TransLucid User Documentation

Jarryd P. Beck Blanca Mancilla John Plaice

School of Computer Science and Engineering
The University of New South Wales
SYDNEY NSW 2052, Australia

{jarrydb,mancilla,plaice}@cse.unsw.edu.au

Thu Apr 5 16:15:35 EST 2012

Contents

Explanations of Annotations

This documentation is work in progress, and the TransLucid interpreter itself is also under development. As a result, the documentation may include the following different kinds of annotation.

TODO: Items to be added to the documentation.

MISSING: Missing functionality in the implementation.

WRONG: The current implementation is wrong. It needs to be rethought and then fixed.

CLEANUP: The functionality will need extra precision. This might involve more careful error detection, or more precise specifications.

Chapter 1

Overview

In this document, we describe in detail how to write TransLucid programs and how to get them executed. The TransLucid programming environment is accessible in three ways:

`libtl`: programming in `C++`, and using the `C++` API for the `libtl` library directly;

`tltext`: programming in TransLucid textually, using the `tltext` program on a Unix/Linux box; and

`tlweb`: programming in TransLucid textually, using the `tlweb` Web interface, currently available at `http://translucid.web.cse.unsw.edu.au/tlweb`.

For the third case, you are welcome to simply point your browser to the above URL, and to experiment programming in TransLucid straightaway.

For the first two cases, you must download and install the TransLucid source, written in `C++11`, the latest `C++` standard, hence you need a working `C++11` compiler. Chapter 2 describes in detail how to do the complete installation, including installing the GNU `gcc` program.

The rest of the documentation is currently focused on `tltext`, but much of it will be relevant for the other cases as well. Documentation for these will be available as the APIs become stable.

TransLucid is a functional programming language in which the "value" of a variable encompasses all of its variance, with respect to all of the possible parameters that may influence it. These parameters are called *dimensions*, and as a result, a variable defines a multidimensional entity, where the number of dimensions is unlimited. Each dimension corresponds to a different parameter, some fixed—such as `time`—, others created as needed, as would happen when a new `where` clause with local parameters were entered, and still others are defined from an initial context created from the settings of the environment variables of the `tltext` process. However it is created or manipulated, a runtime context is simply a set of (*dimension*, *ordinate*) pairs, defining all the dimensions relevant to the behavior of the program.

Following is a program written in TransLucid and run in `tltext`. (Once installed, `tltext` is run from the command line, and execution is terminated with `Ctl-D`.) The program has two parts divided by `%%`. In the first part (lines `1`–`6`), we declare three variables and two dimensions, used as parameters for the variables. Variable `fortytwo` is constant, and is therefore not affected by these dimensions. Variable `sum` varies with respect to both dimensions `m` and `n`, while variable `fact` varies with respect to dimension `n`. (We give an overview of the syntax in Chapter 5.)

```
1 dim m;;
2 dim n;;
3 var fortytwo = 42 ;;
4 var sum = #!m + #!n ;;
5 var fact = if #!n == 0 then 1
6            else #!n * (fact @ [n <- #!n -1]) fi ;;
```

```
 7 %%
 8 fortytwo ;;
 9 fortytwo @ [n <- 10] ;;
10 sum ;;
11 sum @ [m <- 3, n <- 5] ;;
12 fact @ [n <- 10] ;;
```

The second part of the program (lines 8–12) consists of the expressions to be evaluated. The program can be run by typing `tltext -i p004.tl` and the expected output is

```
TLText...
// instant 0 beginning
// demand 0
42
// demand 1
42
// demand 2
spundef
// demand 3
8
// demand 4
3628800
// instant 0 end
```

During the evaluation of an expression, there is always a runtime context. This context can be implicit (defined by the runtime system), or partially defined within the program. Lines 9, 11 and 12 of the example program each partially define a context of evalution, by defining dimensions `n` and/or `m`. Lines 8 and 10 on the other hand, are using the implicit context, with no definitions for `n` or `m`. As expected, `fortytwo` gives the same result (42) with no further definition of the context, or when the context yields 10 for dimension `n`. On the other hand, `sum` yields an error (`spundef`), as it requires the runtime context to have ordinates for dimensions `m` and `n` and these are not defined (line 10); when these are respectively set to 3 and 5 (line 11), the value of `sum` is 8. Finally, the evaluation of `fact` requires continually perturbing the ordinate of `n` from 10 down to 0 in order to compute the 10-th factorial number, as defined is line 12.

A running TransLucid program is a TransLucid system S, which is a synchronous, reactive system. This means that a system S runs through a series of instants, each corresponding to the special dimension `time` taking value 0, then value 1, then 2, and so on. In instant 0, system S starts out empty. In each instant, the user of S may add, replace or delete some declarations before computation takes place. These declarations, affecting only the semantics of the system from that instant on, consist of equations, inputs, outputs, and demands for computation of these outputs, along with configuration information to allow proper parsing of the equations, and importing of host-language data types and data operators over these types.

In the example below, there are three instants, separated by `$$`. In each instant, the `time` special dimension is being probed.

```
%%
#!time ;;
$$
%%
#!time ;;
$$
%%
#!time ;;
$$
```

The expected output from a newly started system is

```
TLText...
// instant 0 beginning
// demand 0
0
// instant 0 end
// instant 1 beginning
// demand 0
1
// instant 1 end
// instant 2 beginning
// demand 0
2
// instant 2 end
```

If the text were being typed in interactively, then the interleaved input and output would look like this:

```
TLText...
%%
#!time ;;
$$
// instant 0 beginning
// demand 0
0
// instant 0 end
%%
#!time ;;
$$
// instant 1 beginning
// demand 0
1
// instant 1 end
%%
#!time ;;
$$
// instant 2 beginning
// demand 0
2
// instant 2 end
```

In the example below, there are four instants. There is one declaration for `y`, valid for all instants, and three declarations for `x`, each guarded by the current ordinate of the `time` dimension.

```
var x [time:0] = 2 ;;
var y = x + 1 ;;
%%
x ;;
y ;;
$$
var x [time:1] = 3 ;;
%%
x ;;
y ;;
$$
```

```
var x [time:2] = 4 ;;
%%
x ;;
y ;;
$$
%%
x ;;
y ;;
$$
```

The expected output is

```
TLText...
// instant 0 beginning
// demand 0
2
// demand 1
3
// instant 0 end
// instant 1 beginning
// demand 0
3
// demand 1
4
// instant 1 end
// instant 2 beginning
// demand 0
4
// demand 1
5
// instant 2 end
// instant 3 beginning
// demand 0
spundef
// demand 1
spundef
// instant 3 end
```

In each instant, the best definition for each variable is chosen. In instant 3, there is no definition for `x`, hence the errors for both `x` and `y`.

If we add one more declaration for `x`, as in

```
var x = 42 ;;
var x [time:0] = 2 ;;
var y = x + 1 ;;
%%
x ;;
y ;;
$$
var x [time:1] = 3 ;;
%%
x ;;
y ;;
$$
var x [time:2] = 4 ;;
```

```
%%
x ;;
y ;;
$$
%%
x ;;
y ;;
```

then the expected output is

```
TLText...
// instant 0 beginning
// demand 0
2
// demand 1
3
// instant 0 end
// instant 1 beginning
// demand 0
3
// demand 1
4
// instant 1 end
// instant 2 beginning
// demand 0
4
// demand 1
5
// instant 2 end
// instant 3 beginning
// demand 0
42
// demand 1
43
// instant 3 end
```

since the very first declaration for `x` is always valid, but instants `0`, `1` and `2` have more precise definitions of `x`. This is an example of *bestfitting*, where the best definition of a variable, with respect to the current running context, will be chosen when that variable is to be evaluated.

In the example below, the `fact` variable has two declarations, one for the (zero) base case, one for the recurrent case.

```
dim n;;
var fact [n:0] = 1 ;;
var fact [n:1..infty] = #!n * (fact @ [n <- #!n -1]) ;;
%%
fact @ [n <- 10] ;;
```

The expected output is

```
TLText...
// instant 0 beginning
// demand 0
3628800
// instant 0 end
```

The guard `[n:1..infty]` means any integer n such that $1 \leqslant n < \infty$. The integers being used are of type `intmp`, the GNU arbitrary-precision integer type, and `infty` is a special variable defined in TransLucid to represent infinity.

In the sample programs above, we have seen the two key aspects of TransLucid:

- the evaluation of expressions with respect to a dynamic, runtime context, and
- the bestfitting of the declarations of a variable with respect to the context.

As we shall see in the rest of the document, *every* aspect of the TransLucid language and interpreter is governed by these aspects. The power of TransLucid lies in taking advantage of these two aspects.

Chapter 2

Installing TransLucid on a Linux or Unix machine

2.1 Prerequisites

Installing TransLucid may take a while, because of prerequisites, listed below. Depending on the state of your system, you may not need to build all of these. For the purposes of discussion, we assume that the version numbers referred to in the instructions below are the ones we are using.

- GCC 4.7.0 or higher, we are using 4.7.0.
- ICU `4.6` or higher, we are using `49_1`.
- MPI 1.0.x or higher, we are using 1.5.5.
- BOOST `1_46_0` or higher, we are using `1_49_0`.
- TransLucid, latest version.

2.2 Summary

This is the full build sequence. If you already have some of these prerequisites, then you will not need to build everything.

- Set `PATH` and `LD_LIBRARY_PATH` variables.
- Ensure GCC is installed.
- Ensure ICU and MPI are installed.
- Ensure BOOST is installed.
- Download TransLucid.
- Install TransLucid.

2.3 Detailed Instructions

TransLucid needs to be compiled with a `C++` compiler which supports `C++11`, the new `C++` standard. We use GCC version 4.7.0, which provides a partial implementation of this standard.

The compilation and running of TransLucid also uses a number of the BOOST libraries. You will probably need to install a recent version of BOOST on your system; before this can be done, you will need to ensure you have recent versions of ICU and MPI.

Given that there are many packages to install, we recommend creating a directory called "`soft`". Under this directory, there should be a "`downloads`" subdirectory to store the downloads, a "`src`" subdirectory to do the compilations, and an "`install`" subdirectory to place the generated binaries, header files and libraries. The instructions given below assume this setup, which will make it easier to maintain, clean, remove, update and find the installed packages. All sequences of instructions assume that you are starting in the `soft` directory.

If you follow this advice, then all of the installation directories specified after `--prefix` in the instructions given below will be this "`install`" directory. This directory will also need to be added to the `PATH` and `LD_LIBRARY_PATH` environment variables.

Specifically, in the instructions below:

- `$SOFT` is the *full* path of the `soft` directory.
- `$SRC` is the `$SOFT/src` directory.
- `$DOWNLOADS` is the `$SOFT/downloads` directory.
- `$PREFIX` is the `$SOFT/install` directory.

2.3.1 ICU

TransLucid requires BOOST to be compiled with ICU, which is a collection of libraries providing Unicode and Globalization support for software applications. ICU can be downloaded from

```
http://site.icu-project.org/download
```

as a distribution tarball (`$DOWNLOADS/icu4c-49_1-src.tgz`). For detailed installation instructions, read the `INSTALL` file in the root directory of the source. Here is a short version of the instructions:

```
cd $SRC
tar xzf $DOWNLOADS/icu4c-49_1-src.tgz
cd icu/source
./configure --prefix="$PREFIX" --enable-static
make
make install
```

2.3.2 MPI

TransLucid requires BOOST to be compiled with MPI, which is a collection of Standard MPI (Message Passing Interface) runtime programs. BOOST needs the MPI header files to compile correctly. MPI can be downloaded from

```
http://www.open-mpi.org/software/ompi/v1.5/
```

as a distribution tarball (`downloads/openmpi-1.5.5.tar.bz2`). For detailed installation instructions, read the `INSTALL` file in the root directory of the source. Here is a short version of the instructions:

```
cd $SRC
tar xjf $DOWNLOADS/openmpi-1.5.5.tar.bz2
cd openmpi-1.5.5
./configure --prefix="$PREFIX"
make
make install
```

2.3.3 BOOST

BOOST provides free peer-reviewed portable C++ source libraries and the emphasis is on creating libraries that work well with the C++ Standard Library. It can be downloaded from

```
http://www.boost.org/
```

as a distribution tarball (`$DOWNLOADS/boost_1_49_0.tar.bz2`). Detailed installation instructions can be found at

```
http://www.boost.org/doc/libs/1_49_0/more/getting_started/index.html
```

The BOOST distribution must be informed that you wish the MPI code to be compiled in. Here is the complete instruction sequence:

```
cd $SRC
tar xjf $DOWNLOADS/boost_1_49_0.tar.bz2
cd boost_1_49_0
echo "using mpi ;" >> tools/build/v2/user-config.jam
./bootstrap.sh --with-icu="$PREFIX" --prefix="$PREFIX"
./bjam install
```

2.3.4 TransLucid

TransLucid's source code is hosted on `sourceforge.net` in a `git` repository. To get the source:

```
cd $SRC
git clone \
    git://translucid.git.sourceforge.net/gitroot/translucid/translucid \
    translucid-git
```

This creates a new directory, `translucid-git`, and clones the source there. (Note that if you already downloaded TransLucid from `sourceforge` yourself, you would know this first step, as you did it already!)

To build TransLucid:

```
cd $SRC
export PKG_CONFIG_PATH="$PREFIX/lib/pkgconfig:"
cd translucid-git
./bootstrap.sh
./configure --prefix="$PREFIX" \
            --with-boost="$PREFIX" \
            --disable-static
make
make install
```

The `PKG_CONFIG_PATH` line allows the `configure` script to find the `icu-uc` package. As for the `--disable-static` flag, it will make TransLucid compile faster.

Chapter 3

TLtext

The `tltext` program is the command-line interface to the TransLucid interpreter. TLtext evaluates expressions in a series of instants, one instant at a time, each in a context where the dimension `time` is one greater than at the previous instant. It takes as input a sequence of declarations (see Chapter 5) which add information to the system, followed by the symbol `%%`, then a sequence of expressions to evaluate, followed by the symbol `$$`, at which point the current instant is evaluated, the results printed to the output, and the system becomes ready to receive more input for the next instant.

The `tltext` program responds to the following command-line arguments:

```
--args arg            arguments to pass to TransLucid in the CLARGS variable
-d [ --debug ]        debug mode
-h [ --help ]         show this message
-i [ --input ] arg    input file
-o [ --output ] arg   output file
--uuid                print uuids
-v [ --verbose ]      verbose output
--version             show version
```

The input to TLtext is read from the file given by the argument to `--input`, or standard input if none is supplied. The output is written to the file given as the argument to `--output`, or standard output if none is supplied. The `--uuid` option prints the unique universal identifiers of each declaration as it is added to the system. The `--verbose` option prints expressions with the minimum necessary parentheses after parsing the expression and adding it to the system. The `--debug` option prints a number of diagnostics, some of which are useful to TransLucid programmers, others to the developers of the interpreter.

MISSING: The `--uuid` option is not implemented.

The `tltext` program can interact with the outside world through all of the means available to a Linux program, including environment variables, the command line, the return code, standard input, standard output, standard error, input files, and output files.

3.1 The initial context

The initial context for `tltext`, used as the default runtime context for the evaluation of expressions, is defined through the Linux environment and the non-positional command-line options. Suppose, for example, that running `printenv` yields these two lines, among others:

```
TERM=xterm
SHELL=/bin/bash
```

Suppose furthermore, that the command line is:

```
tltext --language=French
```

and that the input typed in by the user is of the form:

```
%%
#!TERM ;;
#!SHELL ;;
#!language ;;
```

the expected output is:

```
TLText...
// instant 0 beginning
// demand 0
"xterm"
// demand 1
"/bin/bash"
// demand 2
"French"
// instant 0 end
```

3.2 The return code

In each instant, the variable `RETURN` is evaluated in a context holding just the current time and the settings of the environment variables. Its default value is 0, as if there were a declaration from the beginning

```
var RETURN = 0 ;;
```

The user can add other declarations for `RETURN`. Should, in instant n, the value of `RETURN` be non-zero, then the `tltext` program will halt after completing instant n and place the value held by `RETURN` in the return code.

In Linux, return codes must be between 0 and 255. In `tltext`, return codes 0 through 127 are reserved for the interpreter, while return codes 128 through 255 are for the TransLucid programmer. Should a value outside of 0 through 255 be given to `RETURN`, then `tltext` will halt with its own error code.

For example,

```
%%
RETURN ;;
$$
var RETURN [time:1] = 255 ;;
%%
RETURN ;;
$$
```

has expected output

```
TLText...
// instant 0 beginning
// demand 0
0
// instant 0 end
// instant 1 beginning
// demand 0
255
// instant 1 end
```

If you then type `echo $?` on the command line, the output will be `255`.

On the other hand,

```
%%
RETURN ;;
$$
var RETURN [time:1] = 1000 ;;
%%
RETURN ;;
$$
```

has expected output

```
TLText...
// instant 0 beginning
// demand 0
0
// instant 0 end
// instant 1 beginning
// demand 0
1000
// instant 1 end
```

but typing `echo $?` on the command line, will yield `2`, because 1000 is not a valid return code.

3.3 Passing arguments to TransLucid

All options for the TransLucid interpreter must appear on the command line before the use of the `--args` command-line option. Once the token `--args` appears on the command line, all subsequent text is considered to be positional command-line arguments to the interpreter.

The positional command-line arguments are numbered from 0 up, and are accessible in the program by indexing the `CLARGS` variable with the `arg0` dimension, also starting from 0. If n command-line arguments are provided, indexing beyond position $n-1$ will simply yield an empty string, rather than an error. For example, with the following command line:

```
tltext --args hello world
```

and the following input given on standard input:

```
%%
CLARGS @ [arg0 <- 0] + " " + CLARGS @ [arg0 <- 1] + " " + CLARGS @ [arg0 <- 2] ;;
```

the following would be printed to standard output:

```
// instant 0 beginning
// demand 0
"hello world "
// instant 0 end
```

Chapter 4

Lexical Conventions

The TransLucid language has been designed to have a very flexible parser, in which new operators and identifiers can be introduced, for a wide variety of applications and in a wide variety of cultural situations. Before presenting the details of the lexical conventions of TransLucid, we introduce the Unicode character set which we use.

4.1 Background on Unicode

The Unicode character set is designed to cover all of the world's character sets, both actual and historical. In the "What is Unicode" summary on the Unicode Web site,

> Unicode provides a unique number for every character, no mattter what the platform, no matter what the program, no matter what the language.
>
> `www.unicode.org`

For each Unicode character, there is a unique code point (number), a unique name, and a set of character properties. For example, the entry for `"O"` in the Unicode Data database (`www.unicode.org/Public/UNIDATA/UnicodeData.txt`) reads:

```
004F:LATIN CAPITAL LETTER O;Lu;0;L;;;;;N;;;;006F;
```

If we ignore the default values, then we can read this as:

- The code point is `004F`, read as a hexadecimal number.
- The name is `LATIN CAPITAL LETTER O`.
- The general category is `Lu`, meaning upper-case letter.
- The bidirectional character type is `L`, meaning that in mixed left-to-right and right-to-left text, it is left-to-right.
- The bidirectional mirrored property is `N`, meaning that in mixed left-to-right and right-to-left text, it cannot be mirrored.
- The lowercase mapping is `006F`, i.e., letter `o`.

The numbers for the code points range from `0000` to `10FFFF`—read, once again, as hexadecimal numbers—with the range from `D800` to `DFFF` forbidden. The range `0000`–`FFFF` is called the Basic Multilingual Plane, as Unicode began as a 16-bit character set.

Each Unicode character belongs to a character class. The classes of interest here are:

`Letter` (`L`): any kind of letter from any language, including

`Letter, Case` (`Lc`)
`Letter, Lowercase` (`Ll`)
`Letter, Modifier` (`Lm`)
`Letter, Other` (`Lo`)
`Letter, Titlecase` (`Lt`)
`Letter, Uppercase` (`Lu`);

below, in grammars, a Unicode `Letter` will be written *Letter*;

`Separator` (`Z`): any kind of whitespace or invisible separator, including

`Separator, Line` (`Zl`)
`Separator, Paragraph` (`Zp`)
`Separator, Space` (`Zs`);

below, in grammars, a Unicode `Separator` will be written *Separator*;

`Symbol` (`S`): math symbols, currency signs, dingbats, box-drawing characters, etc., including

`Symbol, Currency` (`Sc`)
`Symbol, Modifier` (`Sk`)
`Symbol, Math` (`Sm`)
`Symbol, Other` (`So`);

below, in grammars, a Unicode `Symbol` will be written *Symbol*;

`Number` (`N`): any kind of numeric character in any script, including

`Number, Decimal Digit` (`Nd`)
`Number, Letter` (`Nl`)
`Number, Other` (`No`)

below, in grammars, a Unicode `Number` will be written *Number*.

Since a one-byte encoding cannot be used for Unicode, there are a number of possible ways of encoding Unicode in a byte stream. The two that we retain are UTF-8 for input and output, and UTF-32 for internal processing. The UTF-8 encoding uses a variable number of bytes to encode Unicode code points, and works as follows:

```
0x00000000 - 0x0000007F:
    0xxxxxxx

0x00000080 - 0x000007FF:
    110xxxxx 10xxxxxx

0x00000800 - 0x0000FFFF:
    1110xxxx 10xxxxxx 10xxxxxx

0x00010000 - 0x001FFFFF:
    11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

The lead byte designates unambiguously how many bytes will follow, and the relevant bits are the ones marked `x`.

The UTF-8 encoding ensures that transmission of information passes unambiguously from little-ending machines to big-endian machines and back.

Internally, all processing is done using UTF-32, i.e., all characters are manipulated as 32-bit unsigned integers.

4.2 Comments

A comment in TransLucid consists of a `//` followed by all the characters on the rest of the line.

4.3 Identifier literals

An identifier (*id*) in TransLucid is of the following form:

$$id ::= (\ Letter \mid \_\ )\ (\ Letter \mid Number \mid \_\ )^{+}$$

Examples of uses of unusual characters in identifiers are `H`$_2$`O` (water) and `_`^{3}`_2He` (helium-3).

The following TransLucid keywords are reserved.

- declaration introductions:

`data`	`dim`	`fun`	`in`	`out`	`var`
`bestselect`	`infixl`	`infixn`	`infixr`	`postfix`	`prefix`

- conditional expressions:

`if`	`then`	`elsif`	`else`	`fi`

- local declarations:

`where`	`end`

- Boolean values:

`true`	`false`

- special values:

`sperror`	`spaccess`	`sptype`	`spdim`	`sparith`
`spundef`	`spconst`	`spmultidef`	`sploop`	`now`

4.4 Operator literals

An operator (*op*) in TransLucid has the following form:

$$op ::= (\ Symbol\ \mid \texttt{!} \mid \texttt{\%} \mid \texttt{*} \mid \texttt{-} \mid \texttt{.} \mid \texttt{\&} \mid \texttt{/} \mid \texttt{:}\ )^{+}$$

with six exceptions:

`=`	`:`	`|`	`!`	`.`	`//`

The following operators are defined in the initial header.

- arithmetic operators:

`+`	`-`	`*`	`/`	`%`

- comparison operators:

`<`	`<=`	`>`	`>=`	`==`	`!=`

- Boolean operators:

`&&`	`||`

- range operator:

  `..`

4.5 Other literals

The remaining symbols used in TransLucid are given below:

- declaration punctuation:

  ```
  |       =       :=      ;;
  ```

- tuple manipulation:

  ```
  [       :       <-      ]
  ```

- context manipulation:

  ```
  #       @
  ```

- functional abstraction:

  ```
  \       \\      ->
  ```

- functional application:

  ```
  !       .
  ```

- general grouping

  ```
  (       ,       )
  ```

4.6 Character literals

A character literal consists of a single *cooked character* between single quotes (`'c'`). In the syntax for expressions, a character is written *Character*. A cooked character is either a printable Unicode character, or one of the following recognized escape sequences:

`\n` for a newline (`000A`);

`\r` for a carriage return (`000D`);

`\t` for a horizontal tab (`0009`);

`\'` for a single quote (`0027`);

`\"` for a double quote (`0022`);

`\\` for a backslash (`005C`);

`\uXXXX` where `XXXX` are four hex digits, for a Unicode character in the Basic Multilingual Plane, range `0000`–`FFFF`;

`\UXXXXXXXX` where `XXXXXXXX` are eight hex digits, for a Unicode character not in the Basic Multilingual Plane, range `10000`–`10FFFF`;

`\xXX` for a valid one-byte UTF-8 sequence, designating a Unicode character in the range `0000`–`007F`;

`\xXX\xXX` for a valid two-byte UTF-8 sequence, designating a Unicode character in the range `0080`–`07FF`;

`\xXX\xXX\xXX` for a valid three-byte UTF-8 sequence, designating a Unicode character in the range `0800`–`FFFF`;

`\xXX\xXX\xXX\xXX` for a valid four-byte UTF-8 sequeence, designating a Unicode character in the range `10000`–`10FFFF`.

When a cooked character is being printed out, the `\XX` escapes are not used. For a given character c, its printed form becomes

c if c is printable;

`\n` if c is a newline;

`\r` if c is a carriage return;

`\t` if c is a horizontal tab;

`\'` if c is a single quote;

`\"` if c is a double quote;

`\\` if c is a backslash;

`\uXXXX` if c is in the range `0000`–`FFFF`;

`\uXXXXXXXX` if c is in the range `10000`–`10FFFF`.

4.7 Cooked string literals

A cooked string literal of length n, $n > 0$, consists of a sequence of n cooked characters in double quotes ("$c_0 \cdots c_{n-1}$"). In the expression syntax, a cooked string is written *CookedString*.

4.8 Raw string literals

A raw string of length n is a sequence of placed between back quotes (‘$c_0 \cdots c_{n-1}$‘), uninterpreted, valid Unicode characters, ranges `0000`–`D7FF` and `E000`–`10FFFF`, with the exception of the backquote (`0060`) itself. In the syntax for expressions, a raw string is written *RawString*.

4.9 Integer literals

In the syntax for expressions, integers are written *Integer*. Their syntax is outlined below:

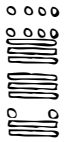
- Negative integers are an integer literal preceded by character `~`.
- Any integer starting with characters `1` through `9` is interpreted as base 10.
- The character `0` by itself corresponds to the value 0.
- An integer beginning with `01` followed by n more `1`s is base-1 notation for the number n.
- An integer beginning `0` followed by a character in the range `[2-9A-Za-z]` uses that second character as base-designator as follows:
  - `2` through `9` mean bases 1 through 9, respectively;
  - `A` through `Z` mean bases 10 through 35, respectively;
  - `a` through `z` mean base 36 through 61, respectively.

  The subsequent characters are interpreted as digits in that base. For a number in base n, only ‘digits’ from 0 to $n - 1$ may be used.

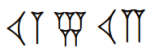
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
W	X	Y	Z	a	b	c	d	e	f	g	h	i	j	k	l
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
m	n	o	p	q	r	s	t	u	v	w	x	y	z		
48	49	50	51	52	53	54	55	56	57	58	59	60	61		

In computer science, the most commonly used bases are 2 (binary), 8 (octal) and 16 (hexadecimal); in everyday life, we use base 10 (decimal). For example, the number 39912 becomes

- `021001101111101000` in binary (base 2);
- `08115750` in octal (base 8);
- `0A39912` in decimal (base 10);
- `0G9BE8` in hexadecimal (base 16);
- `0K4JFC` in vigesimal (base 20), as used by the Mayans:



- `0yB5C` in sexagesimal (base 60), as used by the Babylonians:



Chapter 5

Declarations

The behavior of a TransLucid program is cut up into a series of "instants", numbered by integers starting from 0. In each instant, a set of declarations is provided to the interpreter. In this chapter, we summarize the different kinds of declaration.

It should be noted that in the overview, an instant was split into two, with a set of declarations and a set of expressions to be evaluated. The set of expressions is in fact just syntactic sugar for a set of declarations, with specific implementation in `tltext` and `tlweb`.

Below is an overview of all the declarations that can be added to the system. They are described in other chapters as referenced.

5.1 Global-only declarations

The following declarations may not appear within a `where` clause.

data Add an enumerated or inductively defined data type.

$$\begin{aligned} \mathit{datadecl} &::= \texttt{data}\ \mathit{id}\ \mathit{id}^*\ \texttt{=}\ \mathit{constr}\ (\texttt{|}\ \mathit{constr})^*\ \texttt{;;} \\ \mathit{constr} &::= \mathit{id}\ \mathit{tyvar}^* \\ \mathit{tyvar} &::= \mathit{id} \mid \texttt{(}\ \mathit{constr}\ \texttt{)} \end{aligned}$$

This mechanism is similar to the inductively defined data type of Haskell or other functional languages. The optional arguments (id^*) are type variables, and may be used as arguments for the type constructors.

Example. The following data type declaration:

```
data BinaryTree a = BinNode a BinaryTree BinaryTree
                  | BinLeaf a ;;
```

would result in the following declarations:

```
fun BinNode.a.b.c =
  [type <- "BinaryTree", cons <- "BinNode", arg0 <- a, arg1 <- b, arg2 <- c] ;;
fun BinLeaf.a =
  [type <- "BinaryTree", cons <- "BinLeaf", arg0 <- a] ;;
```

The following expression constructs a binary tree with root node 10, and left and right leaf nodes 4 and 15 respectively:

```
BinNode.10.(BinLeaf.4).(BinLeaf.15)
```

host Add an external identifier to the interpreter (§6.3).

$$\begin{aligned} \textit{hostdecl} ::= & \texttt{host}\ \textit{id}\ \texttt{=}\ \textit{hostarg}\ \texttt{;;} \\ \textit{hostarg} ::= & \texttt{HostDim}.\textit{int} \\ & \mid \texttt{HostType}.\textit{int} \\ & \mid \texttt{HostFunc}.\textit{int}.\textit{int} \end{aligned}$$

Identifier *id* is being mapped to an integer index, provided by a registry external to the interpreter. This mechanism allows dimensions, types and functions to be shared between TransLucid and the host environment, including other TransLucid systems.

- `HostDim`: There is a dimension registry that has provided a new number for this identifier, and TransLucid is being told of this mapping, and that this identifier should now be considered to be a dimension.
- `HostType`: There is a host-language atomic-type registry that has provided a new number for this identifier, and TransLucid is being told of this mapping, and that this identifier should now be considered to be an atomic type.

  It is assumed that, for type T, function `construct_`T, of arity 1, will be added, once T has been added. An additional definition will have to be given to `construct_literal`, informing it of the new `construct_`T.
- `HostFunc`: There is a function—whose address is determined by the loader—in the current interpreter for this identifier, and TransLucid is being told of the mapping from identifier to address, and that this identifier should now be considered to be a function whose arity is the second integer.

  The function *id* is applied to its arguments as *id*`!`E if its arity is 1, or *id*`!`$(E_1, \ldots, E_n)$, if its arity is $n \geqslant 1$.

op Add an operator to the interpreter (§6.3).

$$\begin{aligned} \textit{opdecl} ::= & \texttt{op}\ \textit{op}\ \texttt{=}\ \textit{oparg}\ \texttt{;;} \\ \textit{oparg} ::= & \texttt{OpPostfix}.\textit{string}.\textit{bool} \\ & \mid \texttt{OpPrefix}.\textit{string}.\textit{bool} \\ & \mid \texttt{OpInfix}.\textit{string}.\textit{bool}.\textit{assoc}.\textit{int} \end{aligned}$$

Symbol *op* is mapped to function identifier *id*, along with information for the parser.

- `OpPostfix`: A postfix operator is being declared to map to an identifier, to be interpreted as a function. The second argument states whether the function is call-by-name (`true`) or call-by-value (`false`).
- `OpPrefix`: A prefix operator is being declared to map to an identifier, to be interpreted as a function. The second argument states whether the function is call-by-name (`true`) or call-by-value (`false`).
- `OpInfix`: An infix operator is being declared to map to an identifier, to be interpreted as a function. The second argument states whether the function is call-by-name (`true`) or call-by-value (`false`). The third argument is the associativity (`assocn`, `assocl`, `assocr`) and the last argument is the precedence level.

hd Add a *hyperdaton*, a multidimensional, physical data structure providing an interface with the external world (§8).

$$\begin{aligned} \textit{hddecl} ::= & \texttt{hd}\ \textit{id}\ \textit{guard}\ \texttt{=}\ \textit{hdarg}\ \texttt{;;} \\ \textit{hdarg} ::= & \texttt{HdIn.}\textit{string} \\ & \mid \texttt{HdOut.}\textit{string} \end{aligned}$$

The part of hyperdaton *id* defined by *guard* is declared to come from (input) or go to (output) a physical location.

- `HdIn`: An input hyperdaton is being declared, with a given source, given as a URL in a string. The guard specifies when this source is valid, which means that an input hyperdaton could have several sources for different regions.
- `HdOut`: An output hyperdaton is being declared, with a given sink, given as a URL in a string. The guard specifies when this sink is valid, which means that an output hyperdaton could have several sinks for different regions.

assign Demand that the cells in an output hyperdaton be filled using expressions containing variables defined by input hyperdatons or by the system of equations.

$$\textit{assigndecl} ::= \texttt{assign}\ \textit{id}\ \textit{guard}\ \texttt{:=}\ E\ \texttt{;;}$$

The part of output hyperdaton *id* defined by *guard* is filled in by evaluating expression E.

5.2 Global or local declarations

In this section are declarations which may appear inside a `where` clause.

dim Declare an identifier to be a local dimension (§6.4.8).

$$\textit{dimdecl} ::= \texttt{dim}\ \textit{id}\ (\ \texttt{<-}\ E\ )^{?}\ \texttt{;;}$$

var Add a declaration for a variable (§7.1).

$$\textit{vardecl} ::= \texttt{var}\ \textit{id}\ \textit{guard}\ \texttt{=}\ E\ \texttt{;;}$$

fun Add a declaration for a function (§7.2).

$$\textit{fundecl} ::= \texttt{fun}\ \textit{id}\ \textit{params}\ \textit{guard}\ \texttt{=}\ E\ \texttt{;;}$$

bestselect Change the default behavior of how multiple declarations are resolved (§7.1).

$$\textit{bestdecl} ::= \texttt{bestselect}\ \textit{id}\ \texttt{=}\ \textit{op}\ \texttt{;;}$$

Chapter 6

Expressions

The evaluation of expressions in TransLucid takes place in a context-dependent manner, leading to two implications. The first is that the set of TransLucid primitives is larger than in other languages, as the context needs to be taken account by these primitives. The second is that the context-dependent evaluation gives a much greater flexibility in the interactions between the host language and TransLucid.

The presentation of expressions given in this chapter will therefore be made at a leisurely pace, bottom-up, covering all of the details necessary to understand even the meaning of a literal or a data operator, which are themselves unconventional.

6.1 Syntax

An expression is written E, and its concrete syntax is given below:

E ::=	*atomic*	
|	*id*	identifier
|	E *op*	postfix operator
|	*op* E	prefix operator
|	E *op* E	infix operator
|	`#`	context
|	`if` E `then` E `else` E `fi`	conditional
|	E `@` E	context perturbation
|	`[` E `<-` E `,` ... `,` E `<-` E `]`	tuple builder
|	E `!` `(` E `,` ... `,` E `)`	base application
|	`\` *id* `->` E	call-by-value abstraction
|	E `.` E	call-by-value application
|	`\\` *id* `->` E	call-by-name abstraction
|	E E	call-by-name application
|	E `where` *localdecls* `end`	local declarations
localdecls ::=	*dimdecl*	
|	*vardecl*	
|	*fundecl*	

where *atomic* corresponds to an atomic literal and *op* corresponds to declared operator symbols.

When parsing, TransLucid considers that postfix operators bind higher than all others, followed by prefix operators. Infix operators can have different binding precedence levels, which must be specified when they are declared.

6.2 Atomic values

In TransLucid, an atomic value is a pair consisting of a host-language type—itself defining a set of concrete values—and one of those concrete values. Here is the concrete syntax.

$$
\begin{aligned}
\mathit{atomic} ::= {} & \mathit{special} \\
| {} & \mathit{bool} \\
| {} & \mathit{int} \\
| {} & \mathit{char} \\
| {} & \mathit{rawString} \\
| {} & \mathit{cookedString} \\
| {} & \mathit{id}\ \mathit{rawString} \\
| {} & \mathit{id}\ \mathit{cookedString}
\end{aligned}
$$

The first six cases correspond to values of the predefined types, `special`, `bool`, `intmp`, `uchar` and `ustring`. The last two cases each correspond to a single lexeme, with no intervening whitespace.

6.2.1 Predefined type literals

As mentioned above, the predefined types do not require a type prefix.

Specials

Special values are used by the system to encode erroneous behavior during the evaluation of an expression. The set of possible values is quite restricted:

`sperror` An error occurred.

`spaccess` You can't get there from here.

`sptypeerror` A type error occurred.

`spdim` Invalid dimension.

`sparith` Arithmetic error.

`spundef` Undefined variable.

`spconst` Error building constant.

`spmultidef` Multiple definitions of a variable.

`sploop` A loop has been detected.

For each of these values v, the value `special"`v`"` is the same as v.

Booleans

Boolean values, or truth values, are used by conditional expressions. There are two possible values:

`true`

`false`

For each of these values v, the value `bool"`v`"` is the same as v.

Other predefined types

Literals of type `intmp`, `uchar` and `ustring` are presented in detail in Chapter 4.

6.2.2 Literals with type prefix

Suppose a value belongs to a type whose typename is T and for which the representation of the concrete value as an interpreted string literal is "v" (or ‘v‘). Then the canonical textual representation of the atomic value is T"v" (or T‘v‘), and it can appear as such in a program or be printed as such as output.

The parsing of such an atomic value takes place by executing `construct_literal."`T`"."`v`"`, which means applying function `construct_literal` to the arguments "T" and "v". For example, the expression

```
cheese"mascarpone"
```

becomes

```
construct_literal."cheese"."mascarpone"
```

where `construct_literal` is a function taking two string literals as argument, one for the typename and the other for the concrete value. The adding of new host-language types and the operators to manipulate values belonging to these types is given in Chapter 5.

The type name of a value is provided by the `typename` function. Hence

```
typename.(cheese"mascarpone")
```

should yield

```
"cheese"
```

The concrete value of a literal is provided by the `print` function. Hence

```
print.(cheese"mascarpone")
```

should yield

```
"mascarpone"
```

Note that the functions `construct_`T, `typename` and `print` are all context-dependent, so we could imagine that these could yield different results under different situations.

Suppose, for example, that the `construct_intmp` function had been extended so that it could read numbers in textual form in different languages, then we could imagine that the following input

```
%%
intmp"eleven"   @ [intstring <- true, lg <- "en"] ;;
intmp"dix-neuf" @ [intstring <- true, lg <- "fr"] ;;
```

could yield

```
11
19
```

Similarly, output of numbers, or of any other types, could be made context-dependent. Note that this is not currently implemented. Note also that once a literal is created, it remains as it was when it was created. The only potential context-dependence is upon its creation.

6.3 Operator symbols

There are no predefined operator symbols in TransLucid, although there is a standard header called `header.tl`, loaded by default by `tltext`, which defines a number of infix operators:

`+` mapping to `plus`, implemented for `intmp` and `ustring`.

`-` mapping to `minus`, implemented for `intmp`.

`*` mapping to `times`, implemented for `intmp`.

`/` mapping to `div`, implemented for `intmp`.

`%` mapping to `mod`, implemented for `intmp`.

`<` mapping to `lt`, implemented for `intmp`.

`<=` mapping to `le`, implemented for `intmp`.

`>` mapping to `gt`, implemented for `intmp`.

`>=` mapping to `ge`, implemented for `intmp`.

`==` mapping to `eq`, implemented for `bool`, `intmp`, `uchar`, `ustring`.

`!=` mapping to `ne`, implemented for `bool`, `intmp`, `uchar`, `ustring`.

`&&` mapping to `andalso`, implemented for `bool`.

`||` mapping to `orelse`, implemented for `bool`.

`..` mapping to `range`, implemented for `intmp`.

6.4 The evaluation of expressions

All TransLucid expressions are manipulated in an arbitrary-dimensional *context*, which corresponds to an index in the Cartesian coördinate system. As an expression is evaluated, the context may be *queried*, dimension by dimension, in order to produce an answer. In so doing, other expressions may need to be evaluated in other contexts.

6.4.1 Constants

The simplest expression in TransLucid is the *constant*. If we consider the literal `42`, its value is 42, whatever the context. Below, we show the variance of `42` if we allow dimension 1 to vary:[1]

	dim 1 →						
42	0	1	2	3	4	5	⋯
	42	42	42	42	42	42	⋯

What this table means is that in context $\{1 \mapsto 0\}$, i.e., where dimension 1 takes on the value of 0, the value of expression `42` is 42. The same holds true for all contexts $\{1 \mapsto j\}$, where $j \in \mathbb{N}$.

The next example uses two dimensions. In context $\{0 \mapsto 0, 1 \mapsto 0\}$, i.e., where both dimensions 0 and 1 take on the value 0, the value of expression `42` is still 42. The same holds true for

[1] Note that all our tables have dimension variance in $\mathbb{N}$. This is just to simplify the visual presentation.

all contexts $\{0 \mapsto i, 1 \mapsto j\}$, where $i, j \in \mathbb{N}$.

	dim 1 →						
`42`	0	1	2	3	4	5	⋯
dim 0 ↓ 0	42	42	42	42	42	42	⋯
1	42	42	42	42	42	42	⋯
2	42	42	42	42	42	42	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

The 'variance' of '42' in further dimensions would still yield, of course, the same result.

6.4.2 Contexts and dimension queries

For it to be possible for the context to affect the result of the evaluation of an expression, the context must be *queried*. To do this, we introduce `#`, which is the current context, a simple function from dimensions to values. To query the context for dimension 1 is written `#!1`. The `!` symbol is used for *base-function application*, which corresponds to the application of a function defined in a language such as C, i.e., the function is applied to all of the arguments at once, and the behavior of the function itself (not of the arguments) is not affected by the runtime context. Below we show how the value of `#!1` varies when we let dimension 1 vary:

	dim 1 →						
`#!1`	0	1	2	3	4	5	⋯
	0	1	2	3	4	5	⋯

In, say, context $\{1 \mapsto 4\}$, the value of expression `#!1` is 4. In fact, for all contexts $\{1 \mapsto j\}$, where $j \in \mathbb{N}$, the value of `#!1` is j.

Below, we see the variance of `#!1` in two dimensions. There we can see that no matter what the value of dimension 0, only the value of dimension 1 in the current context is of relevance for evaluating expression `#!1`. For all contexts $\{0 \mapsto i, 1 \mapsto j\}$, where $i, j \in \mathbb{N}$, the value of `#!1` is j.

	dim 1 →						
`#!1`	0	1	2	3	4	5	⋯
dim 0 ↓ 0	0	1	2	3	4	5	⋯
1	0	1	2	3	4	5	⋯
2	0	1	2	3	4	5	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

6.4.3 Pointwise operators

As in all languages, TransLucid expressions allow the use of operators such as `+` for addition, `*` for multiplication, and so on. In TransLucid, these operators are translated using the `op` declaration into identifiers which must define functions. Most of these functions are in turn ultimately implemented in terms of base functions with names such as `int_plus` or `int_times`. These last base functions are applied *pointwise* to their arguments. In other words, in a given context κ, the expression E_1 `+` E_2 will ultimately be translated into `int_plus!`$(E_1,\ E_2)$, yielding the sum of E_1 in κ and of E_2 in κ. Below is the variance of `#!0 + #!1` in two dimensions:

	dim 1 →						
`#!0 + #!1`	0	1	2	3	4	5	⋯
dim 0 ↓ 0	0	1	2	3	4	5	⋯
1	1	2	3	4	5	6	⋯
2	2	3	4	5	6	7	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

In context $\{0 \mapsto i, 1 \mapsto j\}$, for all $i, j \in \mathbb{N}$, `#!0 + #!1` has the value $i + j$. In the discussion below, we ignore the translation of the operators into functions, and pretend that it is the operators that are themselves applied pointwise.

6.4.4 Context changes

If the context can be queried in TransLucid, then it also needs to be changeable. This is done using the `@` operator, which takes a tuple as parameter and uses it to change the current context before continuing with the evaluation of the expression. Below, the expression `#!0 + #!1` is evaluated in a new context, which is created by incrementing dimension-1's ordinate by 1.

`(#!0 + #!1) @ [1 <- #!1 + 1]`	dim 1 →						
	0	1	2	3	4	5	⋯
dim 0 ↓ 0	1	2	3	4	5	6	⋯
1	2	3	4	5	6	7	⋯
2	3	4	5	6	7	8	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

So, expression `(#!0 + #!1) @ [1 <- #!1 + 1]` evaluates to $i + j + 1$ in context $\{0 \mapsto i, 1 \mapsto j\}$.

6.4.5 Factorial: version one

TransLucid, of course, needs variables, defined through equations. We introduce these with the factorial function, presented as a variable varying in dimension 1, whose first few entries can be found below:

	dim 1 →								
`fact`	0	1	2	3	4	5	6	7	⋯
	1	1	2	6	24	120	720	5040	⋯

The definition in TransLucid is recursive, with a base case and an inductive case.

```
var fact = if #!1 == 0 then 1
           else #!1 * (fact @ [1 <- #!1 - 1]) fi ;;
```

6.4.6 Ackermann: version one

The Ackermann function is one of the first recursive functions discovered that is not primitive recursive. It grows so fast that in general it cannot be computed once its first argument is greater than 3. Here it is presented as a variable varying in dimensions 0 and 1.

	dim 1 →						
`ack`	0	1	2	3	4	5	⋯
dim 0 ↓ 0	1	2	3	4	5	6	⋯
1	2	3	4	5	6	7	⋯
2	3	5	7	9	11	13	⋯
3	5	13	29	61	125	253	⋯
4	13	65533	⋯				
5	65533	⋯					
⋮	⋱						

Here is the definition for Ackermann in TransLucid.

```
var ack = if #!0 == 0 then #!1 + 1
          elsif #!1 == 0 then ack @ [0 <- #!0 - 1, 1 <- 1]
          else ack @ [0 <- #!0 - 1, 1 <- ack @ [1 <- #!1 - 1]] fi ;;
```

In the general (`else`) case, note that the nested context change is only changing the value for dimension 1, since the value for dimension 0 need not be changed. This is similar to what happens with differential equations, in which only the dimensions of relevance are written down.

6.4.7 Standard functions

A function can carry both *value parameters* and *named parameters*. Below are some standard TransLucid functions.

```
fun index.d   = #!d + 1 ;;
fun first.d X = X @ [d <- 0] ;;
fun next.d X  = X @ [d <- #!d+1] ;;
fun prev.d X  = X @ [d <- #!d-1] ;;
fun fby.d X Y = if #!d <= 0 then X else prev.d Y fi ;;
```

Each of these declared functions has a single dimensional parameter `d`. Function `index` takes no named parameters, while `first`, `prev`, and `next` take one named parameter each, and `fby` takes two named parameters.

6.4.8 Factorial: version two

If we wish to write factorial as a function, we write it as taking a value parameter.

```
fun fact.n = f
where
  dim d <- n ;;
  var f = fby.d 1 (index.d * f) ;;
end ;;
```

It uses a local dimension `d`, which is initially set to `n`. The stream `f` varies in dimension `d`. Note that `index.d` increments dimension `d`, while the second argument of `fby.d` decrements dimension `d`, so the two cancel each other out, yielding `#!d`.

6.4.9 Ackermann: version two

Ackermann takes two value parameters, and is defined using two local dimensions.

```
fun ack.m.n = f
where
  dim d_m <- m
  dim d_n <- n
  var f = fby.d_m (index.d_n)
                  (fby.d_n (next.d_n f) (next.d_m f))
end ;;
```

Note the elimination of explicit manipulation of dimensions.

6.4.10 Sieve of Eratosthenes

The sieve of Erastosthenes generates a stream in dimension `d` of the prime numbers. It is built using a local dimension $\mathtt{d_p}$. The first line is the naturals $\geqslant 2$, and subsequent lines are the previous line without the multiples of the first element of the previous line.

```
fun sieve.d = f
where
  dim d_p <- 0 ;;
  var f = fby.d (#!d_p + 2)
```

```
                (wvr.dp f (f % (first.dp f) != 0)) ;;
  fun wvr.d  X  Y = if first.d Y
                    then fby.d X (wvr.d (next.d X) (next.d Y))
                    else wvr.d (next.d X) (next.d Y) fi ;;
end ;;
```

The function `wvr.d` is a *filter* in the `d` dimension. It returns a stream in the `d` dimension that retains elements of the `X` input when the corresponding `Y` element is true.
The sequence of primes is formed by the first column.

		dim d_p →								
	f	0	1	2	3	4	5	6	7	···
dim d ↓	0	2	3	4	5	6	7	8	9	···
	1	3	5	7	9	11	13	15	17	···
	2	5	7	11	13	17	19	23	25	···
	3	7	11	13	17	19	23	29	31	···
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Each row corresponds to the removal of the multiples of the first element in the previous row.

6.4.11 Matrix multiplication

Here is the complete matrix multiplication example.

```
fun multiply.dr.dc.k X Y = W
where
  dim d <- 0 ;;
  var Xp = rotate.dc.d X ;;
  var Yp = rotate.dr.d Y ;;
  var Z = Xp * Yp ;;
  var W = sum.d.k Z ;;

  fun rotate.d1.d2 X = X @ [d1 <- #!d2] ;;
  fun sum.dx.n X = Y @ [dx <- n-1]
  where
    var Y = fby.dx X (Y + next.dx X) ;;
  end ;;
end ;;
```

Function `rotate.`d_1`.`d_2 changes variance in dimension d_1 to variance in dimension d_2. Function `sum.`d_x`.n X` adds up the first `n` elements in direction d_x of stream `X`.

6.4.12 Taylor series expansion

Our last example is to compute the Taylor series expansion for a function f around point a for point x.

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^{\mathrm{n}}$$

Function *taylor* takes as input a stream *derivs* of derivatives of f at point a in direction d.

```
fun taylor.d.a.x derivs = F
where
  var F = sum.d.(index.d) G ;;
  var G = derivs / (fact.(#!d)) * (pow.(#!d).(x-a)) ;;
```

```
  fun pow.n = g
  where
    dim d_p <- n ;;
    var g = fby.d_p (\s -> 1) (\s -> s * g.s) ;;
  end ;;
end ;;
```

The local function `pow.n` returns a function, namely the `n`-th–power function. Its definition uses a local dimension $\mathtt{d_p}$, and `g` is defined to be a stream of the powers of `s`.

The Taylor series expansion for the sine function around integral multiples of 2π yields:

```
taylor.d.a.x sinderivs
where
  var sinderivs = fby.d 0 (fby.d 1 (fby.d 0 (fby.d ~1 sinderivs))) ;;
end ;;
```

Chapter 7

Equations and Bestfitting

The `var` and `fun` declarators were introduced in Chapter 5 and further developed in Chapter 6. In this chapter, we present how having multiple declarations for the same variable or same function in understood by the TransLucid interpreter.

7.1 Variables

A variable is defined by one or more equations, each guarded by the context in which it is valid. The syntax for a variable declaration, adding one equation, is:

$$\begin{aligned} \textit{vardecl} &::= \texttt{var}\ \textit{id}\ \textit{guard}\ \texttt{=}\ E\ \texttt{;;} \\ \textit{guard} &::= \textit{regionGuard}^{?}\ \textit{booleanGuard}^{?} \\ \textit{regionGuard} &::= \texttt{[}\ E\ \texttt{:}E\ \texttt{,}\ \cdots\ \texttt{]} \\ \textit{booleanGuard} &::= \texttt{|}\ E \end{aligned}$$

For a given variable *id*, there can be many declarations. What distingues them is the guards, which define the context in which the relevant equation is applicable.

Each *guard* may contain a *regionGuard* component and a *booleanGuard* component.

- The *regionGuard* is a tuple in which each dimension is separated by its ordinate by a colon (`:`); a region guard is valid if the current context is situated inside the geometric region defined by the guard.
- The *booleanGuard*, introduced by an `|` (read "such that"), must be a Boolean expression, evaluating either to `true` or `false`. The Boolean guard is only evaluated if the *regionGuard* is valid.

Bestfitting is the process by which the most applicable equation of a variable is chosen from a set of guarded equations. The most applicable equation from a set of definitions, is the one that is valid in the smallest region of space, and is completely contained within all of the other applicable definitions.

When a variable *id* is to be evaluated in a particular context κ, the guards of all declarations for *id* are examined. The declarations whose guards are valid for context κ are deemed to be *applicable*. Among the applicable declarations, should the region guard for A define a region strictly contained in the region guard for B, then A is said to *refine* B; those declarations that are refined by other declarations are removed, leaving those declarations with no refinement: these are called the *bestfit* declarations.

Should there be exactly one bestfit declaration for *id*, then the expression to the right of the `=` is evaluated in that context, and the result is returned. Should there be more than one bestfit

declaration for *id*, the default behavior is to return a `spmultidef` value, meaning that the identifier is multiply defined.

For example, given the following definitions of the variable `foo`:

```
foo [a : 0..10, b : 0..10] = 0;;
foo [a : 5, b : 5..7] = 42;;
```

in the context `[a <- 5, b <- 5]`, the second will be chosen.

To the right of a dimension in the region guard, one can place

- an exact value,
- an atomic type,
- a range,
- an identifier declared as a type in a `data` declaration,
- an identifier declared as a constructor in a `data` declaration,
- or a predicate (a unary call-by-value function returning Booleans), effectively defining subtypes.

It is possible to change the default behavior of how bestfits are dealt with, on a variable-by-variable basis. This is done with the `bestselect` declaration:

$$\textit{bestdecl} ::= \texttt{bestselect}\ \textit{id}\ \texttt{=}\ \textit{op}\ \texttt{;;}$$

Its reading is that the bestfitter selector for *id* is defined using operator *op*, which must be implemented as an associative and commutative binary operator.

7.2 Functions

As for variables, there can be many declarations for the same function. The syntax for a function declaration is:

$$\textit{fundecl} ::= \texttt{fun}\ \textit{id}\ \textit{param}^{+}\ \textit{guard}\ \texttt{=}\ E\ \texttt{;;}$$
$$\textit{param} ::= \texttt{.}^{?}\ \textit{id}$$

The function parameters can be either call-by-name or call-by-value parameters, and are written in the same style as the applications thereof, with a space or a period respectively. The guard is written in the same manner as for variables, except that the parameters may also be used as dimensions in the region guard.

For example, to declare a function `f` with two call-by-value parameters `a` and `b`, which must both be integers, one can write:

```
fun f.a.b [a : intmp, b : intmp] = E1 ;;
```

where $\texttt{E}_1$ is the expression which defines the value of the function.

After a function has been declared, further definitions of the same function with different bestfit guards can be added. The parameter names can be left off, in which case they will be the same as in the first definition. For example, to add another definition of `f` where `a` and `b` are strings, write:

```
fun f [a : ustring, b : ustring] = E2 ;;
```

7.3 Time

Since a TransLucid interpreter is a reactive system, with clearly defined and separate instants enumerated by the `time` dimension, it is possible to modify the set of equations, and the meaning of variables, over time. We describe here how this can be done.

The first thing to note is that at any given instant n, a new equation for identifier *id* can be added. This is recorded in the TransLucid system using what we call a *provenance dimension*, where provenance means "where it comes from". Hence the provenance tag for this equation is `[time_provenance <- n]`. (Note that we envisage adding more provenance tags in the future, but for the moment, only time is considered.)

The second thing to note is that the `time` dimension can be used in the region guard of the equation, thereby limiting the time validity of this equation. It should be noted, however, that one cannot "change the past", i.e., prior instants in the time validity range are ignored. It is also possible to state that the time validity is just for the current instant, using the special identifier `now`. For example, if the current instant is 23

```
var x [time:now]    = 42 ;;
var y [time:22..24] = 43 ;;
var z [time:24..26] = 44 ;;
var w = 45 ;;
```

defines

- variable `x` to have value 42 just in instant 23;
- variable `y` to have value 43 in instants 23 and 24, i.e., the current instant and the next, not affecting instant 22;
- variable `z` to have value 44 in instants 24 through 26, i.e., for the three next future instants;
- variable `w` to have value 45 from instant 23 on.

Bestfitting with respect to time initially takes place ignoring the time provenance information. This means, of course, that if the `time` dimension appears in some of the region guards, that it will affect the bestfitting process.

Once a set of bestfit declarations is selected using the initial algorithm, if there are several of these, then only the ones with the greatest ordinate for the `time_provenance` dimension are retained. In other words, more recent declarations have priority over older declarations.

7.4 Priority

It is possible to prioritize the declarations using a *priority dimension*. Currently, the TransLucid interpreter recognizes one, called `priority`, whose ordinate must be an unsigned integer. (Note that a range could be used, meaning that a declaration is valid for a range of priorities.) If this dimension appears inside the region guard, then it affects the bestfitting process.

In the presence of a set of declarations of differing priorities, the bestfitter first considers the set of declarations with the highest priority. If bestfitting yields nothing, then the next higher priority is considered. This process is repeated until the bestfitter finds something—which includes multiple definitions producing a `spmultidef`—or until there are no more declarations to be considered.

7.5 UUID

Each declaration in TransLucid is assigned a Universal Unique Identifier (UUID), a 128-bit number usually printed as a 32-hex-digit number (with lower-case `a` through `f`). When the interpreter is in `--uuid` mode, the user is informed of the UUIDs that have been assigned to each declaration. As a result, it becomes possible for these to be edited directly.

Two new declarations are introduced.

$$\textit{replDecl} ::= \texttt{repl}\ \textit{uuid}\ \textit{decl}$$
$$\textit{delDecl} ::= \texttt{del}\ \textit{uuid}\ \texttt{;;}$$

In both cases, the `uuid` must correspond to an existing declaration. In the first case, the existing declaration is replaced, *from this instant on* (but not affecting previous instants) with the new declaration *decl*, which must be consistent with the previous one (variables to variables, functions of m parameters to functions of m parameters, etc.) In the second case, the existing declaration is deleted, once again, from this instant on and without affecting previous instants.

This mechanism will probably be more useful when using the `libtl` library directly, perhaps through a browser-editor interface, rather than with `tltext`.

Chapter 8

Hyperdatons

Hyperdatons are physical (as opposed to virtual) multidimensional data structures that are used by the TransLucid interpreter to read data from and write data to the external world, possibly other TransLucid interpreters. This part of the interpreter is only partially developed, with a limited set of possible interactions. The area is under current development.

As we saw in Chapter 5, a hyperdaton is introduced by a `hd` declaration:

$$\begin{aligned} \textit{hddecl} &::= \texttt{hd}\ \textit{id}\ \textit{guard}\ \texttt{=}\ \textit{hdarg}\ \texttt{;;} \\ \textit{hdarg} &::= \texttt{HdIn.}\textit{string} \\ &\mid \texttt{HdOut.}\textit{string} \end{aligned}$$

The strings passed as arguments to `HdIn` and `HdOut` must decribe the protocol to be used for reading or writing data. Then, to write the data, the `assign` declaration must be used to inform the interpreter how the output hyperdaton is to be filled in.

For example, suppose file `input.hd` looks like:

```
indexedby {0};;
entries = {0,1,2,3,4,5};;
```

Then the following program reads from file `input.hd` and copies its data into file `output.hd`.

```
hd input                  = HdIn."file_array_in_hd:input.hd" ;;
hd output [0 : 0..5]      = HdOut."file_array_out_hd:output.hd" ;;
assign output [0 : 0..5] := input;;
```

8.1 File hyperdatons

Currently, the only hyperdatons implemented are file hyperdatons, which are text files describing the data inside the hyperdaton. The grammar of such a file is given below:

$$\begin{aligned} \textit{file} &::= \textit{indexingDescription}\ \texttt{;;} \\ &\quad\ \textit{entriesDescription}\ \texttt{;;} \\ \textit{indexingDescription} &::= \texttt{indexedby}\ \textit{expr-list} \\ \textit{entriesDescription} &::= \texttt{entries =}\ \textit{array-init} \\ \textit{expr-list} &::= \texttt{\{}\ E\ (\texttt{,}\ E)^*\ \texttt{\}} \\ \textit{arrayInit} &::= \texttt{\{}\ \textit{arrayEntry}\ (\texttt{,}\ \textit{arrayEntry})^*\ \texttt{\}} \\ \textit{arrayEntry} &::= E \mid \textit{arrayInit} \end{aligned}$$

The number of indexes must agree with the dimensionality of the array initializer. Expressions can only appear in the last dimension of the initializer.

For example, a one-dimensional hyperdaton varying in dimension 0 could look like:

```
indexedby {0};;
entries = {1, 2, 3, 4};;
```

A two-dimensional hyperdaton varying in dimensions 0 and 1 could look like:

```
indexedby {0, 1};;
entries =
{
  {1, 2, 3, 4},
  {5, 6, 7, 8},
  {9, 10, 11, 12},
  {13, 14, 15, 16}
}
;;
```