{[\![\mathtt{s}_N]\!](t)\}$$

$$s_i(t) = \begin{cases} \texttt{None} & \text{if } [\![\mathtt{s}_i]\!](t) = \bot \\ \diamondsuit & \text{if } [\![\mathtt{s}_i]\!](t) = \dagger \\ \texttt{Some}([\![\mathtt{s}_i]\!](t)) & \text{otherwise} \end{cases}$$

$$unwrap(x) = \begin{cases} l & \text{if } x = \texttt{Some}(l) \text{ for some } l \\ \bot & \text{if } x = \texttt{None} \\ \dagger & \text{if } x = \diamondsuit \end{cases}$$

f is the value of the function representation `f`.

The function f is only evaluated if at least one argument stream contains a valid event. This means that a lift operator can only modify and remove events, but never create new ones.

`Option`s are used to represent existence or absence of events (see chapter 5).

Due to auto conversion for `lazy`, `expand` and `strict` parameters (see section 3.4) it is also possible to pass an f which takes (some) parameters `strict` or `expand`. However this will not make any semantical difference since the `Option` containers the arguments are wrapped into already behave lazy.

$\texttt{lift}_N$ is defined at least until $N = 5$, but may be defined for larger N.

6.6 Last

```
def last[T, U](stream: lazy Events[T], trigger: strict Events[U]):
        Events[T]
```

The `last` operator carries an event with the last value of `stream` if `trigger` has an event.

$$[\![\texttt{last(s, t)}]\!] = z$$

where

$$z(t) = \begin{cases} \dagger & \text{if } [\![\texttt{t}]\!](t) = \dagger \text{ and } \exists_{t'<t,d} \mathit{isLast}(t, t', [\![\texttt{s}]\!], d) \\ \bot & \text{if } [\![\texttt{t}]\!](t) = \bot \text{ or } \forall_{t'<t} [\![\texttt{s}]\!](t') = \bot \\ d & \text{if } t \in \mathit{ticks}([\![\texttt{t}]\!]) \text{ and } \exists_{t'<t} \mathit{isLast}(t, t', [\![\texttt{s}]\!], d) \end{cases}$$

where

$$\mathit{isLast}(t, t', v, d) := v(t') = d \wedge \forall_{t''|t'<t''<t} v(t'') = \bot$$

holds if v's last event before timestamp t is at timestamp t' and has value d.

Note that the `stream` parameter of `last` is defined lazy to avoid infinite loops during macro expansion, since `last` may be used in a recursive manner in terms of its first parameter. Such a recursion is necessary to define a stream dependent on its last event.

6.7 Delay

```
def delay[T](delayTime: lazy Events[Int], reset: strict Events[T]):
        Events[Unit]
```

`delay` produces a unit event, always if the reset stream carries an event or when `delay` produces an event, delayed by a flexible time. With this operator events can be generated at timestamps where no input stream has an event. Furthermore between two input events infinitely many other events can be generated.

In detail `delay` takes a delay stream `d` (`delayTime`) and a reset stream `r` (`reset`).

If either $[\![\texttt{d}]\!](t) \in \{\dagger, \diamondsuit\}$ or $[\![\texttt{r}]\!](t) = \dagger$ or $[\![\texttt{d}]\!](t) \leq 0$ then `delay` panics. Otherwise

$$[\![\texttt{delay}(\texttt{d}, \texttt{r})]\!] = z$$

where

$$z(t) = \begin{cases} () & \text{if } \exists_{t'<t} [\![\texttt{d}]\!](t') = t - t' \wedge \mathit{setable}(z, [\![\texttt{r}]\!], t') \wedge \mathit{noreset}([\![\texttt{r}]\!], t', t) \\ \bot & \text{if } \forall_{t'<t} ([\![\texttt{d}]\!](t') \neq t - t' \vee \mathit{unsetable}(z, [\![\texttt{r}]\!], t') \vee \mathit{reset}([\![\texttt{r}]\!], t', t)) \end{cases}$$

with the following auxiliary functions:

$$\mathit{setable}(z, r, t') := z(t') = () \vee t' \in \mathit{ticks}(r),$$

which holds if z is ready to receive a new delay time with respect to reset stream r

$$\mathit{unsetable}(z, r, t') := z(t') = \bot \wedge r(t') = \bot,$$

which holds if z is not ready to receive a new delay time with respect to reset stream r

$$reset(r, t, t') := \exists_{t''|t<t''<t'} t'' \in ticks(r),$$

which holds if reset stream r contained an event between t and t' and

$$noreset(r, t, t') := \forall_{t''|t<t''<t'} r(t'') = \bot.$$

which holds if reset stream r does not contain an event between t and t'

Note that the `delayTime` parameter of `delay` is defined lazy to avoid infinite loops during macro expansion, since `delay` may be used in a recursive manner in terms of its first parameter. Such a recursion is necessary to define a stream with periodic events.

6.8 Merge

```
def merge_N[T](s_1: strict Events[T], s_2: strict Events[T], ...
               s_N: strict Events[T]): Events[T]
```

The `merge` operator combines streams with preference from left to right.

The semantics of `merge`$_N$ can be fully defined by other TeSSLa stream operators:

```
import Option

def merge_N[T](s_1: strict Events[T], s_2: strict Events[T], ...
               s_N: strict Events[T]): Events[T] = {
  lift_N(s_1, s_2, ... s_N, (x_1: lazy Option[T],
                             x_2: lazy Option[T], ...,
                             x_N: lazy Option[T]) => {
      if (!isNone(x_1))
      then x_1
      else if (!isNone(x_2))
      then x_2
      ⋮
      else if (!isNone(x_N))
      then x_N
      else None[T]
  })
}
```

Details about `Option` can be found in chapter 5.

`merge`$_N$ is defined at least until $N = 5$, but may be defined for larger N.

6.9 Signal lift

```
def slift_N[T_1, T_2, ... T_{N+1}](stream_1: strict Events[T1],
                                   stream_2: strict Events[T_2], ...
                                   stream_N: strict Events[T_N],
                                   f: (lazy T_1,
                                       lazy T_2, ...
                                       lazy T_N) => T_{N+1}) :
           Events[T_{N+1}]

def slift = slift_2
```

The signal lift applies a function to the current or last value of each stream.

It does not produce an event unless it has received at least one event on each stream. Afterwards it produces an event whenever it receives an event on any stream. This resulting event contains the value of the function `f` applied to the current or last value of each stream.

The semantics of $\texttt{slift}_N$ can be fully defined by other TeSSLa stream operators:

```
import Option

def slift_N[T_1, T_2, ... T_{N+1}](stream_1: strict Events[T1],
                                   stream_2: strict Events[T_2], ...
                                   stream_N: strict Events[T_N],
                                   f: (lazy T_1,
                                       lazy T_2, ...
                                       lazy T_N) => T_{N+1}) :
           Events[T_{N+1}] = {

  def s_1 = merge_N(stream_1, last(stream_1, stream_2), last(stream_1, stream_3),
                  ..., last(stream_1, stream_N))
  def s_2 = merge_N(stream_2, last(stream_2, stream_1), last(stream_2, stream_3),
                  ..., last(stream_2, stream_N))
  ⋮
  def s_N = merge_N(stream_N, last(stream_N, stream_1), last(stream_N, stream_2),
                  ..., last(stream_N, stream_{N-1}))

  lift_N(s_1, s_2, ..., s_N,
        (x_1: lazy Option[T_1], x_2: lazy Option[T_2], ...,
         x_N: lazy Option[T_N]) => {
            if (isNone(x_1) || isNone(x_2) || ... || isNone(x_N)) then
                None[T_{N+1}]
            else
```

```
                Some(f(getSome(x_1), getSome(x_2), ..., getSome(x_N)))
        })
}
```

Due to auto conversion for `lazy`, `expand` and `strict` parameters (see section 3.4) it is also possible to pass an f which takes (some) parameters `strict` or `expand`. If any of the streams of a strict parameter contains an error, the `slift` also evaluates to an error.

$\texttt{slift}_N$ is defined at least until $N = 5$, but may be defined for larger N.