KiCS2

The Kiel Curry System (Version 2)

User Manual

Version 3.0.0 of 2022-09-09

Michael Hanus[1] [editor]

Additional Contributors:

Bernd Braßel[2]
Björn Peemöller[3]
Fabian Reck[4]
Jan Rasmus Tikovsky[5]
Finn Teegen[6]

(1) University of Kiel, Germany, `mh@informatik.uni-kiel.de`
(2) University of Kiel, Germany, `bbr@informatik.uni-kiel.de`
(3) University of Kiel, Germany, `bjp@informatik.uni-kiel.de`
(4) University of Kiel, Germany, `fre@informatik.uni-kiel.de`
(5) University of Kiel, Germany, `jrt@informatik.uni-kiel.de`
(6) University of Kiel, Germany, `fte@informatik.uni-kiel.de`

Contents

Preface

This document describes KiCS2 (**Ki**el **C**urry **S**ystem Version **2**), an implementation of the multi-paradigm language Curry [16, 28] that is based on compiling Curry programs into Haskell programs. Curry is a universal programming language aiming at the amalgamation of the most important declarative programming paradigms, namely functional programming and logic programming. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). The current KiCS2 implementation does not support concurrent constraints. Alternatively, one can write distributed applications by the use of sockets that can be registered and accessed with symbolic names. Moreover, KiCS2 also supports the high-level implementation of graphical user interfaces and web services (as described in more detail in [17, 18, 19, 22]).

We assume familiarity with the ideas and features of Curry as described in the Curry language definition [28]. Therefore, this document only explains the use of the different components of KiCS2 and the differences and restrictions of KiCS2 (see Section 1.3) compared with the language Curry (Version 0.9.0). The basic ideas of the implementation of KiCS2 can be found in [12, 13].

Important Note

This version of KiCS2 implements **type classes**. The concept of type classes is not yet part of the Curry language report. The recognized syntax of type classes is specified in Section 4. Although the implemented concept of type classes is not fully described in this manual, it is quite similar to Haskell 98 [32] so that one can look there to find a detailed description.

Acknowledgements

This work has been supported in part by the DFG grants Ha 2457/5-1 and Ha 2457/5-2.

1 Overview of KiCS2

1.1 Installation

This version of KiCS2 has been developed and tested on Linux systems. In principle, it should be also executable on other platforms on which a Haskell implementation (Glasgow Haskell Compiler and Cabal) exists, like in many Linux distributions, Sun Solaris, or Mac OS X systems.

Installation instructions for KiCS2 can be found in the file `INSTALL.txt` stored in the KiCS2 installation directory. Note that KiCS2 can be installed with experimental support for profiling of executables. To use profiling, two requirements have to be met:

- The libraries that are shipped with the GHC that is used by KiCS2 have to be installed with profiling enabled. This is the default for the system libraries contained in the GHC release, but may not be the case for additional libraries.
- The `Makefile` of KiCS2 contains an option `PROFILING` which has to be set to `yes` to enable profiling support. You may either change the Makefile to

  ```
  PROFILING = yes
  ```

  or specify this setting while starting the installation process using

  ```
  make <optional target> PROFILING=yes
  ```

In the following, *kics2home* denotes the installation directory of the KiCS2 installation.

1.2 General Use

All executables required to use the different components of KiCS2 are stored in the directory *kics2home*`/bin`. You should add this directory to your path (e.g., by the `bash` command "`export PATH=`*kics2home*`/bin:$PATH`").

The source code of the Curry program must be stored in a file with the suffix "`.curry`", e.g., `prog.curry`. Literate programs must be stored in files with the extension "`.lcurry`".

Since the translation of Curry programs with KiCS2 creates some auxiliary files (see Section C for details), you need write permission in the directory where you have stored your Curry programs. The auxiliary files for all Curry programs in the current directory can be deleted by the command

```
cleancurry
```

(this is a shell script stored in the `bin` directory of the KiCS2 installation, see above). The command

```
cleancurry -r
```

also deletes the auxiliary files in all subdirectories.

1.3 Restrictions

There are a few minor restrictions on Curry programs when they are processed with KiCS2:

- *Singleton pattern variables*, i.e., variables that occur only once in a rule, should be denoted as an anonymous variable "`_`", otherwise the parser will print a warning since this is a typical source of programming errors.

- KiCS2 translates all *local declarations* into global functions with additional arguments ("lambda lifting", see Appendix D of the Curry language report). Thus, in the various run-time systems, the definition of functions with local declarations look different from their original definition (in order to see the result of this transformation, you can use the CurryBrowser, see Section 8).

- Tabulator stops instead of blank spaces in source files are interpreted as stops at columns 9, 17, 25, 33, and so on. In general, tabulator stops should be avoided in source programs.

- Encapsulated search: The general definition of encapsulated search of the Curry report [26] is not supported. Thus, the corresponding prelude operations like `try`, `solveAll`, `once`, `findall`, or `best` are not defined in the KiCS2 prelude. However, KiCS2 supports appropriate alternatives to encapsulate non-deterministic computations:

  Strong encapsulation: This means that all potential non-determinism is encapsulated. Since this might result in dependencies on the evaluation strategy (see [10] for a detailed discussion), this kind of encapsulation is only available as I/O operations. For instance, the library `Control.AllSolutions` (available in package `searchtree`) defines the operation

  ```
  getAllValues :: a → IO [a]
  ```

  to compute all values of a given argument expression. There is also the library `Control.SearchTree` (also available in package `searchtree`) which supports user-programmable search strategies and contains some predefined strategies like depth-first, breadth-first, iterative deepening search.

  Weak encapsulation: This means that only the non-determinism defined inside an encapsulation operator is encapsulated. Conceptually, these operators are offered as *set functions* [2] which compute the set of all results but do not encapsulate non-determinism in the actual arguments. See the library `SetFunctions` (available in package `setfunctions`) for more details.

- Concurrent computations based on the suspension of expressions containing free variables are not yet supported. KiCS2 supports *value generators* for free variables so that a free variable is instantiated when its value is demanded. For instance, the initial expression

  ```
  x == True where x free
  ```

  is non-deterministically evaluated to `False` and `True` by instantiating `x` to `False` and `True`, respectively. Thus, a computation is never suspended due to free variables. This behavior also applies to free variables of primitive types like integers. For instance, the initial expression

  ```
  x*y=:=1 where x,y free
  ```

  is non-deterministically evaluated to the two solutions

  ```
  {x = -1, y = -1} True
  {x = 1, y = 1} True
  ```

- Unification is performed without an occur check.
- There is currently no general connection to external constraint solvers.

1.4 Modules in KiCS2

KiCS2 searches for imported modules in various directories. By default, imported modules are searched in the directory of the main program and the system module directory "`kics2home/lib`". This search path can be extended by setting the environment variable `CURRYPATH` (which can be also set in a KiCS2 session by the option "`:set path`", see below) to a list of directory names separated by colons ("`:`"). In addition, a local standard search path can be defined in the "`.kics2rc`" file (see Section 2.7). Thus, modules to be loaded are searched in the following directories (in this order, i.e., the first occurrence of a module file in this search path is imported):

1. Current working directory ("`.`") or directory prefix of the main module (e.g., directory "`/home/joe/curryprogs`" if one loads the Curry program "`/home/joe/curryprogs/main`").
2. The directories enumerated in the environment variable `CURRYPATH`.
3. The directories enumerated in the "`.kics2rc`" variable "`libraries`".
4. The directory "`kics2home/lib`".

The same strategy also applies to modules with a hierarchical module name with the only difference that the hierarchy prefix of a module name corresponds to a directory prefix of the module. For instance, if the main module is stored in directory `MAINDIR` and imports the module `Test.Func`, then the module stored in `MAINDIR/Test/Func.curry` is imported (without setting any additional import path) according to the module search strategy described above.

Note that the standard prelude (`kics2home/lib/Prelude.curry`) will be always implicitly imported to all modules if a module does not contain an explicit import declaration for the module `Prelude`.

2 Using the Interactive Environment of KiCS2

This section describes the interactive environment KiCS2 that supports the development of applications written in Curry. The implementation of KiCS2 contains also a separate compiler which is automatically invoked by the interactive environment.

2.1 Invoking KiCS2

To start KiCS2, execute the command "`kics2`" or "`curry`" (these are shell scripts stored in *kics2home*`/bin` where *kics2home* is the installation directory of KiCS2). When the system is ready (i.e., when the prompt "`Prelude>`" occurs), the prelude (*kics2home*`/lib/Prelude.curry`) is already loaded, i.e., all definitions in the prelude are accessible. Now you can type various commands (see next section) or an expression to be evaluated.

One can also invoke KiCS2 with parameters. These parameters are usual a sequence of commands (see next section) that are executed before the user interaction starts. For instance, the invocation

```
kics2 :load Mod :add List
```

starts KiCS2, loads the main module `Mod`, and adds the additional module `List`. The invocation

```
kics2 :load Mod :eval config
```

starts KiCS2, loads the main module `Mod`, and evaluates the operation `config` before the user interaction starts. As a final example, the invocation

```
kics2 :load Mod :save :quit
```

starts KiCS2, loads the main module `Mod`, creates an executable, and terminates KiCS2. This invocation could be useful in "make" files for systems implemented in Curry.

There are also some specific options that can be used when invoking KiCS2:

`-h` **or** `--help` : Print only a help message.

`-V` **or** `--version` : Print the version information of KiCS2 and quit.

`--compiler-name` : Print just the compiler name (`kics2`) and quit.

`--numeric-version` : Print just the version number and quit.

`--noreadline` (if used, this must be the first option): Do not use input line editing (see Section 2.6).

`-Dname=val` (these options must come before any KiCS2 command): Overwrite values defined in the configuration file "`.kics2rc`" (see Section 2.7), where `name` is a property defined in the configuration file and `val` its new value.

One can also invoke KiCS2 with some run-time arguments that can be accessed inside a Curry program by the I/O operation `getArgs` (see library `System.Environment`, Section A.2.18). These run-time arguments must be written at the end after the separator "`--`". For instance, if KiCS2 is invoked by

```
kics2 :load Mod -- first and second
```

then a call to the I/O operation `getArgs` returns the list value

```
["first","and","second"]
```

2.2 Commands of KiCS2

The **most important commands** of KiCS2 are (it is sufficient to type a unique prefix of a command if it is unique, e.g., one can type "`:r`" instead of "`:reload`"):

`:help` Show a list of all available commands.

`:load` *prog* Compile and load the program stored in *prog*`.curry` together with all its imported modules.

`:reload` Recompile all currently loaded modules.

`:compile` *prog* Compile and load the program stored in *prog*`.curry`, as with the command "`:load` *prog*", but compile also the generated Haskell modules. Usually, this is automatically done when an expression is evaluated. Hence, it is not necessary to use this command.

`:add` $m_1 \ldots m_n$ Add modules $m_1, \ldots, m_n$ to the set of currently loaded modules so that their exported entities are available in the top-level environment.

expr Evaluate the expression *expr* to normal form and show the computed results. In the default mode, all results of non-deterministic computations are printed. One can also print first one result and the next result only if the user requests it. This behavior can be set by the option `interactive` (see below).

Free variables in initial expressions must be declared as in Curry programs. In order to see the results of their bindings,[1] they must be introduced by a "`where...free`" declaration. For instance, one can write

```
not b where b free
```

in order to obtain the following bindings and results:

```
{b = False} True
{b = True} False
```

Without these declarations, an error is reported in order to avoid the unintended introduction of free variables in initial expressions by typos.

If the free variables in the initial goal are of a polymorphic type, as in the expression

```
xs++ys=:=[z] where xs,ys,z free
```

they are specialized to the type "`()`" (since the current implementation of KiCS2 does not support computations with polymorphic logic variables).

[1] Currently, bindings are only printed if the initial expression is not an I/O action (i.e., not of type "`IO...`") and there are not more than ten free variables in the initial expression.

$\boxed{\textbf{:eval } \textit{expr}}$ Same as *expr*. This command might be useful when putting commands as arguments when invoking `kics2`.

`:quit` Exit the system.

There are also a number of **further commands** that are often useful:

`:type` *expr* Show the type of the expression *expr*.

`:programs` Show the list of all Curry programs that are available in the load path.

`:cd` *dir* Change the current working directory to *dir*.

`:edit` Load the source code of the current main module into a text editor. If the variable `editcommand` is set in the configuration file "`.kics2rc`" (see Section 2.7), its value is used as an editor command, otherwise the environment variable "`EDITOR`" is used as the editor program.

`:edit` *m* Load the source text of module *m* (which must be accessible via the current load path if no path specification is given) into a text editor which is defined as in the command "`:edit`".

`:show` Show the source text of the currently loaded Curry program. If the variable `showcommand` is set in the configuration file "`.kics2rc`" (see Section 2.7), its value is used as a command to show the source text, otherwise the environment variable `PAGER` (or "`cat`", if `PAGER` is undefined) is used to show the source text.

`:show` *m* Show the source text of module *m* which must be accessible via the current load path if no path specification is given.

`:source` *f* Show the source code of function *f* (which must be visible in the currently loaded module) in a separate window.

`:source` *m.f* Show the source code of function *f* defined in module *m* in a separate window.

`:browse` Start the CurryBrowser to analyze the currently loaded module together with all its imported modules (see Section 8 for more details).

`:interface` Show the interface of the currently loaded module, i.e., show the names of all imported modules, the fixity declarations of all exported operators, the exported datatypes declarations and the types of all exported functions.

`:interface` *m* Similar to "`:interface`" but shows the interface of the module *m* which must be in the load path of KiCS2.

`:usedimports` Show all calls to imported functions in the currently loaded module. This might be useful to see which import declarations are really necessary.

`:set` *option* Set or turn on/off a specific option of the KiCS2 environment (see 2.3 for a description of all options). Options are turned on by the prefix "`+`" and off by the prefix "`-`". Options that can only be set (e.g., `path`) must not contain a prefix.

`:set` Show a help text on the possible options together with the current values of all options.

`:save` Save the currently loaded program as an executable evaluating the main expression "`main`". The executable is stored in the file `Mod` if `Mod` is the name of the currently loaded main module.

`:save` *expr* Similar as "`:save`" but the expression *expr* (typically: a call to the main function) will be evaluated by the executable.

`:fork` *expr* The expression *expr*, which is typically of type "`IO ()`", is evaluated in an independent process which runs in parallel to the current KiCS2 process. All output and error messages from this new process are suppressed. This command is useful to test distributed Curry programs where one can start a new server process by this command. The new process will be terminated when the evaluation of the expression *expr* is finished.

`:!`*cmd* Shell escape: execute *cmd* in a Unix shell.

2.3 Options of KiCS2

The following options (which can be set by the command "`:set`") are currently supported:

`path` *path* Set the additional search path for loading modules to *path*. Note that this search path is only used for loading modules inside this invocation of KiCS2.

The path is a list of directories separated by "`:`". The prefix "`~`" is replaced by the home directory as in the following example:

```
:set path aux:~/tests
```

Relative directory names are replaced by absolute ones so that the path is independent of later changes of the current working directory.

`bfs` Set the search mode to evaluate non-deterministic expressions to breadth-first search. This is the default search strategy. Usually, all non-deterministic values are enumerated and printed with a breadth-first strategy, but one can also print only the first value or all values by interactively requesting them (see below for these options).

`dfs` Similarly to `bfs` but use a depth-first search strategy to compute and print the values of the initial expression.

`ids` Similarly to `bfs` but use an iterative-deepening strategy to compute and print the values of the initial expression. The initial depth bound is 100 and the depth-bound is doubled after each iteration.

`ids` *n* Similarly to `ids` but use an initial depth bound of *n*.

`parallel` Similarly to `bfs` but use a parallel search strategy to compute and print the values of the initial expression. The system chooses an appropriate number of threads according the current number of available processors.

`parallel` *n* Similarly to `parallel` but use *n* parallel threads.

`prdfs` Set the search mode to evaluate non-deterministic expressions to primitive depth-first search. This is usually the fastest method to print *all* non-deterministic values. However, it does not support the evaluation of values by interactively requesting them.

`choices` n Show the internal choice structure (according to the implementation described in [12]) resulting from the complete evaluation of the main expression in a tree-like structure. This mode is only useful for debugging or understanding the implementation of non-deterministic evaluations used in KiCS2. If the optional argument n is provided, the tree is shown up to depth n.

`supply` i (only available in the local compilation mode, see below) Use implementation i as the identifier supply for choice structures (see [12] for a detailed explanation). Currently, the following values for i are supported:

- `integer`**:** Use unbounded integers as choice identifiers. This implementation is described in [12].
- `ghc` **(default):** Use a more sophisticated implementation of choice identifiers (based on the ideas described in [9]) provided by the Glasgow Haskell Compiler.
- `pureio`**:** Use IO references (i.e., memory cells) for choice identifiers. This is the most efficient implementation for top-level depth-first search but cannot be used for more sophisticated search methods like encapsulated search.
- `ioref`**:** Use a mixture of `ghc` and `pureio`. IO references are used for top-level depth-first search and `ghc` identifiers are used for encapsulated search methods.

`v`n Set the verbosity level to n. The following values are allowed for n:

- $n = 0$**:** Do not show any messages (except for errors).
- $n = 1$**:** Show only messages of the front-end, like loading of modules.
- $n = 2$**:** Show also messages of the back end, like compilation messages from the Haskell compiler.
- $n = 3$**:** Show also intermediate messages and commands of the compilation process.
- $n = 4$**:** Show also all intermediate results of the compilation process.

`prompt` p Sets the user prompt which is shown when KiCS2 is waiting for input. If the parameter p starts with a letter or a percent sign, the prompt is printed as the given parameter, where the sequence "`%s`" is expanded to the list of currently loaded modules and "`%%`" is expanded to a percent sign. If the prompt starts with a double quote, it is read as a string and, therefore, also supports the normal escape sequences that can occur in Curry programs. The default setting is

```
:set prompt "%s> "
```

`+/-interactive` Turn on/off the interactive mode. In the interactive mode, the next non-deterministic value is only computed when the user requests it. Thus, one has also the possibility to terminate the enumeration of all values after having seen some values.

`+/-first` Turn on/off the first-only mode. In the first-only mode, only the first value of the main expression is printed (instead of all values).

`+/-optimize` Turn on/off the optimization of the target program.

`+/-bindings` Turn on/off the binding mode. If the binding mode is on (default), then the bindings of the free variables of the initial expression are printed together with the result of the expression.

`+/-time` Turn on/off the time mode. If the time mode is on, the cpu time and the elapsed time of the computation is always printed together with the result of an evaluation.

`+/-trace` Turn on/off the trace mode. If the trace mode is on, it is possible to trace the sources of failing computations.

`+/-profile` (only available when configured during installation, see Section 1.1) Turn on/off the profile mode. If the profile mode is on, expressions as well as programs are compiled with GHC's profiling capabilities enabled. For expressions, evaluation will automatically generate a file `Main.prof` containing the profiling information of the evaluation. For compiled programs, the profiling has to be manually activated using runtime options when executed:

```
kics2 :set +profile :load MyProgram.curry :save :quit
./MyProgram +RTS -p -RTS [additional arguments]
```

`+/-local` Turn on/off the local compilation and linking mode. In the standard execution mode of KiCS2, the local mode is turned off so that compiled Curry programs are linked against pre-compiled libraries that are packaged with Cabal. This makes the compilation and linking process faster so that this is the preferred execution mode. However, one cannot change the system libraries and directly use these changed libraries. Therefore, for experimental purposes, one can use the local mode. In this mode, a compiled Curry program is linked against the current versions of the system libraries which might require their re-compilation. Hence, it is important for the local mode that KiCS2 is installed in a local directory where one has write access, otherwise this mode might not work.

Note that some options for experimenting with KiCS2 (like `supply` or `ghc`) are available only in the local compilation mode (since they require the recompilation of parts of the installed system).

`+/-ghci` Turn on/off the ghci mode. In the ghci mode, the initial goal is send to the interactive version of the Glasgow Haskell Compiler. This might result in a slower execution but in a faster startup time since the linker to create the main executable is not used.

`safe` Turn on the safe execution mode. In the safe execution mode, the initial goal is not allowed to be of type `IO` and the program should not import the module `Unsafe`. Furthermore, the allowed commands are `eval`, `load`, `quit`, and `reload`. This mode is useful to use KiCS2 in uncontrolled environments, like a computation service in a web page, where KiCS2 could be invoked by

```
kics2 :set safe
```

`parser` *opts* Define additional options passed to the KiCS2 front end, i.e., the parser program *kics2home*`/bin/kics2-frontend`. For instance, setting the option

```
:set parser -F --pgmF=transcurry
```

has the effect that each Curry module to be compiled is transformed by the preprocessor command `transcurry` into a new Curry program which is actually compiled.

`cmp` *opts* Define additional options passed to the KiCS2 compiler. For instance, setting the option

```
:set cmp -O 0
```

has the effect that all optimizations performed by the KiCS2 compiler are turned off.

`ghc` *opts* Define additional options passed to the Glasgow Haskell Compiler (GHC) when the generated Haskell programs are compiled. Many options necessary to compile Curry programs are already set (you can see them by setting the verbosity level to 2 or greater). One has to be careful when providing additional options. For instance, in the default global installation of KiCS2, libraries are pre-compiled so that inconsistencies might occur if compilation options might be changed.

It is safe to pass specific GHC linking options. For instance, to enforce the static linking of libraries in order to generate an executable (see command "`:save`") that can be executed in another environment, one could set the options

```
:set ghc -static -optl-static -optl-pthread
```

Other options are useful for experimental purposes, but those should be used only in the local compilation mode (see above) to avoid inconsistent target codes for different libraries. For instance, setting the option

```
:set ghc -DDISABLE_CS
```

has the effect that the constraint store used to enable an efficient access to complex bindings is disabled. Similarly,

```
:set ghc -DSTRICT_VAL_BIND
```

has the effect that expressions in a unification constraint (`=:=`) are always fully evaluated (instead of the evaluation to a head normal form only) before unifying both sides. Since these options influence the compilation of the run-time system, one should also enforce the recompilation of Haskell programs by the GHC option "`-fforce-recomp`", e.g., one should set

```
:set ghc -DDISABLE_CS -fforce-recomp
```

`rts` *opts* Define additional run-time options passed to the executable generated by the Glasgow Haskell Compiler, i.e., the parameters "`+RTS` *o* `-RTS`" are passed to the executable. For instance, setting the option

```
:set rts -H512m
```

has the effect that the minimum heap size is set to 512 megabytes.

$\boxed{\texttt{args}\ \textit{arguments}}$ Define run-time arguments passed to the executable generated by the Glasgow Haskell Compiler. For instance, setting the option

```
:set args first second
```

has the effect that the I/O operation `getArgs` (see library `System.Environment` (Section A.2.18) returns the value `["first","second"]`.

2.4 Source-File Options

If the evaluation of operations in some main module loaded into KiCS2 requires specific options, like an iterative-deepening search strategy, one can also put these options into the source code of this module in order to avoid setting these options every time when this module is loaded. Such **source-file options** must occur before the module header, i.e., before the first declaration (module header, imports, fixity declaration, defining rules, etc) occurring in the module. Each source file option must be in a line of the form

{-# KiCS2_OPTION *opt* #-}

where *opt* is an option that can occur in a "`:set`" command (compare Section 2.3). Such a line in the source code (which is a comment according to the syntax of Curry) has the effect that this option is set by the KiCS2 command "`:set` *opt*" whenever this module is loaded (not reloaded!) as a main module. For instance, if a module starts with the lines

```
{-# KiCS2_OPTION ids #-}
{-# KiCS2_OPTION +ghci #-}
{-# KiCS2_OPTION v2 #-}
module M where
...
```

then the load command "`:load M`" will also set the options for iterative deepening, using `ghci` and verbosity level 2.

2.5 Using KiCS2 in Batch Mode

Although KiCS2 is primarily designed as an interactive system, it can also be used to process data in batch mode. For example, consider a Curry program, say `myprocessor`, that reads argument strings from the command line and processes them. Suppose the entry point is a function called `just_doit` that takes no arguments. Such a processor can be invoked from the shell as follows:

```
> kics2 :set args string1 string2 :load myprocessor.curry :eval just_doit :quit
```

The "`:quit`" directive in necessary to avoid KiCS2 going into interactive mode after the excution of the expression being evaluated. The actual run-time arguments (`string1`, `string2`) are defined by setting the option `args` (see above).

Here is an example to use KiCS2 in this way:

```
> kics2 :set args Hello World :add System :eval "getArgs >>= putStrLn . unwords" :quit
Hello World
>
```

2.6 Command Line Editing

In order to have support for line editing or history functionality in the command line of KiCS2 (as often supported by the `readline` library), you should have the Unix command `rlwrap` installed on your local machine. If `rlwrap` is installed, it is used by KiCS2 if called on a terminal. If it should not be used (e.g., because it is executed in an editor with `readline` functionality), one can call KiCS2 with the parameter "`--noreadline`" (which must occur as the first parameter).

2.7 Customization

In order to customize the behavior of KiCS2 to your own preferences, there is a configuration file which is read by KiCS2 when it is invoked. When you start KiCS2 for the first time, a standard version of this configuration file is copied with the name "`.kics2rc`" into your home directory. The file contains definitions of various settings, e.g., about showing warnings, using Curry extensions, programs etc. After you have started KiCS2 for the first time, look into this file and adapt it to your own preferences.

2.8 Emacs Interface

Emacs is a powerful programmable editor suitable for program development. It is freely available for many platforms (see http://www.emacs.org). The distribution of KiCS2 contains also a special *Curry mode* that supports the development of Curry programs in the Emacs environment. This mode includes support for syntax highlighting, finding declarations in the current buffer, and loading Curry programs into KiCS2 in an Emacs shell.

The Curry mode has been adapted from a similar mode for Haskell programs. Its installation is described in the file `README` in directory "*kics2home*`/tools/emacs`" which also contains the sources of the Curry mode and a short description about the use of this mode.

3 Extensions

KiCS2 supports some extensions in Curry programs that are not (yet) part of the definition of Curry. These extensions are described below.

3.1 Narrowing on Int Literals

In addition to narrowing on algebraic data types, KiCS2 also implements narrowing on values of the primitive type `Int`. For example, the goal "`x == 3 where x free`" is evaluated to the solutions

```
Prelude> x == 3 where x free
{x = (-_x2)       } False
{x = 0            } False
{x = 1            } False
{x = (2 * _x3)    } False
{x = 3            } True
{x = (4 * _x4 + 1)} False
{x = (4 * _x4 + 3)} False
```

Note that the free variables occuring in the binding are restricted to positive numbers greater than 0 (the output has been indented to increase readability). This feature is implemented by an internal binary representation of integer numbers. If necessary, this representation can be exposed to the user by setting the flag `BinaryInt` during installation:

```
make [kernel|install] RUNTIMEFLAGS=BinaryInt
```

In an experimental (local) installation, the flag can also be set in the interpreter:

```
:set ghc -DBinaryInt
```

The example above will then be evaluated (without indentation) to:

```
Prelude> x == 3 where x free
{x = (Neg _x2)        } False
{x = 0                } False
{x = 1                } False
{x = (Pos (O _x3))    } False
{x = 3                } True
{x = (Pos (I (O _x4)))} False
{x = (Pos (I (I _x4)))} False
```

In this output, values without free variables are presented as before. For values containing a free variable, the constructors `Neg` and `Pos` denote negative and positive numbers (without 0), while the constructors `O` and `I` denote a 0– and 1–bit where the *least significant bit comes first*. That is, `(Pos (I (O _x4)))` $= +$`(I (O _x4))` $= +(2 *$ `(O _x4)`$) + 1 = +(4 *$ `_x4`$) + 1$ which meets the output above.

3.2 Recursive Variable Bindings

Local variable declarations (introduced by `let` or `where`) can be (mutually) recursive in KiCS2. For instance, the declaration

```
ones5 = let ones = 1 : ones
         in take 5 ones
```

introduces the local variable `ones` which is bound to a *cyclic structure* representing an infinite list of `1`'s. Similarly, the definition

```
onetwo n = take n one2
 where
   one2 = 1 : two1
   two1 = 2 : one2
```

introduces a local variables `one2` that represents an infinite list of alternating `1`'s and `2`'s so that the expression `(onetwo 6)` evaluates to `[1,2,1,2,1,2]`.

3.3 Functional Patterns

Functional patterns [1] are a useful extension to implement operations in a more readable way. Furthermore, defining operations with functional patterns avoids problems caused by strict equality ("`=:=`") and leads to programs that are potentially more efficient.

Consider the definition of an operation to compute the last element of a list `xs` based on the prelude operation "`++`" for list concatenation:

```
last xs | _ ++ [y] =:= xs = y   where y free
```

Since the equality constraint "`=:=`" evaluates both sides to a constructor term, all elements of the list `xs` are fully evaluated in order to satisfy the constraint.

Functional patterns can help to improve this computational behavior. A *functional pattern* is a function call at a pattern position. With functional patterns, we can define the operation `last` as follows:

```
last (_ ++ [y]) = y
```

This definition is not only more compact but also avoids the complete evaluation of the list elements: since a functional pattern is considered as an abbreviation for the set of constructor terms obtained by all evaluations of the functional pattern to normal form (see [1] for an exact definition), the previous definition is conceptually equivalent to the set of rules

```
last [y] = y
last [_,y] = y
last [_,_,y] = y
...
```

which shows that the evaluation of the list elements is not demanded by the functional pattern.

In general, a pattern of the form $(f\ t_1 \ldots t_n)$ for $n > 0$ (or of the qualified form $(M.f\ t_1 \ldots t_n)$ for $n \geq 0$) is interpreted as a functional pattern if f is not a visible constructor but a defined function that is visible in the scope of the pattern. Furthermore, for a functional pattern to be well defined, there are two additional requirements to be satisfied:

1. If a function f is defined by means of a functional pattern fp, then the evaluation of fp must not depend on f, i.e., the semantics of a function defined using functional patterns must not (transitively) depend on its own definition. This excludes definitions such as

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

and is necessary to assign a semantics to funtions employing functional patterns (see [1] for more details).

2. Only functions that are globally defined may occur inside a functional pattern. This restriction ensures that no local variable might occur in the value of a functional pattern, which might lead to an non-intuitive semantics. Consider, for instance, the following (complicated) equality operation

   ```
   eq :: a  →  a  →  Bool
   eq x y = h y
    where
     g True  = x
     h (g a) = a
   ```

   where the locally defined function `g` occurs in the functional pattern `(g a)` of `h`. Since `(g a)` evaluates to the value of `x` whereas `a` is instantiated to `True`, the call `h y` now evaluates to `True` if the value of `y` equals the value of `x`. In order to check this equality condition, a strict unification between `x` and `y` is required so that an equivalent definition without functional patterns would be:

   ```
   eq :: a  →  a  →  Bool
   eq x y = h y
    where
     h x1 | x =:= x1 = True
   ```

   However, this implies that variables occuring in the value of a functional pattern imply a strict unification if they are defined in an outer scope, whereas variables defined *inside* a functional pattern behave like pattern variables. In consequence, the occurrence of variables from an outer scope inside a functional pattern might lead to an non-intuitive behavior. To avoid such problems, locally defined functions are excluded as functional patterns. Note that this does not exclude a functional pattern inside a local function, which is still perfectly reasonable.

It is also possible to combine functional patterns with as-patterns. Similarly to the meaning of as-patterns in standard constructor patterns, as-patterns in functional patterns are interpreted as a sequence of pattern matching where the variable of the as-pattern is matched before the given pattern is matched. This process can be described by introducing an auxiliary operation for this two-level pattern matching process. For instance, the definition

```
f (_ ++ x@[(42,_)] ++ _) = x
```

is considered as syntactic sugar for the expanded definition

```
f (_ ++ x ++ _) = f' x
 where
  f' [(42,_)] = x
```

However, as-patterns are usually implemented in a more efficient way without introducing auxiliary operations.

3.4 Order of Pattern Matching

Curry allows multiple occurrences of pattern variables in standard patterns. These are an abbreviation of equational constraints between pattern variables. Functional patterns might also contain multiple occurrences of pattern variables. For instance, the operation

```
f (_ ++ [x] ++ _ ++ [x] ++ _) = x
```

returns all elements with at least two occurrences in a list.

If functional patterns as well as multiple occurrences of pattern variables occur in a pattern defining an operation, there are various orders to match an expression against such an operation. In the current implementation, the order is as follows:

1. Standard pattern matching: First, it is checked whether the constructor patterns match. Thus, functional patterns and multiple occurrences of pattern variables are ignored.

2. Functional pattern matching: In the next phase, functional patterns are matched but occurrences of standard pattern variables in the functional patterns are ignored.

3. Non-linear patterns: If standard and functional pattern matching is successful, the equational constraints which correspond to multiple occurrences pattern variables are solved.

4. Guards: Finally, the guards supplied by the programmer are checked.

The order of pattern matching should not influence the computed result. However, it might have some influence on the termination behavior of programs, i.e., a program might not terminate instead of finitely failing. In such cases, it could be necessary to consider the influence of the order of pattern matching. Note that other orders of pattern matching can be obtained using auxiliary operations.

3.5 Type Classes

The concept of type classes is not yet part of the Curry language report. The recognized syntax of type classes is specified in Section 4. Although the implemented concept of type classes is not fully described in this manual, it is quite similar to Haskell 98 [32] so that one can look there to find a detailed description.

3.6 Free Variables, Equality, and the Type Class `Data`

Curry extends purely functional programming languages, like Haskell, with built-in non-determinism and free variables. The value of a *free variable* is unknown when it is introduced. A free variable is instantiated to some value if it occurs as a demanded argument of an operation to be evaluated (or by unification, which can be considered as an optimization of evaluating an equality operator [6]). Since patterns occurring in program rules are built from variables and data constructors, free variables cannot be instantiated to values of a functional type. As a consequence, the type of a polymorphic variable should be restricted to non-functional types only.

Another potential problem when dealing with free variables and unification is the precise notion of equality. Since Curry is intended as an extension of Haskell, Curry supports the type class `Eq` with operations "`==`" and "`/=`". Although standard textbooks on Haskell define this operation as *equality*, its actual implementation can be different since, as a member of the type class `Eq`, it can be defined with a behavior different than equality on concrete type instances. Actually, the documentation of the type class `Eq`[2] denotes "`==`" as "equality" but also contains the remark: "`==` is customarily expected to implement an equivalence relationship where two values comparing equal are indistinguishable by "public" functions." Thus, it is intended that e_1 `==` e_2 evaluates to `True` even if e_1 and e_2 have not the same but only equivalent values.

For instance, consider a data type for values indexed by a unique number:

```
data IVal a = IVal Int a
```

If the index is assumed to be unique when `IVal` values are used, one might define the comparison of indexed values by just comparing the indices:

```
instance Eq a => Eq (IVal a) where
  IVal i1 _ == IVal i2 _  =  i1 == i2
```

With this definition, the prelude operation `elem` yields surprising results:

```
> elem (IVal 1 'b') [IVal 1 'a']
True
```

Such a result is not intended since the element (first argument) does not occur in the list.

As a further example, consider the functional logic definition of the operation `last` to compute the last element of a list:

```
last xs | _ ++ [e] == xs = e
  where e free
```

Since "`==`" denotes equivalence rather than equality, `last` might not return the last element of a list but one (or more than one) value which is equivalent to the last element. For instance, we get the following answer when computing the last element of a given `IVal` list:

```
> last [IVal 1 'a']
IVal 1 _
```

Hence, instead of the actual last element, we get a rather general representation of it where "`_`" denotes a free variable of type `Char`.

These problems are avoided in KiCS2 by the predefined type class `Data`, as proposed in [27]:

```
class Data a where
  (===)  :: a → a → Bool
  aValue :: a
```

The operation "`===`" implements strict equality (rather than an equivalence relation) on type `a`, i.e., e_1 `===` e_2 evaluates to `True` if both expressions e_1 and e_2 evaluate to some ground value v. The operation `aValue` non-deterministically returns all values of type `a`. In contrast to other type classes, `Data` is predefined so that the following holds:

1. It is not allowed to define explicit `Data` instances for particular types. This avoids the definition of unintended instances.

2. `Data` instances are automatically derived for all first-order types. A type is *first-order* if all its

[2] http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Eq.html

values do not contain functional components, i.e., all constructors have non-functional type arguments and refer to other first-order types only.

Thus, the prelude base types `Bool`, `Char`, `Int`, `Float`,[3] `Ordering` as well as type constructors like `Maybe`, `Either`, list and tuple constructors have `Data` instances. For instance, we can non-deterministically enumerate values by specifying the desired type instance for `aValue`:

```
> aValue :: Maybe Bool
Nothing
Just False
Just True
```

Moreover, free variables have the class constraint `Data` so that they cannot be used as unknown functional values. Hence, the definition of `last` shown above can be modified as follows to work as intended:

```
last :: Data a => [a]  → a
last xs | _ ++ [e] === xs = e
  where e free
```

The type signature implies that `last` cannot be applied to a list of functional values.

The unification operation "`=:=`" returns `True` if both arguments can be evaluated to unifiable data values. Thus, it can be considered as an optimization of "`===`" that can be used when only `True` should be computed, as in conditions of rules (see [6]). As a consequence, the type of "`=:=`" is identical to the type of "`===`":

```
(=:=) :: Data a => a  → a  → Bool
```

Hence, the operation `last` can also be defined by

```
last :: Data a => [a]  → a
last xs | _ ++ [e] =:= xs = e
  where e free
```

3.7 Parser Options in Source Programs

The front end of KiCS2 understands various options. These options can be passed to the front end by setting the KiCS2 option `parser`, e.g., by

```
:set parser -F --pgmF=transcurry
```

or for individual modules by providing an option line as a specific comment at the beginning of the source program. For instance, the option above can be set for a specific module by putting the line

```
{-# OPTIONS_FRONTEND -F --pgmF=transcurry #-}
```

at the beginning of the module. This is useful to switch off specific warnings when parsing a module. For instance,

```
{-# OPTIONS_FRONTEND -Wno-incomplete-patterns -Wno-overlapping #-}
```

[3]Since there is no reasonable value generator for floats, `aValue :: Float` returns a free variable.

suppresses warnings about incompletely defined operations and operations defined by overlapping rules.

Generally, the string following `OPTIONS_FRONTEND` will be split at white spaces and treated like an ordinary command line argument string passed to the front end. If one wishes to provide options containing spaces, e.g., directory paths, this can be achieved by quoting the respective argument using either single or double quotes. The list of all available options can be listed by the help command of the front end:

```
kics2home/bin/pakcs-frontend --help
```

Note that the following options are excluded:

- A change of the compilation targets (e.g., change from FlatCurry to AbstractCurry).
- A change of the import paths.
- A change of the library paths.

These options can only be set via the command line.

3.8 Conditional Compilation

KiCS2 also supports conditional compilation in the C preprocessor (CPP) style. Actually, only a subset of the C preprocessor is supported (see below), e.g., "includes" are not allowed. Although conditional compilation might cause problems and should be avoided, sometimes it is useful to support libraries across different Curry compilers with different features in their back ends.

To enable conditional compilation, the header of the program text should contain the line

```
{-# LANGUAGE CPP #-}
```

Then the source code might contain compilation directives like

```
#ifdef __KICS2__
eqChar external
#elif defined(__PAKCS__)
eqChar x y = (prim_eqChar $# y) $# x

prim_eqChar :: Char → Char → Bool
prim_eqChar external
#endif
```

Thus, if the front end is invoked with option

```
-D__PAKCS__=303
```

(which is automatically done by KiCS2 in version 3.3.x), the first three and the last lines are replaced by blank lines in the source code above before it is passed to the parser. Thus, the line numbers of the remaining code are not changed by preprocessing.

Each directive has to be written in a separate line and will be replaced by a blank line after processing it. In the following, we discuss the supported directives.

`#define` *id val*

In the subsequent source text following that directive, the identifier *id* is defined with value *val*. An identifier is a letter or an underscore followed by zero or more letters, underscores or digits. The value *val* consists of one or more digits.

`#undef` *id*

In the subsequent source text following that directive, the identifier *id* becomes undefined (regardless whether it was defined before).

`#if` *cond*

If the condition *cond* is true, then all lines between the subsequent matching `#else` or `#elif` and the corresponding `#endif` directive, if present, are replaced by blank lines. Otherwise, all lines up to the subsequent matching `#else`, `#elif`, or `#endif` directive, if present, are replaced by blank lines. Conditions have one of the following forms:

- *id op val*: If the comparison expression evaluates to true, this condition is true. The operator *op* is one of `==`, `/=`, `<`, `<=`, `>`, or `>=`. If the identifier used in the expression is not currently defined, it is assumed to have value `0`.
- `defined(`*id*`)`: If the identifier *id* is currently defined, then this condition is true.
- `!defined(`*id*`)`: If the identifier *id* is not currently defined, then this condition is true.

`#ifdef` *id*

This directive is equivalent to `#if defined(`*id*`)`.

`#ifndef` *id*

This directive is equivalent to `#if !defined(`*id*`)`.

`#else`

This directive marks the start of the lines which are kept if the preceding `#if` or `#elif` has a false condition.

`#elif` *cond*

This directive is interpreted as an `#else` followed by a new `#if`.

`#endif`

This directive terminates the preceding `#if`, `#else`, or `#elif` directive.

3.9 Language Pragmas

KiCS2 supports a couple of language pragmas to influence the kind of the source language to be processed. One such pragma, conditional compilation, has been described in the previous section. In this section we describe two pragmas which might be useful for experimental purposes.

The Curry prelude (library `Prelude`) contains many definition of standard data types, operations, and type classes and instances. Thus, it is a fairly large module. When developing new tools for analyzing or manipulating programs, the complexity of the prelude, which is imported by any simple program, hinders sometimes the initial development of such tools. For this purpose, it could be useful to compile a program without the prelude. This can be achieved by putting the following language pragma into the header of the module:

```
{-# LANGUAGE NoImplicitPrelude #-}
```

Note that such a module has to define all data types on which operations are defined, since nothing from the prelude is available in such a module.

As described in Section 3.6, instances of class `Data` are automatically derived by KiCS2. Since the implementation of these instances refer to the prelude and are sometimes complex, one can suppress the derivation of `Data` instances by the language pragma

```
{-# LANGUAGE NoDataDeriving #-}
```

Since the implementation of `Data` instances refer to operations defined in the prelude, it is not possible to derive such instances without the prelude. Therefore, the language pragma `NoImplicitPrelude` automatically implies the pragma `NoDataDeriving`.

For example, the compilation target of the following program contains two type declarations and two operations without any implicitly generated auxiliary operations:

```
{-# LANGUAGE NoImplicitPrelude #-}

data Nat = Z | S Nat

data MyBool = False | True

-- Addition on natural numbers.
add         :: Nat  → Nat  → Nat
add Z     n = n
add (S m) n = S (add m n)

-- Less-or-equal predicate on natural numbers.
leq :: Nat  → Nat  → MyBool
leq Z     _     = True
leq (S _) Z     = False
leq (S x) (S y) = leq x y
```

Note that it is not possible to use free variables in this program, since free variables require the type class constraint `Data` (see Section 3.6).

4 Recognized Syntax of Curry

The KiCS2 Curry compiler accepts a slightly extended version of the grammar specified in the Curry Report [28]. Furthermore, the syntax recognized by KiCS2 differs from that specified in the Curry Report regarding numeric or character literals. We therefore present the complete description of the syntax below, whereas syntactic extensions are highlighted.

4.1 Notational Conventions

The syntax is given in extended Backus-Naur-Form (eBNF), using the following notation:

NonTerm ::= α	production
NonTerm	nonterminal symbol
`Term`	terminal symbol
$[\alpha]$	optional
$\{\alpha\}$	zero or more repetitions
(α)	grouping
$\alpha \mid \beta$	alternative
$\alpha_{\langle\beta\rangle}$	difference – elements generated by α without those generated by β

The Curry files are expected to be encoded in UTF-8. However, source programs are biased towards ASCII for compatibility reasons.

4.2 Lexicon

4.2.1 Comments

Comments either begin with "`--`" and terminate at the end of the line, or begin with "`{-`" and terminate with a matching "`-}`", i.e., the delimiters "`{-`" and "`-}`" act as parentheses and can be nested.

4.2.2 Identifiers and Keywords

The case of identifiers is important, i.e., the identifier "abc" is different from "ABC". Although the Curry Report specifies four different case modes (Prolog, Gödel, Haskell, free), the KiCS2 only supports the *free* mode which puts no constraints on the case of identifiers in certain language constructs.

Letter ::=	any ASCII letter
Dashes ::=	`--` {`-`}
Ident ::=	(*Letter* {*Letter* | *Digit* | `_` | `'`})$_{\langle \mathit{ReservedID} \rangle}$
Symbol ::=	`~` | `!` | `@` | `#` | `$` | `%` | `^` | `&` | `*` | `+` | `-` | `=` | `<` | `>` | `?` | `.` | `/` | `|` | `\` | `:`
ModuleID ::=	{*Ident* `.`} *Ident*
TypeConstrID ::=	*Ident*
TypeVarID ::=	*Ident* | `_`
ClassVarID ::=	*Ident*

$$
\begin{aligned}
\textit{ExistVarID} &::= \textit{Ident} \\
\textit{DataConstrID} &::= \textit{Ident} \\
\textit{InfixOpID} &::= (\textit{Symbol}\ \{\textit{Symbol}\})_{\langle \textit{Dashes}\ |\ \textit{ReservedSym} \rangle} \\
\textit{FunctionID} &::= \textit{Ident} \\
\textit{VariableID} &::= \textit{Ident} \\
\textit{LabelID} &::= \textit{Ident} \\
\textit{ClassID} &::= \textit{Ident}
\end{aligned}
$$

$$
\begin{aligned}
\textit{QTypeConstrID} &::= [\textit{ModuleID}\ \texttt{.}]\ \textit{TypeConstrID} \\
\textit{QDataConstrID} &::= [\textit{ModuleID}\ \texttt{.}]\ \textit{DataConstrID} \\
\textit{QInfixOpID} &::= [\textit{ModuleID}\ \texttt{.}]\ \textit{InfixOpID} \\
\textit{QFunctionID} &::= [\textit{ModuleID}\ \texttt{.}]\ \textit{FunctionID} \\
\textit{QLabelID} &::= [\textit{ModuleID}\ \texttt{.}]\ \textit{LabelID} \\
\textit{QClassID} &::= [\textit{ModuleID}\ \texttt{.}]\ \textit{ClassID}
\end{aligned}
$$

The following identifiers are recognized as keywords and cannot be used as regular identifiers.

$$
\begin{aligned}
\textit{ReservedID} ::= {} & \texttt{case} \mid \texttt{class} \mid \texttt{data} \mid \texttt{default} \mid \texttt{deriving} \mid \texttt{do} \mid \texttt{else} \mid \texttt{external} \\
\mid {} & \texttt{fcase} \mid \texttt{free} \mid \texttt{if} \mid \texttt{import} \mid \texttt{in} \mid \texttt{infix} \mid \texttt{infixl} \mid \texttt{infixr} \\
\mid {} & \texttt{instance} \mid \texttt{let} \mid \texttt{module} \mid \texttt{newtype} \mid \texttt{of} \mid \texttt{then} \mid \texttt{type} \mid \texttt{where}
\end{aligned}
$$

Note that the identifiers `as`, `forall`, `hiding` and `qualified` are no keywords. They have only a special meaning in module headers and can thus be used as ordinary identifiers elsewhere. The following symbols also have a special meaning and cannot be used as an infix operator identifier.

$$
\textit{ReservedSym} ::= \texttt{..} \mid \texttt{:} \mid \texttt{::} \mid \texttt{=} \mid \texttt{\textbackslash} \mid \texttt{|} \mid \texttt{<-} \mid \texttt{->} \mid \texttt{@} \mid \texttt{\~{}} \mid \texttt{=>}
$$

4.2.3 Numeric and Character Literals

In contrast to the Curry Report, KiCS2 adopts Haskell's notation of literals for both numeric as well as character and string literals, extended with the ability to denote binary integer literals.

$$
\begin{aligned}
\textit{Int} ::= {} & \textit{Decimal} \\
\mid {} & \texttt{0b}\ \textit{Binary} \mid \texttt{0B}\ \textit{Binary} \\
\mid {} & \texttt{0o}\ \textit{Octal} \mid \texttt{0O}\ \textit{Octal} \\
\mid {} & \texttt{0x}\ \textit{Hexadecimal} \mid \texttt{0X}\ \textit{Hexadecimal}
\end{aligned}
$$

$$
\begin{aligned}
\textit{Float} ::= {} & \textit{Decimal}\ \texttt{.}\ \textit{Decimal}\ [\textit{Exponent}] \\
\mid {} & \textit{Decimal}\ \textit{Exponent} \\
\textit{Exponent} ::= {} & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \textit{Decimal}
\end{aligned}
$$

$$
\begin{aligned}
\textit{Decimal} &::= \textit{Digit}\ \{\textit{Digit}\} \\
\textit{Binary} &::= \textit{Binit}\ \{\textit{Binit}\} \\
\textit{Octal} &::= \textit{Octit}\ \{\textit{Octit}\} \\
\textit{Hexadecimal} &::= \textit{Hexit}\ \{\textit{Hexit}\}
\end{aligned}
$$

$$
\begin{aligned}
\textit{Digit} &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{Binit} &::= \texttt{0} \mid \texttt{1} \\
\textit{Octit} &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \\
\textit{Hexit} &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f}
\end{aligned}
$$

For character and string literals, the syntax is as follows:

$$
\begin{aligned}
\textit{Char} &::= \texttt{'}\ (\ \textit{Graphic}_{\langle \texttt{\textbackslash} \rangle} \mid \textit{Space} \mid \textit{Escape}_{\langle \texttt{\textbackslash\&} \rangle}\ )\ \texttt{'} \\
\textit{String} &::= \texttt{"}\ \{\ \textit{Graphic}_{\langle \texttt{"}\ |\ \texttt{\textbackslash} \rangle} \mid \textit{Space} \mid \textit{Escape} \mid \textit{Gap}\ \}\ \texttt{"}
\end{aligned}
$$

```
    Escape ::= \ ( CharEsc | AsciiEsc | Decimal | o Octal | x Hexadecimal )
   CharEsc ::= a | b | f | n | r | t | v | \ | " | ' | &
  AsciiEsc ::= ^ Cntrl | NUL | SOH | STX | ETX | EOT | ENQ | ACK
             |  BEL | BS | HT | LF | VT | FF | CR | SO | SI | DLE
             |  DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN
             |  EM | SUB | ESC | FS | GS | RS | US | SP | DEL
     Cntrl ::= A | ... | Z | @ | [ | \ | ] | ^ | _
       Gap ::= \ WhiteChar {WhiteChar} \
   Graphic ::= any graphical character
 WhiteChar ::= any whitespace character
```

4.3 Layout

Similarly to Haskell, a Curry programmer can use layout information to define the structure of blocks. For this purpose, we define the indentation of a symbol as the column number indicating the start of this symbol, and the indentation of a line is the indentation of its first symbol.[4]

The layout (or "off-side") rule applies to lists of syntactic entities after the keywords `let`, `where`, `do`, or `of`. In the subsequent context-free syntax, these lists are enclosed with curly braces (`{ }`) and the single entities are separated by semicolons (`;`). Instead of using the curly braces and semicolons of the context-free syntax, a Curry programmer can also specify these lists by indentation: the indentation of a list of syntactic entities after `let`, `where`, `do`, or `of` is the indentation of the next symbol following the `let`, `where`, `do`, `of`. Any item of this list starts with the same indentation as the list. Lines with only whitespaces or an indentation greater than the indentation of the list continue the item in the previous line. Lines with an indentation less than the indentation of the list terminate the entire list. Moreover, a list started by `let` is terminated by the keyword `in`. Thus, the sentence

```
f x = h x where { g y = y + 1 ; h z = (g z) * 2 }
```

which is valid w.r.t. the context-free syntax, can be written with the layout rules as

```
f x = h x
 where g y = y + 1
       h z = (g z) * 2
```

or also as

```
f x = h x  where
  g y = y + 1
  h z = (g z)
         * 2
```

To avoid an indentation of top-level declarations, the keyword `module` and the end-of-file token are assumed to start in column 0.

[4] In order to determine the exact column number, we assume a fixed-width font with tab stops at each 8th column.

4.4 Context-Free Grammar

Module ::= `module` *ModuleID* [*Exports*] `where` *Block*
| *Block*

Block ::= `{` [*ImportDecls* `;`] $BlockDecl_1$ `;` … `;` $BlockDecl_n$ `}` (*no fixity declarations here*, $n \geq 0$)

Exports ::= `(` $Export_1$ `,` … `,` $Export_n$ `)` ($n \geq 0$)
Export ::= *QFunction*
| *QTypeConstrID* [`(` $ConsLabel_1$ `,` … `,` $ConsLabel_n$ `)`] ($n \geq 0$)
| *QTypeConstrID* `(..)`
| *QClassID* [`(` $Function_1$ `,` … `,` $Function_n$ `)`] ($n \geq 0$)
| *QClassID* `(..)`
| `module` *ModuleID*
ConsLabel ::= *DataConstr* | *Label*

ImportDecls ::= $ImportDecl_1$ `;` … `;` $ImportDecl_n$ ($n \geq 1$)
ImportDecl ::= `import` [`qualified`] *ModuleID* [`as` *ModuleID*] [*ImportSpec*]
ImportSpec ::= `(` $Import_1$ `,` … `,` $Import_n$ `)` ($n \geq 0$)
| `hiding` `(` $Import_1$ `,` … `,` $Import_n$ `)` ($n \geq 0$)
Import ::= *Function*
| *TypeConstrID* [`(` $ConsLabel_1$ `,` … `,` $ConsLabel_n$ `)`] ($n \geq 0$)
| *TypeConstrID* `(..)`
| *ClassID* [`(` $Function_1$ `,` … `,` $Function_n$ `)`] ($n \geq 0$)
| *ClassID* `(..)`

BlockDecl ::= *TypeSynDecl*
| *DataDecl*
| *NewtypeDecl*
| *FixityDecl*
| *FunctionDecl*
| *DefaultDecl*
| *ClassDecl*
| *InstanceDecl*

TypeSynDecl ::= `type` *SimpleType* `=` *TypeExpr*
SimpleType ::= *TypeConstrID* $TypeVarID_1$ … $TypeVarID_n$ ($n \geq 0$)

DataDecl ::= `external` `data` *SimpleType* (*external data type*)
| `data` *SimpleType* [`=` *ConstrDecls*] [`deriving` *DerivingDecl*]
ConstrDecls ::= $ConstrDecl_1$ `|` … `|` $ConstrDecl_n$ ($n \geq 1$)
ConstrDecl ::= [*ExistVars*] [*Context* `=>`] *ConDecl*
ExistVars ::= `forall` $ExistVarID_1$ … $ExistVarID_n$ `.` ($n \geq 1$)
ConDecl ::= *DataConstr* $SimpleTypeExpr_1$ … $SimpleTypeExpr_n$ ($n \geq 0$)
| *TypeAppExpr* *ConOp* *TypeAppExpr* (*infix data constructor*)
| *DataConstr* `{` $FieldDecl_1$ `,` … `,` $FieldDecl_n$ `}` ($n \geq 0$)
FieldDecl ::= $Label_1$ `,` … `,` $Label_n$ `::` *TypeExpr* ($n \geq 1$)
DerivingDecl ::= `(` $QClassID_1$ `,` … `,` $QClassID_n$ `)` ($n \geq 0$)

NewtypeDecl ::= `newtype` *SimpleType* `=` *NewConstrDecl* [`deriving` *DerivingDecl*]
NewConstrDecl ::= *DataConstr* *SimpleTypeExpr*
| *DataConstr* `{` *Label* `::` *TypeExpr* `}`

QualTypeExpr ::= [*Context* `=>`] *TypeExpr*
Context ::= *Constraint*

| (*Constraint*$_1$, … , *Constraint*$_n$) — ($n \geq 0$)
Constraint ::= *QClassID* *ClassVarID*
| *QClassID* (*ClassVarID* *SimpleTypeExpr*$_1$ … *SimpleTypeExpr*$_n$) — ($n \geq 1$)

TypeExpr ::= *TypeAppExpr* [`->` *TypeExpr*]
TypeAppExpr ::= [*TypeAppExpr*] *SimpleTypeExpr*
SimpleTypeExpr ::= *TypeVarID*
| *GTypeConstr*
| (*TypeExpr*$_1$, … , *TypeExpr*$_n$) — (*tuple type*, $n \geq 2$)
| [*TypeExpr*] — (*list type*)
| (*TypeExpr*) — (*parenthesized type*)
GTypeConstr ::= `()` — (*unit type constructor*)
| `[]` — (*list type constructor*)
| `(->)` — (*function type constructor*)
| `(, {,} )` — (*tuple type constructor*)
| *QTypeConstrID*

DefaultDecl ::= `default` (*TypeExpr*$_1$, … , *TypeExpr*$_n$) — ($n \geq 0$)

ClassDecl ::= `class` [*SimpleContext* `=>`] *ClassID* *ClassVarID* [`where` *ClsDecls*]
ClsDecls ::= { *ClsDecl*$_1$; … ; *ClsDecl*$_n$ } — ($n \geq 0$)
ClsDecl ::= *Signature*
| *Equat*
SimpleContext ::= *SimpleConstraint*
| (*SimpleConstraint*$_1$, … , *SimpleConstraint*$_n$) — ($n \geq 0$)
SimpleConstraint ::= *QClassID* *ClassVarID*

InstanceDecl ::= `instance` [*SimpleContext* `=>`] *QClassID* *InstType* [`where` *InstDecls*]
InstDecls ::= { *InstDecl*$_1$; … ; *InstDecl*$_n$ } — ($n \geq 0$)
InstDecl ::= *Equat*
InstType ::= *GTypeConstr*
| (*GTypeConstr* *ClassVarID*$_1$ … *ClassVarID*$_n$) — ($n \geq 0$)
| (*ClassVarID*$_1$, … , *ClassVarID*$_n$) — ($n \geq 2$)
| [*ClassVarID*]
| (*ClassVarID* `->` *ClassVarID*)

FixityDecl ::= *Fixity* [*Int*] *Op*$_1$, … , *Op*$_n$ — ($n \geq 1$)
Fixity ::= `infixl` | `infixr` | `infix`

FunctionDecl ::= *Signature* | *ExternalDecl* | *Equation*
Signature ::= *Functions* `::` *QualTypeExpr*
ExternalDecl ::= *Functions* `external` — (*externally defined operations*)
Functions ::= *Function*$_1$, … , *Function*$_n$ — ($n \geq 1$)
Equation ::= *FunLhs* *Rhs*
FunLhs ::= *Function* *SimplePat*$_1$ … *SimplePat*$_n$ — ($n \geq 0$)
| *ConsPattern* *FunOp* *ConsPattern*
| (*FunLhs*) *SimplePat*$_1$ … *SimplePat*$_n$ — ($n \geq 1$)

Rhs ::= `=` *Expr* [`where` *LocalDecls*]
| *CondExprs* [`where` *LocalDecls*]
CondExprs ::= `|` *InfixExpr* `=` *Expr* [*CondExprs*]

LocalDecls ::= { *LocalDecl*$_1$; … ; *LocalDecl*$_n$ } — ($n \geq 0$)
LocalDecl ::= *FunctionDecl*

	|	*PatternDecl*	
	|	$\mathit{Variable}_1$ `,` … `,` $\mathit{Variable}_n$ `free`	$(n \geq 1)$
	|	*FixityDecl*	
PatternDecl	::=	*Pattern* *Rhs*	
Pattern	::=	*ConsPattern* [*QConOp* *Pattern*]	(*infix constructor pattern*)
ConsPattern	::=	*GDataConstr* $\mathit{SimplePat}_1$ … $\mathit{SimplePat}_n$	(*constructor pattern*, $n \geq 1$)
	|	`-` (*Int* | *Float*)	(*negative pattern*)
	|	*SimplePat*	
SimplePat	::=	*Variable*	
	|	`_`	(*wildcard*)
	|	*GDataConstr*	(*constructor*)
	|	*Literal*	(*literal*)
	|	`(` *Pattern* `)`	(*parenthesized pattern*)
	|	`(` $\mathit{Pattern}_1$ `,` … `,` $\mathit{Pattern}_n$ `)`	(*tuple pattern*, $n \geq 2$)
	|	`[` $\mathit{Pattern}_1$ `,` … `,` $\mathit{Pattern}_n$ `]`	(*list pattern*, $n \geq 1$)
	|	*Variable* `@` *SimplePat*	(*as-pattern*)
	|	`~` *SimplePat*	(*irrefutable pattern*)
	|	`(` *QFunction* $\mathit{SimplePat}_1$ … $\mathit{SimplePat}_n$ `)`	(*functional pattern*, $n \geq 1$)
	|	`(` *ConsPattern* *QFunOp* *Pattern* `)`	(*infix functional pattern*)
	|	*QDataConstr* `{` $\mathit{FieldPat}_1$ `,` … `,` $\mathit{FieldPat}_n$ `}`	(*labeled pattern*, $n \geq 0$)
FieldPat	::=	*QLabel* `=` *Pattern*	
Expr	::=	*InfixExpr* `::` *QualTypeExpr*	(*expression with type signature*)
	|	*InfixExpr*	
InfixExpr	::=	*NoOpExpr* *QOp* *InfixExpr*	(*infix operator application*)
	|	`-` *InfixExpr*	(*unary minus*)
	|	*NoOpExpr*	
NoOpExpr	::=	`\` $\mathit{SimplePat}_1$ … $\mathit{SimplePat}_n$ `->` *Expr*	(*lambda expression*, $n \geq 1$)
	|	`let` *LocalDecls* `in` *Expr*	(*let expression*)
	|	`if` *Expr* `then` *Expr* `else` *Expr*	(*conditional*)
	|	`case` *Expr* `of` `{` Alt_1 `;` … `;` Alt_n `}`	(*case expression*, $n \geq 1$)
	|	`fcase` *Expr* `of` `{` Alt_1 `;` … `;` Alt_n `}`	(*fcase expression*, $n \geq 1$)
	|	`do` `{` Stmt_1 `;` … `;` Stmt_n `;` *Expr* `}`	(*do expression*, $n \geq 0$)
	|	*FuncExpr*	
FuncExpr	::=	[*FuncExpr*] *BasicExpr*	(*application*)
BasicExpr	::=	*Variable*	(*variable*)
	|	`_`	(*anonymous free variable*)
	|	*QFunction*	(*qualified function*)
	|	*GDataConstr*	(*general constructor*)
	|	*Literal*	(*literal*)
	|	`(` *Expr* `)`	(*parenthesized expression*)
	|	`(` Expr_1 `,` … `,` Expr_n `)`	(*tuple*, $n \geq 2$)
	|	`[` Expr_1 `,` … `,` Expr_n `]`	(*finite list*, $n \geq 1$)
	|	`[` *Expr* [`,` *Expr*] `..` [*Expr*] `]`	(*arithmetic sequence*)
	|	`[` *Expr* `|` Qual_1 `,` … `,` Qual_n `]`	(*list comprehension*, $n \geq 1$)
	|	`(` *InfixExpr* *QOp* `)`	(*left section*)
	|	`(` $\mathit{QOp}_{\langle - \rangle}$ *InfixExpr* `)`	(*right section*)
	|	*QDataConstr* `{` FBind_1 `,` … `,` FBind_n `}`	(*record construction*, $n \geq 0$)
	|	$\mathit{BasicExpr}_{\langle \mathit{QDataConstr} \rangle}$ `{` FBind_1 `,` … `,` FBind_n `}`	(*record update*, $n \geq 1$)

$$
\begin{array}{rcll}
\mathit{Alt} & ::= & \mathit{Pattern}\ \texttt{->}\ \mathit{Expr}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
 & | & \mathit{Pattern}\ \mathit{GdAlts}\ [\texttt{where}\ \mathit{LocalDecls}] & \\
\mathit{GdAlts} & ::= & \texttt{|}\ \mathit{InfixExpr}\ \texttt{->}\ \mathit{Expr}\ [\mathit{GdAlts}] & \\
 & & & \\
\mathit{FBind} & ::= & \mathit{QLabel}\ \texttt{=}\ \mathit{Expr} & \\
 & & & \\
\mathit{Qual} & ::= & \mathit{Pattern}\ \texttt{<-}\ \mathit{Expr} & (\mathit{generator}) \\
 & | & \texttt{let}\ \mathit{LocalDecls} & (\mathit{local\ declarations}) \\
 & | & \mathit{Expr} & (\mathit{guard}) \\
 & & & \\
\mathit{Stmt} & ::= & \mathit{Pattern}\ \texttt{<-}\ \mathit{Expr} & \\
 & | & \texttt{let}\ \mathit{LocalDecls} & \\
 & | & \mathit{Expr} & \\
 & & & \\
\mathit{Literal} & ::= & \mathit{Int} \mid \mathit{Float} \mid \mathit{Char} \mid \mathit{String} & \\
 & & & \\
\mathit{GDataConstr} & ::= & \texttt{()} & (\mathit{unit}) \\
 & | & \texttt{[]} & (\mathit{empty\ list}) \\
 & | & \texttt{(,}\{\texttt{,}\}\texttt{)} & (\mathit{tuple}) \\
 & | & \mathit{QDataConstr} & \\
 & & & \\
\mathit{Variable} & ::= & \mathit{VariableID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\mathit{variable}) \\
\mathit{Function} & ::= & \mathit{FunctionID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\mathit{function}) \\
\mathit{QFunction} & ::= & \mathit{QFunctionID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\mathit{qualified\ function}) \\
\mathit{DataConstr} & ::= & \mathit{DataConstrID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\mathit{constructor}) \\
\mathit{QDataConstr} & ::= & \mathit{QDataConstrID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\mathit{qualified\ constructor}) \\
\mathit{Label} & ::= & \mathit{LabelID} \mid \texttt{(}\ \mathit{InfixOpID}\ \texttt{)} & (\mathit{label}) \\
\mathit{QLabel} & ::= & \mathit{QLabelID} \mid \texttt{(}\ \mathit{QInfixOpID}\ \texttt{)} & (\mathit{qualified\ label}) \\
 & & & \\
\mathit{VarOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{VariableID}\ \texttt{`} & (\mathit{variable\ operator}) \\
\mathit{FunOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{FunctionID}\ \texttt{`} & (\mathit{function\ operator}) \\
\mathit{QFunOp} & ::= & \mathit{QInfixOpID} \mid \texttt{`}\ \mathit{QFunctionID}\ \texttt{`} & (\mathit{qualified\ function\ operator}) \\
\mathit{ConOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{DataConstrID}\ \texttt{`} & (\mathit{constructor\ operator}) \\
\mathit{QConOp} & ::= & \mathit{GConSym} \mid \texttt{`}\ \mathit{QDataConstrID}\ \texttt{`} & (\mathit{qualified\ constructor\ operator}) \\
\mathit{LabelOp} & ::= & \mathit{InfixOpID} \mid \texttt{`}\ \mathit{LabelID}\ \texttt{`} & (\mathit{label\ operator}) \\
\mathit{QLabelOp} & ::= & \mathit{QInfixOpID} \mid \texttt{`}\ \mathit{QLabelID}\ \texttt{`} & (\mathit{qualified\ label\ operator}) \\
 & & & \\
\mathit{Op} & ::= & \mathit{FunOp} \mid \mathit{ConOp} \mid \mathit{LabelOp} & (\mathit{operator}) \\
\mathit{QOp} & ::= & \mathit{VarOp} \mid \mathit{QFunOp} \mid \mathit{QConOp} \mid \mathit{QLabelOp} & (\mathit{qualified\ operator}) \\
\mathit{GConSym} & ::= & \texttt{:} \mid \mathit{QInfixOpID} & (\mathit{general\ constructor\ symbol})
\end{array}
$$

5 Optimization of Curry Programs

After the invocation of the Curry front end, which parses a Curry program and translates it into the intermediate FlatCurry representation, KiCS2 applies a transformation to optimize Boolean equalities occurring in the Curry program. The ideas and details of this optimization are described in [4]. Therefore, we sketch only some basic ideas and options to influence this optimization.

Consider the following definition of the operation `last` to extract the last element in list:

```
last :: Data a => [a]  → a
last xs | xs === _ ++ [x]
        = x
  where x free
```

In order to evaluate the condition "`xs === _++[x]`", the Boolean equality is evaluated to `True` or `False` by instantiating the free variables `_` and `x`. However, since we know that a condition must be evaluated to `True` only and all evaluations to `False` can be ignored, we can use the constrained equality to obtain a more efficient program:

```
last :: Data a => [a]  → a
last xs | xs =:= _++[x]
        = x
  where x free
```

Since the selection of the appropriate equality operator is not obvious and might be tedious, KiCS2 encourages programmers to use only the Boolean equality operator "`===`" in programs. The constraint equality operator "`=:=`" can be considered as an optimization of "`===`" if it is ensured that only positive results are required, e.g., in conditions of program rules.

To support this programming style, KiCS2 has a built-in optimization phase on FlatCurry files. For this purpose, the optimizer analyzes the FlatCurry programs for occurrences of "`===`" and replaces them by "`=:=`" whenever the result `False` is not required.[5] The usage of the optimizer can be influenced by setting the property flag `bindingoptimization` in the configuration file `.kics2rc`. The following values are recognized for this flag:

`no`: Do not apply this transformation.

`fast`: This is the default value. The transformation is based on pre-computed values for the prelude operations in order to decide whether the value `False` is not required as a result of a Boolean equality. Hence, the transformation can be efficiently performed without any complex analysis.

`full`: Perform a complete "required values" analysis of the program (see [4]) and use this information to optimize programs. In most cases, this does not yield better results so that the `fast` mode is sufficient.

Hence, to turn off this optimization, one can either modify the flag `bindingoptimization` in the configuration file `.kics2rc` or dynamically pass this change to the invocation of KiCS2 by

```
... -Dbindingoptimization=no ...
```

[5]The current optimizer also replaces occurrences of `(==)` although this transformation is valid only if the corresponding `Eq` instances define equality rather than equivalence.

6 cypm: The Curry Package Manager

The Curry package manager (CPM) is a tool to distribute and install Curry libraries and applications and manage version dependencies between these libraries. Since CPM offers a lot of functionality, there is a separate manual available.[6] Therefore, we describe here only some basic CPM commands.

The executable `cypm` is located in the `bin` directory of KiCS2. Hence, if you have this directory in your path, you can start CPM by cloning a copy of the central package index repository:

```
> cypm update
```

Now you can show a short list of all packages in this index by

```
> cypm list
Name               Synopsis                                    Version
----               --------                                    -------
abstract-curry     Libraries to deal with AbstractCurry programs      2.0.0
abstract-haskell   Libraries to represent Haskell programs in Curry   2.0.0
addtypes           A tool to add missing type signatures in a Curry   2.0.0
                   program
base               Base libraries for Curry systems                   1.0.0
...
```

The command

```
> cypm info PACKAGE
```

can be used to show more information about the package with name `PACKAGE`.

Some packages do not contain only useful libraries but also tools with some binary. In order to install such tools, one can use the command

```
> cypm install PACKAGE
```

This command checks out the package in some internal directory (`$HOME/.cpm/apps_...`) and installs the binary of the tool provided by the package in `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path.

For instance, the most recent version of CPM can be installed by the following commands:

```
> cypm update
...
> cypm install cpm
... Package 'cpm-xxx' checked out ...
...
INFO  Installing executable 'cypm' into '/home/joe/.cpm/bin'
```

Now, the binary `cypm` of the most recent CPM version can be used if `$HOME/.cpm/bin` is in your path (before *kics2home*`/bin`!).

A detailed description how to write your own packages with the use of other packages can be found in the manual of CPM.

[6] http://curry-lang.org/tools/cpm

7 CurryCheck: A Tool for Testing Properties of Curry Programs

CurryCheck is a tool that supports the automation of testing Curry programs. The tests to be executed can be unit tests as well as property tests parameterized over some arguments. The tests can be part of any Curry source program and, thus, they are also useful to document the code. CurryCheck is based on EasyCheck [14]. Actually, the properties to be tested are written by combinators proposed for EasyCheck, which are actually influenced by QuickCheck [15] but extended to the demands of functional logic programming.

7.1 Installation

The current implementation of CurryCheck is a package managed by the Curry Package Manager CPM. Thus, to install the newest version of CurryCheck, use the following commands:

```
> cypm update
> cypm install currycheck
```

This downloads the newest package, compiles it, and places the executable `curry-check` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to execute CurryCheck as described below.

7.2 Testing Properties

To start with a concrete example, consider the following naive definition of reversing a list:

```
rev :: [a]  →  [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

To get some confidence in the code, we add some unit tests, i.e., test with concrete test data:

```
revNull = rev []      -=- []
rev123  = rev [1,2,3] -=- [3,2,1]
```

The operator "`-=-`" specifies a test where both sides must have a single identical value. Since this operator (as many more, see below) are defined in the library `Test.Prop`,[7] we also have to import this library. Apart from unit tests, which are often tedious to write, we can also write a property, i.e., a test parameterized over some arguments. For instance, an interesting property of reversing a list is the fact that reversing a list two times provides the input list:

```
revRevIsId xs = rev (rev xs) -=- xs
```

Note that each property is defined as a Curry operation where the arguments are the parameters of the property. Altogether, our program is as follows:

```
module Rev(rev) where
```

[7] The library `Test.Prop` is a clone of the library `Test.EasyCheck` (see package `easycheck`) which defines only the interface but not the actual test implementations. Thus, the library `Test.Prop` has less import dependencies. When CurryCheck generates programs to execute the tests, it automatically replaces references to `Test.Prop` by references to `Test.EasyCheck` in the generated programs.

```
import Test.Prop

rev :: [a] → [a]
rev []     = []
rev (x:xs) = rev xs ++ [x]

revNull = rev []      -=- []
rev123  = rev [1,2,3] -=- [3,2,1]

revRevIsId xs = rev (rev xs) -=- xs
```

Now we can run all tests by invoking the CurryCheck tool. If our program is stored in the file `Rev.curry`, we can execute the tests as follows:

```
> curry-check Rev
...
Executing all tests...
revNull (module Rev, line 7):
 Passed 1 test.
rev123 (module Rev, line 8):
 Passed 1 test.
revRevIsId_ON_BASETYPE (module Rev, line 10):
 OK, passed 100 tests.
```

Since the operation `rev` is polymorphic, the property `revRevIsId` is also polymorphic in its argument. In order to select concrete values to test this property, CurryCheck replaces such polymorphic tests by defaulting the type variable to prelude type `Ordering` (the actual default type can also be set by a command-line flag). If we want to test this property on integers numbers, we can explicitly provide a type signature, where `Prop` denotes the type of a test:

```
revRevIsId :: [Int] → Prop
revRevIsId xs = rev (rev xs) -=- xs
```

The command `curry-check` has some options to influence the output, like "`-q`" for a quiet execution (only errors and failed tests are reported) or "`-v`" for a verbose execution where all generated test cases are shown. Moreover, the return code of `curry-check` is `0` in case of successful tests, otherwise, it is `1`. Hence, CurryCheck can be easily integrated in tool chains for automatic testing.

In order to support the inclusion of properties in the source code, the operations defined the properties do not have to be exported, as show in the module `Rev` above. Hence, one can add properties to any library and export only library-relevant operations. To test these properties, CurryCheck creates a copy of the library where all operations are public, i.e., CurryCheck requires write permission on the directory where the source code is stored.

The library `Test.Prop` defines many combinators to construct properties. In particular, there are a couple of combinators for dealing with non-deterministic operations (note that this list is incomplete):

- The combinator "`<~>`" is satisfied if the set of values of both sides are equal.
- The property x `~>` y is satisfied if x evaluates to every value of y. Thus, the set of values of y must be a subset of the set of values of x.

- The property x `<~` y is satisfied if y evaluates to every value of x, i.e., the set of values of x must be a subset of the set of values of y.
- The combinator "`<~~>`" is satisfied if the multi-set of values of both sides are equal. Hence, this operator can be used to compare the number of computed solutions of two expressions.
- The property `always` x is satisfied if all values of x are true.
- The property `eventually` x is satisfied if some value of x is true.
- The property `failing` x is satisfied if x has no value, i.e., its evaluation fails.
- The property x `#` n is satisfied if x has n different values.

For instance, consider the insertion of an element at an arbitrary position in a list:

```
insert :: a → [a] → [a]
insert x xs     = x : xs
insert x (y:ys) = y : insert x ys
```

The following property states that the element is inserted (at least) at the beginning or the end of the list:

```
insertAsFirstOrLast :: Int → [Int] → Prop
insertAsFirstOrLast x xs = insert x xs ~> (x:xs ? xs++[x])
```

A well-known application of `insert` is to use it to define a permutation of a list:

```
perm :: [a] → [a]
perm []     = []
perm (x:xs) = insert x (perm xs)
```

We can check whether the length of a permuted lists is unchanged:

```
permLength :: [Int] → Prop
permLength xs = length (perm xs) <~> length xs
```

Note that the use of "`<~>`" is relevant since we compare non-deterministic values. Actually, the left argument evaluates to many (identical) values.

One might also want to check whether `perm` computes the correct number of solutions. Since we know that a list of length n has $n!$ permutations, we write the following property:

```
permCount :: [Int] → Prop
permCount xs = perm xs # fac (length xs)
```

where `fac` is the factorial function. However, this test will be falsified with the argument `[1,1]`. Actually, this list has only one permuted value since the two possible permutations are identical and the combinator "`#`" counts the number of *different* values. The property would be correct if all elements in the input list `xs` are different. This can be expressed by a conditional property: the property b `==>` p is satisfied if p is satisfied for all values where b evaluates to `True`. Therefore, if we define a predicate `allDifferent` by

```
allDifferent []     = True
allDifferent (x:xs) = x `notElem` xs && allDifferent xs
```

then we can reformulate our property as follows:

```
permCount xs = allDifferent xs ==> perm xs # fac (length xs)
```

Now consider a predicate to check whether a list is sorted:

```
sorted :: [Int] → Bool
sorted []       = True
sorted [_]      = True
sorted (x:y:zs) = x<=y && sorted (y:zs)
```

This predicate is useful to test whether there are also sorted permutations:

```
permIsEventuallySorted :: [Int] → Prop
permIsEventuallySorted xs = eventually $ sorted (perm xs)
```

The previous operations can be exploited to provide a high-level specification of sorting a list:

```
psort :: [Int] → [Int]
psort xs | sorted ys = ys
 where ys = perm xs
```

Again, we can write some properties:

```
psortIsAlwaysSorted xs = always $ sorted (psort xs)

psortKeepsLength xs = length (psort xs) <~> length xs
```

Of course, the sort specification via permutations is not useful in practice. However, it can be used as an oracle to test more efficient sorting algorithms like quicksort:

```
qsort :: [Int] → [Int]
qsort []     = []
qsort (x:l)  = qsort (filter (<x) l) ++ x : qsort (filter (>x) l)
```

The following property specifies the correctness of quicksort:

```
qsortIsSorting xs = qsort xs <~> psort xs
```

Actually, if we test this property, we obtain a failure:

```
> curry-check ExampleTests
...
qsortIsSorting (module ExampleTests, line 53) failed
Falsified by third test.
Arguments:
[1,1]
Results:
[1]
```

The result shows that, for the given argument `[1,1]`, an element has been dropped in the result. Hence, we correct our implementation, e.g., by replacing `(>x)` with `(>=x)`, and obtain a successful test execution.

For I/O operations, it is difficult to execute them with random data. Hence, CurryCheck only supports specific I/O unit tests:

- a `` `returns` `` x is satisfied if the I/O action a returns the value x.

- a `sameReturns` b is satisfied if the I/O actions a and b return identical values.

Since CurryCheck executes the tests written in a source program in their textual order, one can write several I/O tests that are executed in a well-defined order.

7.3 Generating Test Data

CurryCheck test properties by enumerating test data and checking a given property with these values. Since these values are generated in a systematic way, one can even prove a property if the number of test cases is finite. For instance, consider the following property from Boolean logic:

```
neg_or b1 b2 = not (b1 || b2) -=- not b1 && not b2
```

This property is validated by checking it with all possible values:

```
> curry-check -v ExampleTests
...
0:
False
False
1:
False
True
2:
True
False
3:
True
True
neg_or (module ExampleTests, line 67):
 Passed 4 tests.
```

However, if the test data is infinite, like lists of integers, CurryCheck stops checking after a given limit for all tests. As a default, the limit is 100 tests but it can be changed by the command-line flag "`-m`". For instance, to test each property with 200 tests, CurryCheck can be invoked by

```
> curry-check -m 200 ExampleTests
```

For a given type, CurryCheck automatically enumerates all values of this type (except for function types). In KiCS2, this is done by exploiting the functional logic features of Curry, i.e., by simply collecting all values of a free variable. For instance, the library `Test.EasyCheck` defines an operation

```
valuesOf :: a → [a]
```

which computes the list of all values of the given argument according to a fixed strategy (in the current implementation: randomized level diagonalization [14]). For instance, we can get 20 values for a list of integers by

```
Test.EasyCheck> take 20 (valuesOf (_::[Int]))
[[],[-1],[-3],[0],[1],[-1,0],[-2],[0,0],[3],[-1,1],[-3,0],[0,1],[2],
[-1,-1],[-5],[0,-1],[5],[-1,2],[-9],[0,2]]
```

Since the features of PAKCS for search space exploration are more limited, PAKCS uses in CurryCheck explicit generators for search tree structures which are defined in the module `SearchTreeGenerators`. For instance, the operations

```
genInt :: SearchTree Int

genList :: SearchTree a → SearchTree [a]
```

generates (infinite) trees of integer and lists values. To extract all values in a search tree, the library `Test.EasyCheck` also defines an operation

```
valuesOfSearchTree :: SearchTree a → [a]
```

so that we obtain 20 values for a list of integers in PAKCS by

```
...> take 20 (valuesOfSearchTree (genList genInt))
[[],[1],[1,1],[1,-1],[2],[6],[3],[5],[0],[0,1],[0,0],[-1],[-1,0],[-2],
[-3],[1,5],[1,0],[2,-1],[4],[3,-1]]
```

Apart from the different implementations, CurryCheck can test properties on predefined types, as already shown, as well as on user-defined types. For instance, we can define our own Peano representation of natural numbers with an addition operation and two properties as follows:

```
data Nat = Z | S Nat

add :: Nat → Nat → Nat
add Z     n = n
add (S m) n = S(add m n)

addIsCommutative x y = add x y -=- add y x

addIsAssociative x y z = add (add x y) z -=- add x (add y z)
```

Properties can also be defined for polymorphic types. For instance, we can define general polymorphic trees, operations to compute the leaves of a tree and mirroring a tree as follows:

```
data Tree a = Leaf a | Node [Tree a]

leaves (Leaf x) = [x]
leaves (Node ts) = concatMap leaves ts

mirror (Leaf x) = Leaf x
mirror (Node ts) = Node (reverse (map mirror ts))
```

Then we can state and check two properties on mirroring:

```
doubleMirror t = mirror (mirror t) -=- t

leavesOfMirrorAreReversed t = leaves t -=- reverse (leaves (mirror t))
```

In some cases, it might be desirable to define own test data since the generated structures are not appropriate for testing (e.g., balanced trees to check algorithms that require work on balanced trees). Of course, one could drop undesired values by an explicit condition. For instance, consider the following operation that adds all numbers from 0 to a given limit:

```
sumUp n = if n==0 then 0 else n + sumUp (n-1)
```

Since there is also a simple formula to compute this sum, we can check it:

```
sumUpIsCorrect n = n>=0 ==> sumUp n -=- n * (n+1) `div` 2
```

Note that the condition is important since `sumUp` diverges on negative numbers. CurryCheck tests this property by enumerating integers, i.e., also many negative numbers which are dropped for the tests. In order to generate only valid test data, we define our own generator for a search tree containing only valid data:

```
genInt = genCons0 0 ||| genCons1 (+1) genInt
```

The combinator `genCons0` constructs a search tree containing only this value, whereas `genCons1` constructs from a given search tree a new tree where the function given in the first argument is applied to all values. Similarly, there are also combinators `genCons2`, `genCons3` etc. for more than one argument. The combinator "`|||`" combines two search trees.

If the Curry program containing properties defines a generator operation with the name `gen`τ, then CurryCheck uses this generator to test properties with argument type τ. Hence, if we put the definition of `genInt` in the Curry program where `sumUpIsCorrect` is defined, the values to check this property are only non-negative integers. Since these integers are slowly increasing, i.e., the search tree is actually degenerated to a list, we can also use the following definition to obtain a more balanced search tree:

```
genInt = genCons0 0 ||| genCons1 (\n → 2*(n+1)) genInt
                    ||| genCons1 (\n → 2*n+1)   genInt
```

The library `SearchTree` defines the structure of search trees as well as operations on search trees, like limiting the depth of a search tree (`limitSearchTree`) or showing a search tree (`showSearchTree`). For instance, to structure of the generated search tree up to some depth can be visualized as follows:

```
...SearchTree> putStr (showSearchTree (limitSearchTree 6 genInt))
```

If we want to use our own generator only for specific properties, we can do so by introducing a new data type and defining a generator for this data type. For instance, to test only the operation `sumUpIsCorrect` with non-negative integers, we do not define a generator `genInt` as above, but define a wrapper type for non-negative integers and a generator for this type:

```
data NonNeg = NonNeg { nonNeg :: Int }

genNonNeg = genCons1 NonNeg genNN
 where
   genNN = genCons0 0 ||| genCons1 (\n → 2*(n+1)) genNN
                      ||| genCons1 (\n → 2*n+1)   genNN
```

Now we can either redefine `sumUpIsCorrect` on this type

```
sumUpIsCorrectOnNonNeg (NonNeg n) = sumUp n -=- n * (n+1) `div` 2
```

or we simply reuse the old definition by

```
sumUpIsCorrectOnNonNeg = sumUpIsCorrect . nonNeg
```

7.4 Checking Equivalence of Operations

CurryCheck supports also equivalence tests for operations. Two operations are considered as *equivalent* if they can be replaced by each other in any possible context without changing the computed values (this is also called *contextual equivalence* and precisely defined in [3] for functional logic programs). For instance, the Boolean operations

```
f1 :: Bool → Bool                 f2 :: Bool → Bool
f1 x = not (not x)                f2 x = x
```

are equivalent, whereas

```
g1 :: Bool → Bool                 g2 :: Bool → Bool
g1 False = True                   g2 x = True
g1 True  = True
```

are not equivalent: `g1 failed` has no value but `g2 failed` evaluates to `True`.

To check the equivalence of operations, one can use the property combinator `<=>`:

```
f1_equiv_f2 = f1 <=> f2
g1_equiv_g2 = g1 <=> g2
```

The left and right argument of this combinator must be a defined operation or a defined operation with a type annotation in order to specify the argument types used for checking this property.

CurryCheck transforms such properties into properties where both operations are compared w.r.t. all partial values and partial results. The details are described in [7].

It should be noted that CurryCheck can test the equivalence of non-terminating operations provided that they are *productive*, i.e., always generate (outermost) constructors after a finite number of steps (otherwise, the test of CurryCheck might not terminate). For instance, CurryCheck reports a counter-example to the equivalence of the following non-terminating operations:

```
ints1 n = n : ints1 (n+1)

ints2 n = n : ints2 (n+2)

-- This property will be falsified by CurryCheck:
ints1_equiv_ints2 = ints1 <=> ints2
```

This is done by iteratively guessing depth-bounds, computing both operations up to these depth-bounds, and comparing the computed results. Since this might be a long process, CurryCheck supports a faster comparison of operations when it is known that they are terminating. If the name of a test contains the suffix `'TERMINATE`, CurryCheck assumes that the operations to be tested are terminating, i.e., they always yields a result when applied to ground terms. In this case, CurryCheck does not iterate over depth-bounds but evaluates operations completely. For instance, consider the following definition of permutation sort (the operations `perm` and `sorted` are defined above):

```
psort :: Ord a => [a] → [a]
psort xs | sorted ys = ys
  where ys = perm xs
```

A different definition can be obtained by defining a partial identity on sorted lists:

```
isort :: Ord a => [a] → [a]
```

```
isort xs = idSorted (perm xs)
 where idSorted []              = []
       idSorted [x]             = [x]
       idSorted (x:y:ys) | x<=y = x : idSorted (y:ys)
```

We can test the equivalence of both operations by specializing both operations on some ground type (otherwise, the type checker reports an error due to an unspecified type `Ord` context):

```
psort_equiv_isort = psort <=> (isort :: [Int]  →  [Int])
```

CurryCheck reports a counter example by the 274th test. Since both operations are terminating, we can also check the following property:

```
psort_equiv_isort'TERMINATE = psort <=> (isort :: [Int]  →  [Int])
```

Now a counter example is found by the 21th test.

Instead of annotating the property name to use more efficient equivalence tests for terminating operations, one can also ask CurryCheck to analyze the operations in order to safely approximate termination or productivity properties. For this purpose, one can call CurryCheck with the option "`--equivalence=`*equiv*" or "`-e`*equiv*". The parameter *equiv* determines the mode for equivalence checking which must have one of the following values (or a prefix of them):

`manual`: This is the default mode. In this mode, all equivalence tests are executed with first technique described above, unless the name of the test has the suffix `'TERMINATE`.

`autoselect`: This mode automatically selects the improved transformation for terminating operations by a program analysis, i.e., if it can be proved that both operations are terminating, then the equivalence test for terminating operations is used. It is also used when the name of the test has the suffix `'TERMINATE`.

`safe`: This mode analyzes the productivity behavior of operations. If it can be proved that both operations are terminating or the test name has the suffix `'TERMINATE`, then the more efficient equivalence test for terminating operations is used. If it can be proved that both operations are productive or the test name has the suffix `'PRODUCTIVE`, then the first general test technique is used. Otherwise, the equivalence property is *not* tested. Thus, this mode is useful if one wants to ensure that all equivalence tests always terminate (provided that the additional user annotations are correct).

`ground`: In this mode, only ground equivalence is tested, i.e., each equivalence property

```
g1_equiv_g2 = g1 <=> g2
```

is transformed into a property which states that both operations must deliver the same values on same input values, i.e.,

```
g1_equiv_g2 x1 ... xn = g1 x1 ... xn <~> g2 x1 ... xn
```

Note this property is more restrictive than contextual equivalence. For instance, the non-equivalence of `g1` and `g2` as shown above cannot be detected by testing ground equivalence only.

7.5 Checking Contracts and Specifications

The expressive power of Curry supports writing high-level specifications as well as efficient implementations for a given problem in the same programming language, as discussed in [3]. If a specification or contract is provided for some function, then CurryCheck automatically generates properties to test this specification or contract.

Following the notation proposed in [3], a *specification* for an operation f is an operation f`'spec` of the same type as f. A *contract* consists of a pre- and a postcondition, where the precondition could be omitted. A *precondition* for an operation f of type $\tau \to \tau'$ is an operation

```
f'pre :: τ → Bool
```

whereas a *postcondition* for f is an operation

```
f'post :: τ → τ' → Bool
```

which relates input and output values (the generalization to operations with more than one argument is straightforward).

As a concrete example, consider again the problem of sorting a list. We can write a postcondition and a specification for a sort operation `sort` and an implementation via quicksort as follows (where `sorted` and `perm` are defined as above):

```
-- Postcondition: input and output lists should have the same length
sort'post xs ys = length xs == length ys

-- Specification:
-- A correct result is a permutation of the input which is sorted.
sort'spec :: [Int] → [Int]
sort'spec xs | sorted ys = ys  where ys = perm xs

-- An implementation of sort with quicksort:
sort :: [Int] → [Int]
sort []     = []
sort (x:xs) = sort (filter (<x) xs) ++ [x] ++ sort (filter (>=x) xs)
```

If we process this program with CurryCheck, properties to check the specification and postcondition are automatically generated. For instance, a specification is satisfied if it is equivalent to its implementation, and a postcondition is satisfied if each value computed for some input satisfies the postcondition relation between input and output. For our example, CurryCheck generates the following properties (if there are also preconditions for some operation, these preconditions are used to restrict the test cases via the condition operater "`==>`"):

```
sortSatisfiesPostCondition :: [Int] → Prop
sortSatisfiesPostCondition x = always (sort'post x (sort x))

sortSatisfiesSpecification :: Prop
sortSatisfiesSpecification = sort <=> sort'spec
```

7.6 Combining Testing and Verification

Usually, CurryCheck tests all user-defined properties as well as postconditions or specifications, as described in Section 7.5. If a programmer uses some other tool to verify such properties, it is not necessary to check such properties with test data. In order to advice CurryCheck to do so, it is sufficient to store the proofs in specific files. Since the proof might be constructed by some tool unknown to CurryCheck or even manually, CurryCheck does not check the proof file but trusts the programmer and uses a naming convention for files containing proofs. If there is a property `p` in a module `M` for which a proof in file `proof-M-p.*` (the name is case independent), then CurryCheck assumes that this file contains a valid proof for this property. For instance, the following property states that sorting a list does not change its length:

```
sortlength xs = length (sort xs) <~> length xs
```

If this property is contained in module `Sort` and there is a file `proof-Sort-sortlength.txt` containing a proof for this property, CurryCheck considers this property as valid and does not check it. Moreover, it uses this information to simplify other properties to be tested. For instance, consider the property `sortSatisfiesPostCondition` of Section 7.5. This can be simplified to `always True` so that it does not need to be tested.

One can also provide proofs for generated properties, e.g., determinism, postconditions, specifications, so that they are not tested:

- If there is a proof file `proof-`M`-`f`-IsDeterministic.*`, a determinism annotation for operation $M.f$ is not tested.
- If there is a proof file `proof-`M`-`f`-SatisfiesPostCondition.*`, a postcondition for operation $M.f$ is not tested.
- If there is a proof file `proof-`M`-`f`-SatisfiesSpecification.*`, a specification for operation $M.f$ is not tested.

Note that the file suffix and all non-alpha-numberic characters in the name of the proof file are ignored. Furthermore, the name is case independent This should provide enough flexibility when other verification tools require specific naming conventions. For instance, a proof for the property `Sort.sortlengh` could be stored in the following files in order to be considered by CurryCheck:

```
proof-Sort-sortlength.tex
PROOF_Sort_sortlength.agda
Proof-Sort_sortlength.smt
ProofSortSortlength.smt
```

7.7 Checking Usage of Specific Operations

In addition to testing dynamic properties of programs, CurryCheck also examines the source code of the given program for unintended uses of specific operations (these checks can be omitted via the option "`--nosource`"). Currently, the following source code checks are performed:

- The prelude operation "`=:<=`" is used to implement functional patterns [1]. It should not be used in source programs to avoid unintended uses. Hence, CurryCheck reports such unintended uses.

- Set functions [2] are used to encapsulate all non-deterministic results of some function in a set structure. Hence, for each top-level function f of arity n, the corresponding set function can be expressed in Curry (via operations defined in the library `SetFunctions`) by the application "`set`n f" (this application is used in order to extend the syntax of Curry with a specific notation for set functions). However, it is not intended to apply the operator "`set`n" to lambda abstractions, locally defined operations or operations with an arity different from n. Hence, CurryCheck reports such unintended uses of set functions.

8 CurryBrowser: A Tool for Analyzing and Browsing Curry Programs

CurryBrowser is a tool to browse through the modules and operations of a Curry application, show them in various formats, and analyze their properties.[8] Moreover, it is constructed in a way so that new analyzers can easily be connected to CurryBrowser. A detailed description of the ideas behind this tool can be found in [20, 21].

8.1 Installation

The current implementation of CurryBrowser is a package managed by the Curry Package Manager CPM (see also Section 6). Thus, to install the newest version of CurryBrowser, use the following commands:

```
> cypm update
> cypm install currybrowse
```

This downloads the newest package, compiles it, and places the executable `curry-browse` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to execute CurryBrowser as described below.

8.2 Basic Usage

When CurryBrowser is installed as described above, it can be started in two ways:

- In the KiCS2 environment after loading the module `mod` and typing the command "`:browse`".
- As a shell command (provided that `$HOME/.cpm/bin` is in your path): `curry-browse mod`

Here, "`mod`" is the name of the main module of a Curry application. After the start, CurryBrowser loads the interfaces of the main module and all imported modules before a GUI is created for interactive browsing.

To get an impression of the use of CurryBrowser, Figure 1 shows a snapshot of its use on a particular application (here: the implementation of CurryBrowser). The upper list box in the left column shows the modules and their imports in order to browse through the modules of an application. Similarly to directory browsers, the list of imported modules of a module can be opened or closed by clicking. After selecting a module in the list of modules, its source code, interface, or various other formats of the module can be shown in the main (right) text area. For instance, one can show pretty-printed versions of the intermediate flat programs (see below) in order to see how local function definitions are translated by lambda lifting [29] or pattern matching is translated into case expressions [16, 33]. Since Curry is a language with parametric polymorphism and type inference, programmers often omit the type signatures when defining functions. Therefore, one can also view (and store) the selected module as source code where missing type signatures are added.

[8] Although CurryBrowser is implemented in Curry, some functionalities of it require an installed graph visualization tool (dot `http://www.graphviz.org/`), otherwise they have no effect.

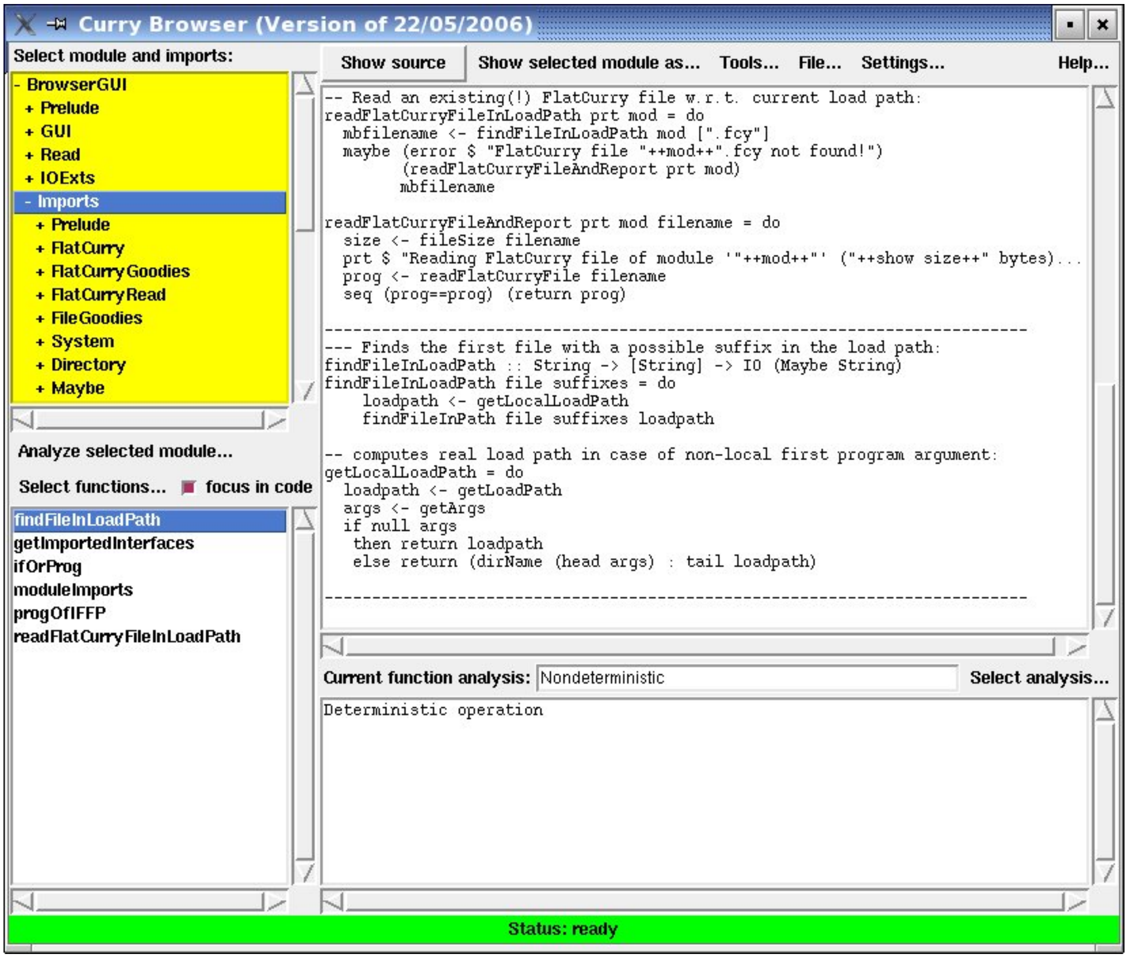

Figure 1: Snapshot of the main window of CurryBrowser

Below the list box for selecting modules, there is a menu ("Analyze selected module") to analyze all functions of the currently selected module at once. This is useful to spot some functions of a module that could be problematic in some application contexts, like functions that are impure (i.e., the result depends on the evaluation time) or partially defined (i.e., not evaluable on all ground terms). If such an analysis is selected, the names of all functions are shown in the lower list box of the left column (the "function list") with prefixes indicating the properties of the individual functions.

The function list box can be also filled with functions via the menu "Select functions". For instance, all functions or only the exported functions defined in the currently selected module can be shown there, or all functions from different modules that are directly or indirectly called from a currently selected function. This list box is central to focus on a function in the source code of some module or to analyze some function, i.e., showing their properties. In order to focus on a function, it is sufficient to check the "focus on code" button. To analyze an individually selected function, one can select an analysis from the list of available program analyses (through the menu "Select analysis"). In this case, the analysis results are either shown in the text box below the main text area or visualized by separate tools, e.g., by a graph drawing tool for visualizing call graphs. Some analyses are local, i.e., they need only to consider the local definition of this function (e.g., "Calls directly," "Overlapping rules," "Pattern completeness"), where other analyses are global, i.e., they consider the definitions of all functions directly or indirectly called by this function (e.g., "Depends on," "Solution complete," "Set-valued"). Finally, there are a few additional tools integrated into CurryBrowser, for instance, to visualize the import relation between all modules as a dependency graph. These tools are available through the "Tools" menu.

More details about the use of CurryBrowser and all built-in analyses are available through the "Help" menu of CurryBrowser.

9 `curry-doc`: A Documentation Generator for Curry Programs

CurryDoc is a tool in the KiCS2 distribution that generates the documentation for a Curry program (i.e., the main module and all its imported modules) in HTML format. The generated HTML pages contain information about all data types and functions exported by a module as well as links between the different entities. Furthermore, some information about the definitional status of functions (like rigid, flexible, external, complete, or overlapping definitions) are provided and combined with documentation comments provided by the programmer.

9.1 Installation

The current implementation of CurryDoc is a package managed by the Curry Package Manager CPM (see also Section 6). Thus, to install the newest version of CurryDoc, use the following commands:

```
> cypm update
> cypm install currydoc
```

This downloads the newest package, compiles it, and places the executable `curry-doc` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to execute CurryDoc as described below.

9.2 Documentation Comments

A *documentation comment* starts at the beginning of a line with "`---` " (also in literate programs!). All documentation comments immediately before a definition of a datatype or (top-level) function are kept together.[9] The documentation comments for the complete module occur before the first "module" or "import" line in the module. The comments can also contain several special tags. These tags must be the first thing on its line (in the documentation comment) and continues until the next tag is encountered or until the end of the comment. The following tags are recognized:

`@author` *comment*
: Specifies the author of a module (only reasonable in module comments).

`@version` *comment*
: Specifies the version of a module (only reasonable in module comments).

`@cons` *id comment*
: A comment for the constructor *id* of a datatype (only reasonable in datatype comments).

`@param` *id comment*
: A comment for function parameter *id* (only reasonable in function comments). Due to pattern matching, this need not be the name of a parameter given in the declaration of the function but all parameters for this functions must be commented in left-to-right order (if they are commented at all).

[9] The documentation tool recognizes this association from the first identifier in a program line. If one wants to add a documentation comment to the definition of a function which is an infix operator, the first line of the operator definition should be a type definition, otherwise the documentation comment is not recognized.

@return *comment*
: A comment for the return value of a function (only reasonable in function comments).

The comment of a documented entity can be any string in Markdown syntax (the currently supported set of elements is described in the Curry package markdown). For instance, it can contain Markdown annotations for emphasizing elements (e.g., _verb_), strong elements (e.g., **important**), code elements (e.g., `3+4`), code blocks (lines prefixed by four blanks), unordered lists (lines prefixed by " * "), ordered lists (lines prefixed by blanks followed by a digit and a dot), quotations (lines prefixed by "> "), and web links of the form "<http://...>" or "[link text](http://...)". If the Markdown syntax should not be used, one could run CurryDoc with the parameter "--nomarkdown".

The comments can also contain markups in HTML format so that special characters like "<" must be quoted (e.g., "<"). However, header tags like <h1> should not be used since the structuring is generated by CurryDoc. In addition to Markdown or HTML markups, one can also mark *references to names* of operations or data types in Curry programs which are translated into links inside the generated HTML documentation. Such references have to be enclosed in single quotes. For instance, the text 'conc' refers to the Curry operation conc inside the current module whereas the text 'Prelude.reverse' refers to the operation reverse of the module Prelude. If one wants to write single quotes without this specific meaning, one can escape them with a backslash:

```
--- This is a comment without a \'reference\'.
```

To simplify the writing of documentation comments, such escaping is only necessary for single words, i.e., if the text inside quotes has not the syntax of an identifier, the escaping can be omitted, as in

```
--- This isn't a reference.
```

The following example text shows a Curry program with some documentation comments:

```
--- This is an
--- example module.
--- @author Michael Hanus
--- @version 0.1

module Example where

--- The function `conc` concatenates two lists.
--- @param xs - the first list
--- @param ys - the second list
--- @return a list containing all elements of `xs` and `ys`
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys
-- this comment will not be included in the documentation

--- The function `last` computes the last element of a given list.
--- It is based on the operation 'conc' to concatenate two lists.
--- @param xs - the given input list
--- @return last element of the input list
last xs | conc ys [x] =:= xs  = x   where x,ys free
```

```
--- This data type defines _polymorphic_ trees.
--- @cons Leaf - a leaf of the tree
--- @cons Node - an inner node of the tree
data Tree a = Leaf a | Node [Tree a]
```

9.3 Generating Documentation

To generate the documentation, execute the command

```
curry-doc Example
```

This command creates the directory `DOC_Example` (if it does not exist) and puts all HTML documentation files for the main program module `Example` and all its imported modules in this directory together with a main index file `index.html`. If one prefers another directory for the documentation files, one can also execute the command

```
curry-doc docdir Example
```

where `docdir` is the directory for the documentation files.

In order to generate the common documentation for large collections of Curry modules (e.g., the libraries contained in the KiCS2 distribution), one can call `curry-doc` with the following options:

`curry-doc --noindexhtml docdir Mod` : This command generates the documentation for module `Mod` in the directory `docdir` without the index pages (i.e., main index page and index pages for all functions and constructors defined in `Mod` and its imported modules).

`curry-doc --onlyindexhtml docdir Mod1 Mod2 ...Mod`n : This command generates only the index pages (i.e., a main index page and index pages for all functions and constructors defined in the modules `Mod1`, `Mod2`,...,`Mod`n and their imported modules) in the directory `docdir`.

10 CurryVerify: A Tool to Support the Verification of Curry Programs

CurryVerify is a tool that supports the verification of Curry programs with the help of other theorem provers or proof assistants. Basically, CurryVerify extends CurryCheck (see Section 7), which tests given properties of a program, by the possibility to verify these properties. For this purpose, CurryVerify translates properties into the input language of other theorem provers or proof assistants. This is done by collecting all operations directly or indirectly involved in a given property and translating them together with the given property.

Currently, only Agda [31] is supported as a target language for verification (but more target languages may be supported in future releases). The basic schemes to translate Curry programs into Agda programs are presented in [8]. That paper also describes the limitations of this approach. Since Curry is a quite rich programming language, not all constructs of Curry are currently supported in the translation process (e.g., no case expressions, local definitions, list comprehensions, do notations, etc). Only a kernel language, where the involved rules correspond to a term rewriting system, are translated into Agda. However, these limitations might be relaxed in future releases. Hence, the current tool should be considered as a first prototypical approach to support the verification of Curry programs.

10.1 Installation

The current implementation of CurryVerify is a package managed by the Curry Package Manager CPM (see also Section 6). Thus, to install the newest version of CurryVerify, use the following commands:

```
> cypm update
> cypm install verify
```

This downloads the newest package, compiles it, and places the executable `curry-verify` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to execute CurryVerify as described below.

10.2 Basic Usage

To translate the properties of a Curry program stored in the file `prog.curry` into Agda, one can invoke the command

```
curry-verify prog
```

This generates for each property p in module `prog` an Agda program "`TO-PROVE-`p`.agda`". If one completes the proof obligation in this file, the completed file should be renamed into "`PROOF-`p`.agda`". This has the effect that CurryCheck does not test this property again but trusts the proof and use this knowledge to simplify other tests.

As a concrete example, consider the following Curry module `Double`, shown in Figure 2, which uses the Peano representation of natural numbers (module `Nat`) to define an operation to double the value of a number, a non-deterministic operation `coin` which returns its argument or its incremented

```
module Double(double,coin,even) where

import Nat
import Test.Prop

double x = add x x

coin x = x ? S x

even Z         = True
even (S Z)     = False
even (S (S n)) = even n

evendoublecoin x = always (even (double (coin x)))
```

Figure 2: Curry program `Double.curry`

argument, and a predicate to test whether a number is even. Furthermore, it contains a property specifying that doubling the coin of a number is always even.

In order to prove the correctness of this property, we translate it into an Agda program by executing

```
> curry-verify Double
...
Agda module 'TO-PROVE-evendoublecoin.agda' written.
If you completed the proof, rename it to 'PROOF-evendoublecoin.agda'.
```

The Curry program is translated with the default scheme (see further options below) based on the "planned choice" scheme, described in [8]. The result of this translation is shown in Figure 3.

The Agda program contains all operations involved in the property and the property itself. Non-deterministic operations, like `coin`, have an additional additional argument of the abstract type `Choice` that represents the plan to execute some non-deterministic branch of the program. By proving the property for all possible branches as correct, it universally holds.

In our example, the proof is quite easy. First, we prove that the addition of a number to itself is always even (lemma `even-add-x-x`, which uses an auxiliary lemma `add-suc`). Then, the property is an immediate consequence of this lemma:

```
add-suc : ∀ (x y : ℕ) → add x (suc y) ≡ suc (add x y)
add-suc zero    y = refl
add-suc (suc x) y rewrite add-suc x y = refl

even-add-x-x : ∀ (x : ℕ) → even (add x x) ≡ tt
even-add-x-x zero    = refl
even-add-x-x (suc x) rewrite add-suc x x | even-add-x-x x = refl

evendoublecoin : (c1 : Choice) → (x : ℕ) → (even (double (coin c1 x))) ≡ tt
evendoublecoin c1 x rewrite even-add-x-x (coin c1 x) = refl
```

As the proof is complete, we rename this Agda program into `PROOF-evendoublecoin.agda` so that the proof can be used by further invocations of CurryCheck.

10.3 Options

The command `curry-verify` can be parameterized with various options. The available options can also be shown by executing

```
curry-verify --help
```

The options are briefly described in the following.

`-h`, `-?`, `--help` These options trigger the output of usage information.

`-q`, `--quiet` Run quietly and produce no informative output. However, the exit code will be non-zero if some translation error occurs.

`-v[`n`]`, `--verbosity[=`n`]` Set the verbosity level to an optional value. The verbosity level `0` is the same as option `-q`. The default verbosity level `1` shows the translation progress. The verbosity level `2` (which is the same as omitting the level) shows also the generated (Agda) program. The verbosity level `3` shows also more details about the translation process.

`-n`, `--nostore` Do not store the translated program in a file but show it only.

`-p` p, `--property=`p As a default, all properties occurring in the source program are translated. If this option is provided, only property p is translated.

`-t` t, `--target=`t Define the target language of the translation. Currently, only t = `Agda` is supported, which is also the default.

`-s` s, `--scheme=`s Define the translation scheme used to represent Curry programs in the target language.

For the target `Agda`, the following schemes are supported:

`choice` Use the "planned choice" scheme, see [8] (this is the default). In this scheme, the choices made in a non-deterministic computation are abstracted by passing a parameter for these choices.

`nondet` Use the "set of values" scheme, see [8], where non-deterministic values are represented in a tree structure.

```
-- Agda program using the Iowa Agda library

open import bool

module TO-PROVE-evendoublecoin
  (Choice : Set)
  (choose : Choice → 𝔹)
  (lchoice : Choice → Choice)
  (rchoice : Choice → Choice)
  where

open import eq
open import nat
open import list
open import maybe

---------------------------------------------------------------------------
-- Translated Curry operations:

add : ℕ → ℕ → ℕ
add zero x = x
add (suc y) z = suc (add y z)

coin : Choice → ℕ → ℕ
coin c1 x = if choose c1 then x else suc x

double : ℕ → ℕ
double x = add x x

even : ℕ → 𝔹
even zero = tt
even (suc zero) = ff
even (suc (suc x)) = even x

---------------------------------------------------------------------------

evendoublecoin : (c1 : Choice) → (x : ℕ) → (even (double (coin c1 x))) ≡ tt
evendoublecoin c1 x = ?
```

Figure 3: Agda program `TO-PROVE-evendoublecoin.agda`

11 CurryPP: A Preprocessor for Curry Programs

The Curry preprocessor "`currypp`" implements various transformations on Curry source programs. It supports some experimental language extensions that might become part of the standard parser of Curry in some future version.

Currently, the Curry preprocessor supports the following extensions that will be described below in more detail:

Integrated code: This extension allows to integrate code written in some other language into Curry programs, like regular expressions, format specifications ("`printf`"), HTML and XML code.

Default rules: If this feature is used, one can add a default rule to top-level operations defined in a Curry module. The idea of default rules is described in [5].

Contracts: If this feature is used, the Curry preprocessor looks for contracts (i.e., specification, pre- and postconditions) occurring in a Curry module and adds them as assertions that are checked during the execution of the program. Currently, only strict assertion checking is supported which might change the operational behavior of the program. The idea and usage of contracts is described in [3].

11.1 Installation

The current implementation of Curry preprocessor is a package managed by the Curry Package Manager CPM. Thus, to install the newest version of `currypp`, use the following commands:

```
> cypm update
> cypm install currypp
```

This downloads the newest package, compiles it, and places the executable `currypp` into the directory `$HOME/.cpm/bin`. Hence one should add this directory to the path in order to use the Curry preprocessor as described below.

11.2 Basic Usage

In order to apply the preprocessor when loading a Curry source program into KiCS2, one has to add an option line at the beginning of the source program. For instance, in order to use default rules in a Curry program, one has to put the line

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=defaultrules #-}
```

at the beginning of the program. This option tells the KiCS2 front end to process the Curry source program with the program `currypp` before actually parsing the source text.

The option "`defaultrules`" has to be replaced by "`contracts`" to enable dynamic contract checking. To support integrated code, one has to set the option "`foreigncode`" (which can also be combined with "`defaultrules`"). If one wants to see the result of the transformation, one can also set the option "`-o`". This has the effect that the transformed source program is stored in the file `Prog.curry.CURRYPP` if the name of the original program is `Prog.curry`.

For instance, in order to use integrated code and default rules in a module and store the transformed program, one has to put the line

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=foreigncode --optF=defaultrules --optF=-o #-}
```

at the beginning of the program. If the options about the kind of preprocessing is omitted, all kinds of preprocessing are applied. Thus, the preprocessor directive

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp #-}
```

is equivalent to

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=foreigncode --optF=defaultrules --optF=contracts #-}
```

11.3 Integrated Code

Integrated code is enclosed in at least two back ticks and ticks in a Curry program. The number of starting back ticks and ending ticks must always be identical. After the initial back ticks, there must be an identifier specifying the kind of integrated code, e.g., `regex` or `html` (see below). For instance, if one uses regular expressions (see below for more details), the following expressions are valid in source programs:

```
  match ``regex (a|(bc*))+''
  match ````regex aba*c''''
```

The Curry preprocessor transforms these code pieces into regular Curry expressions. For this purpose, the program containing this code must start with the preprocessing directive

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=foreigncode #-}
```

The next sections describe the currently supported foreign languages.

11.3.1 Regular Expressions

In order to match strings against regular expressions, i.e., to check whether a string is contained in the language generated by a regular expression, one can specify regular expression similar to POSIX. The foreign regular expression code must be marked by "`regex`". Since this code is transformed into operations of the KiCS2 library `RegExp`, this library must be imported.

For instance, the following module defines a predicate to check whether a string is a valid identifier:

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=foreigncode #-}

import RegExp

isID :: String → Bool
isID = match ``regex [a-zA-Z][a-zA-Z0-9_']*''
```

11.3.2 Format Specifications

In order to format numerical and other data as strings, one can specify the desired format with foreign code marked by "`format`". In this case, one can write a format specification, similarly to the `printf` statement of C, followed by a comma-separated list of arguments. This format specification is transformed into operations of the library `Data.Format` (of package `printf`) so that it must be imported. For instance, the following program defines an operation that formats a string, an integer (with leading sign and zeros), and a float with leading sign and precision 3:

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=foreigncode #-}

import Data.Format

showSIF :: String → Int → Float → String
showSIF s i f = ``format "Name: %s | %+.5i | %+6.3f",s,i,f''

main = putStrLn $ showSIF "Curry" 42 3.14159
```

Thus, the execution of `main` will print the line

```
Name: Curry | +00042 | +3.142
```

Instead of "`format`", one can also write a format specification with `printf`. In this case, the formatted string is printed with `putStr`. Hence, we can rewrite our previous definitions as follows:

```
showSIF :: String → Int → Float → IO ()
showSIF s i f = ``printf "Name: %s | %+.5i | %+6.3f\n",s,i,f''

main = showSIF "Curry" 42 3.14159
```

11.3.3 HTML Code

The foreign language tag "`html`" introduces a notation for HTML expressions (see KiCS2 library `HTML`) with the standard HTML syntax extended by a layout rule so that closing tags can be omitted. In order to include strings computed by Curry expressions into these HTML syntax, these Curry expressions must be enclosed in curly brackets. The following example program shows its use:

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=foreigncode #-}

import HTML

htmlPage :: String → [HtmlExp]
htmlPage name = ``html
 <html>

  <head>
   <title>Simple Test

  <body>
   <h1>Hello {name}!</h1>
    <p>
```

```
      Bye!
     <p>Bye!
    <h2>{reverse name}
    Bye!''
```

If a Curry expression computes an HTML expression, i.e., it is of type `HtmlExp` instead of `String`, it can be integrated into the HTML syntax by double curly brackets. The following simple example, taken from [19], shows the use of this feature:

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=foreigncode #-}

import HTML

main :: IO HtmlForm
main = return $ form "Question" $
         ``html
             Enter a string: {{textfield tref ""}}
             <hr>
             {{button "Reverse string"   revhandler}}
             {{button "Duplicate string" duphandler}}''

 where
  tref free

  revhandler env = return $ form "Answer"
    ``html <h1>Reversed input: {reverse (env tref)}''

  duphandler env = return $ form "Answer"
    ``html
       <h1>
         Duplicated input:
         {env tref ++ env tref}''
```

11.3.4 XML Expressions

The foreign language tag "`xml`" introduces a notation for XML expressions (see KiCS2 library `XML`). The syntax is similar to the language tag "`html`", i.e., the use of the layout rule avoids closing tags and Curry expressions evaluating to strings (`String`) and XML expressions (`XmlExp`) can be included by enclosing them in curly and double curly brackets, respectively. The following example program shows its use:

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=foreigncode #-}

import HTML

import XML

main :: IO ()
main = putStrLn $ showXmlDoc $ head ``xml
```

```
<contact>
 <entry>
  <phone>+49-431-8807271
  <name>Hanus
  <first>Michael
  <email>mh@informatik.uni-kiel.de
  <email>hanus@email.uni-kiel.de

 <entry>
  <name>Smith
  <first>Bill
  <phone>+1-987-742-9388
''
```

11.4 SQL Statements

The Curry preprocessor also supports SQL statements in their standard syntax as integrated code. In order to ensure a type-safe integration of SQL statements in Curry programs, SQL queries are type-checked in order to determine their result type and ensure that the entities used in the queries are type correct with the underlying relational database. For this purpose, SQL statements are integrated code require a specification of the database model in form of entity-relationship (ER) model. From this description, a set of Curry data types are generated which are used to represent entities in the Curry program (see Section 11.4.1). The Curry preprocessor uses this information to type check the SQL statements and replace them by type-safe access methods to the database. In the following, we sketch the use of SQL statements as integrated code. A detailed description of the ideas behind this technique can be found in [24]. Currently, only SQLite databases are supported.

11.4.1 ER Specifications

The structure of the data stored in underlying database must be described as an entity-relationship model. Such a description consists of

1. a list of entities where each entity has attributes,
2. a list of relationships between entities which have cardinality constraints that must be satisfied in each valid state of the database.

Entity-relationships models are often visualized as entity-relationship diagrams (ERDs). Figure 4 shows an ERD which we use in the following examples.

Instead of requiring the use of soem graphical ER modeling tool, ERDs must be specified in textual form as a Curry data term, see also [11]. In this representation, an ERD has a name, which is also used as the module name of the generated Curry code, lists of entities and relationships:

```
data ERD = ERD String [Entity] [Relationship]
```

Each entity consists of a name and a list of attributes, where each attribute has a name, a domain, and specifications about its key and null value property:

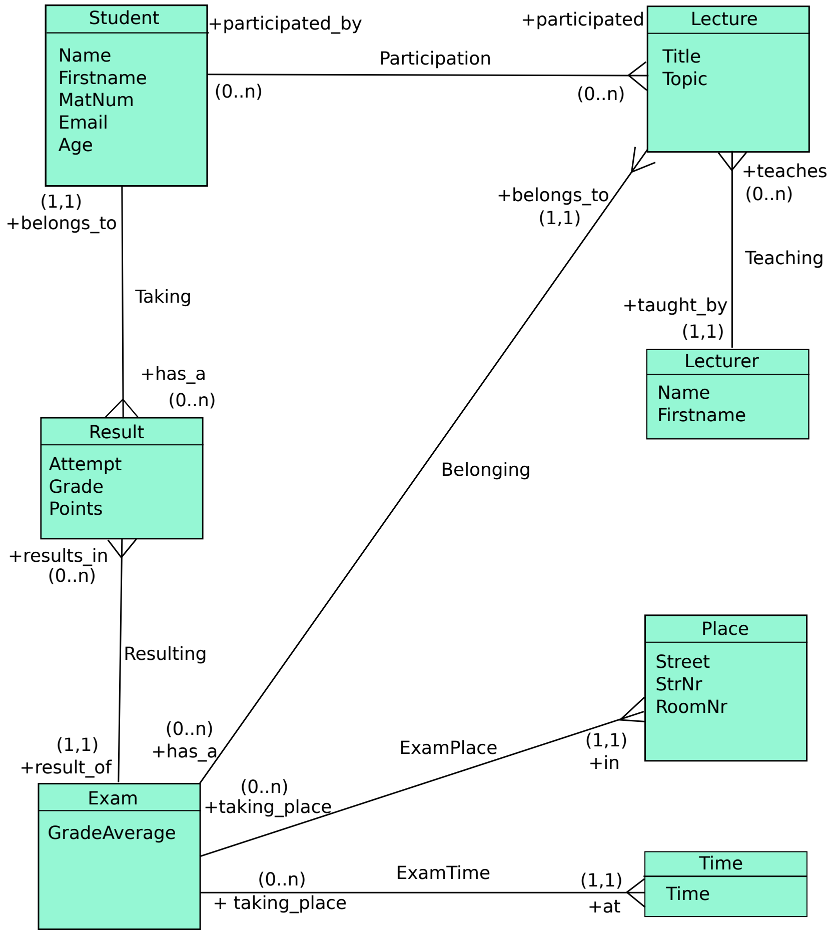

Figure 4: A simple entity-relationship diagram for university lectures [24]

```
data Entity = Entity String [Attribute]

data Attribute = Attribute String Domain Key Null

data Key = NoKey | PKey | Unique

type Null = Bool

data Domain = IntDom            (Maybe Int)
            | FloatDom          (Maybe Float)
            | CharDom           (Maybe Char)
            | StringDom         (Maybe String)
            | BoolDom           (Maybe Bool)
            | DateDom           (Maybe ClockTime)
            | UserDefined String (Maybe String)
            | KeyDom String   -- later used for foreign keys
```

Thus, each attribute is part of a primary key (`PKey`), unique (`Unique`), or not a key (`NoKey`). Furthermore, it is allowed that specific attributes can have null values, i.e., can be undefined. The domain of each attribute is one of the standard domains or some user-defined type. In the latter case, the first argument of the constructor `UserDefined` is the qualified type name used in the Curry application program. For each kind of domain, one can also have a default value (modeled by the `Maybe` type). The constructor `KeyDom` is not necessary to represent ERDs but it is internally used to transform complex ERDs into relational database schemas.

Finally, each relationship has a name and a list of connections to entities (`REnd`), where each connection has the name of the connected entity, the role name of this connection, and its cardinality as arguments:

```
data Relationship = Relationship String [REnd]

data REnd = REnd String String Cardinality

data Cardinality = Exactly Int | Between Int MaxValue

data MaxValue = Max Int | Infinite
```

The cardinality is either a fixed integer or a range between two integers (where `Infinite` as the upper bound represents an arbitrary cardinality). For instance, the simple-complex (1:n) relationship `Teaching` in Fig.4 can be represented by the term

```
Relationship "Teaching"
             [REnd "Lecturer" "taught_by" (Exactly 1),
              REnd "Lecture" "teaches"    (Between 0 Infinite)]
```

The KiCS2 library `Database.ERD` contains the ER datatypes described above. Thus, the specification of the conceptual database model must be a data term of type `Database.ERD.ERD`. Figure 5 on (page 70) shows the complete ER data term specification corresponding to the ERD of Fig. 4.

Such a data term specification should be stored in Curry program file as an (exported!) top-level operation type `ERD`. If our example term is defined as a constant in the Curry program `UniERD.curry`, then one has to use the tool "`erd2curry`" to process the ER model so that it can be used in SQL statements. This tool is invoked with the parameter "`--cdbi`", the (preferably absolute) file name of the SQLite database, and the name of the Curry program containing the ER specification. If the SQLite database file does not exist, it will be initialized by the tool. In our example, we execute the following command (provided that the tool `erd2curry` is already installed:

```
> erd2curry --db `pwd`/Uni.db --cdbi UniERD.curry
```

This initializes the SQLite database `Uni.db` and performs the following steps:

1. The ER model is transformed into tables of a relational database, i.e., the relations of the ER model are either represented by adding foreign keys to entities (in case of (0/1:1) or (0/1:n) relations) or by new entities with the corresponding relations (in case of complex (n:m) relations).

2. A new Curry module `Uni_CDBI` is generated. It contains the definitions of entities and relationships as Curry data types. Since entities are uniquely identified via a database key, each

entity definition has, in addition to its attributes, this key as the first argument. For instance, the following definitions are generated for our university ERD (among many others):

```
data StudentID = StudentID Int

data Student = Student StudentID String String Int String Int

-- Representation of n:m relationship Participation:
data Participation = Participation StudentID LectureID
```

Note that the two typed foreign key columns (`StudentID`, `LectureID`) ensures a type-safe handling of foreign-key constraints. These entity descriptions are relevant for SQL queries since some queries (e.g., those that do not project on particular database columns) return lists of such entities. Moreover, the generated module contains useful getter and setter functions for each entity. Other generated operations, like entity description and definitions of their columns, are not relevant for the programming but only used for the translation of SQL statements.

3. Finally, an *info file* `Uni_SQLCODE.info` is created. It contains information about all entities, attributes and their types, and relationships. This file is used by the SQL parser and translator of the Curry preprocessor to type check the SQL statements and generate appropriate Curry library calls.

11.4.2 SQL Statements as Integrated Code

After specifying and processing the ER model of the database, one can write SQL statements in their standard syntax as integrated code (marked by the language tag "`sql`") in Curry programs. Since the SQL translator checks the correct use of these statements against the ER model, it needs access to the generated info file `Uni_SQLCODE.info`. This can be ensured in one of the following ways:

- The path to the info file is passed as a parameter prefixed by "`--model:`" to the Curry preprocessor, e.g., by the preprocessor directive

  ```
  {-# OPTIONS_FRONTEND -F --pgmF=currypp  --optF=--model:.../Uni_SQLCode.info #-}
  ```

- The info file is placed in the same directory as the Curry source file to be processed or in one of its parent directories. The directories are searched from the directory of the source file up to its parent directories. If one of these directories contain more than one file with the name "`..._SQLCODE.info`", an error is reported.

After this preparation, one can write SQL statements in the Curry program. For instance, to retrieve all students from the database, one can define the following SQL query:

```
allStudents :: IO (SQLResult [Student])
allStudents = ``sql Select * From Student;''
```

Since the execution of database accesses might produce errors, the result of SQL statements is always of type "`SQLResult` τ", where `SQLResult` is a type synonym defined in the KiCS2 library `Database.CDBI.Connection`:

```
type SQLResult a = Either DBError a
```

This library defines also an operation

```
fromSQLResult :: SQLResult a → a
```

which returns the retrieved database value or raises a run-time error. Hence, if one does not want to check the occurrence of database errors immediately, one can also define the above query as follows:

```
allStudents :: IO [Student]
allStudents = liftM fromSQLResult ``sql Select * From Student;''
```

In order to get more control on executing the SQL statement, one can add a star character after the language tag. In this case, the SQL statement is translated into a database action, i.e., into the type `DBAction` defined in the KiCS2 library `Database.CDBI.Connection`:

```
allStudentsAction :: DBAction [Student]
allStudentsAction = ``sql* Select * From Student;''
```

Then one can put `allStudentsAction` inside a database transaction or combine it with other database actions (see `Database.CDBI.Connection` for operations for this purpose).

In order to select students with an age between 20 and 25, one can put a condition as usual:

```
youngStudents :: IO (SQLResult [Student])
youngStudents = ``sql Select * From Student
                      Where Age between 18 and 21;''
```

Usually, one wants to parameterize queries over some values computed by the context of the Curry program. Therefore, one can embed Curry expressions instead of concrete values in SQL statements by enclosing them in curly brackets:

```
studAgeBetween :: Int → Int → IO (SQLResult [Student])
studAgeBetween min max =
  ``sql Select * From Student
        Where Age between {min} and {max};''
```

Instead of retrieving complete entities (database tables), one can also project on some attributes (database columns) and one can also order them with the usual "`Order By`" clause:

```
studAgeBetween :: Int → Int → IO (SQLResult [(String,Int)])
studAgeBetween min max =
  ``sql Select Name, Age
        From Student Where Age between {min} and {max}
        Order By Name Desc;''
```

In addition to the usual SQL syntax, one can also write conditions on relationships between entities. For instance, the following code will be accepted:

```
studGoodGrades :: IO (SQLResult [(String, Float])
studGoodGrades = ``sql Select Distinct s.Name, r.Grade
                       From Student as s, Result as r
                       Where Satisfies s has_a r And r.Grade < 2.0;''
```

This query retrieves a list of pairs containing the names and grades of students having a grade better than `2.0`. This query is beyond pure SQL since it also includes a condition on the relation `has_a` specified in the ER model ("`Satisfies s has_a r`").

The complete SQL syntax supported by the Curry preprocessor is shown in Appendix B. More details about the implementation of this SQL translator can be found in [24, 30].

11.5 Default Rules

Default rules are activated by the preprocessor option "`defaultrules`". In this case, one can add to each top-level operation a default rule. A default rule for a function f is defined as a rule defining the operation "f`'default`" (this mechanism avoids any language extension for default rules). A default rule is applied only if no "standard" rule is applicable, either because the left-hand sides' pattern do not match or the conditions are not satisfiable. The idea and detailed semantics of default rules are described in [5].

As a simple example, the following program defines a lookup operation in association lists by a functional pattern. The default rule is applied only if there is no appropriate key in the association list (the role of the `import` declarations is discussed below):

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=defaultrules #-}

import Control.SetFunctions

mlookup key (_ ++ [(key,value)] ++ _) = Just value
mlookup'default _   _                 = Nothing
```

Default rules are often a good replacement for "negation as failure" used in logic programming. For instance, the following program defines a solution to the n-queens puzzle, where the default rule is useful since it is easier to characterize the unsafe positions of the queens on the chessboard (see the first rule of `safe`):

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=defaultrules #-}

import Control.SetFunctions

-- Some permutation of a list of elements:
perm :: [a] → [a]
perm []     = []
perm (x:xs) = ndinsert (perm xs)
 where ndinsert ys     = x : ys
       ndinsert (y:ys) = y : ndinsert ys

-- A placement is safe if two queens are not in a same diagonal:
safe :: [Int] → [Int]
safe (_++[x]++ys++[z]++_) | abs (x-z) == length ys + 1 = failed
safe'default xs = xs

-- A solution to the n-queens puzzle is a safe permutation:
queens :: Int → [Int]
queens n = safe (permute [1..n])
```

Important notes:

1. The implementation of default rules is based on set functions (implemented by the module `Control.SetFunctions`). Therefore, the package `setfunctions` should be installed as a dependency. This can easily be done by executing

   ```
   > cypm add setfunctions
   ```

   before compiling a program containing default rules with the Curry preprocessor.

2. Default rules can only be added to operations defined at the top-level (i.e., not to locally defined operations). A reason for this restriction is that default rules are applied after searching for all possibilities to apply a previous standard rule. With local definitions, the precise scope of the "previous" search is difficult to define.

11.6 Contracts

Contracts are annotations in Curry program to specify the intended meaning and use of operations by other operations or predicates expressed in Curry. The idea of using contracts for the development of reliable software is discussed in [3]. The Curry preprocessor supports dynamic contract checking by transforming contracts, i.e., specifications and pre-/postconditions, into assertions that are checked during the execution of a program. If some contract is violated, the program terminates with an error.

The transformation of contracts into assertions is described in [3]. Note that only strict assertion checking is supported at the moment. Strict assertion checking might change the operational behavior of the program. The notation of contracts is defined in [3]. To transform such contracts into assertions, one has to use the option "`contracts`" for the preprocessor.

As a concrete example, consider an implementation of quicksort with a postcondition and a specification (where the code for `sorted` and `perm` is not shown here):

```
{-# OPTIONS_FRONTEND -F --pgmF=currypp --optF=contracts #-}

...

-- Trivial precondition:
sort'pre xs = length xs >= 0

-- Postcondition: input and output lists should have the same length
sort'post xs ys = length xs == length ys

-- Specification:
-- A correct result is a permutation of the input which is sorted.
sort'spec :: [Int] → [Int]
sort'spec xs | ys == perm xs && sorted ys = ys  where ys free

-- A buggy implementation of quicksort:
sort :: [Int] → [Int]
sort []     = []
sort (x:xs) = sort (filter (<x) xs) ++ [x] ++ sort (filter (>x) xs)
```

If this program is executed, the generated assertions report a contract violation for some inputs:

```
Quicksort> sort [3,1,4,2,1]
Postcondition of 'sort' (module Quicksort, line 27) violated for:
[1,2,1]  →  [1,2]

ERROR: Execution aborted due to contract violation!
```

Important note: The implementation of default rules is based on set functions (implemented by the module `Control.SetFunctions`) and the auxiliary package to check contracts at run time. Therefore, the packages `setfunctions` and `contracts` should be installed as dependencies. This can easily done by executing

```
> cypm add setfunctions
> cypm add contracts
```

before compiling a program containing contracts with the Curry preprocessor.

```
ERD "Uni"
  [Entity "Student"
          [Attribute "Name" (StringDom Nothing) NoKey False,
           Attribute "Firstname" (StringDom Nothing) NoKey False,
           Attribute "MatNum" (IntDom Nothing) Unique False,
           Attribute "Email" (StringDom Nothing) Unique False,
           Attribute "Age" (IntDom Nothing) NoKey True],
   Entity "Lecture"
          [Attribute "Title" (StringDom Nothing) NoKey False,
           Attribute "Topic" (StringDom Nothing) NoKey True],
   Entity "Lecturer"
          [Attribute "Name" (StringDom Nothing) NoKey False,
           Attribute "Firstname" (StringDom Nothing) NoKey False],
   Entity "Place"
          [Attribute "Street" (StringDom Nothing) NoKey False,
           Attribute "StrNr" (IntDom Nothing) NoKey False,
           Attribute "RoomNr" (IntDom Nothing) NoKey False],
   Entity "Time"
          [Attribute "Time" (DateDom Nothing) Unique False],
   Entity "Exam"
          [Attribute "GradeAverage" (FloatDom Nothing) NoKey True],
   Entity "Result"
          [Attribute "Attempt" (IntDom Nothing) NoKey False,
           Attribute "Grade" (FloatDom Nothing) NoKey True,
           Attribute "Points" (IntDom Nothing) NoKey True]]
  [Relationship "Teaching"
                [REnd "Lecturer" "taught_by" (Exactly 1),
                 REnd "Lecture" "teaches" (Between 0 Infinite)],
   Relationship "Participation"
                [REnd "Student" "participated_by" (Between 0 Infinite),
                 REnd "Lecture" "participates" (Between 0 Infinite)],
   Relationship "Taking"
                [REnd "Result" "has_a" (Between 0 Infinite),
                 REnd "Student" "belongs_to" (Exactly 1)],
   Relationship "Resulting"
                [REnd "Exam" "result_of" (Exactly 1),
                 REnd "Result" "results_in" (Between 0 Infinite)],
   Relationship "Belonging"
                [REnd "Exam" "has_a" (Between 0 Infinite),
                 REnd "Lecture" "belongs_to" (Exactly 1)],
   Relationship "ExamDate"
                [REnd "Exam" "taking_place" (Between 0 Infinite),
                 REnd "Time" "at" (Exactly 1)],
   Relationship "ExamPlace"
                [REnd "Exam" "taking_place" (Between 0 Infinite),
                 REnd "Place" "in" (Exactly 1)]]
```

Figure 5: The ER data term specification of Fig. 4

12 runcurry: Running Curry Programs

`runcurry` is a simple tool to support the execution of Curry programs without explicitly invoking the interactive environment. Hence, it can be useful to write short scripts in Curry intended for direct execution. The Curry program must always contain the definition of an operation `main` of type `IO ()`. The execution of the program consists of the evaluation of this operation.

12.1 Installation

The implementation of `runcurry` is a package managed by the Curry Package Manager CPM. Thus, to install the newest version of `runcurry`, use the following commands:

```
> cypm update
> cypm install runcurry
```

This downloads the newest package, compiles it, and places the executable `runcurry` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to use `runcurry` as described below.

12.2 Using runcurry

Basically, the command `runcurry` supports three modes of operation:

- One can execute a Curry program whose file name is provided as an argument when `runcurry` is called. In this case, the suffix ("`.curry`" or "`.lcurry`") must be present and cannot be dropped. One can write additional commands for the interactive environment, typically settings of some options, before the Curry program name. All arguments after the Curry program name are passed as run-time arguments. For instance, consider the following program stored in the file `ShowArgs.curry`:

  ```
  import System(getArgs)

  main = getArgs >>= print
  ```

  This program can be executed by the shell command

  ```
  > runcurry ShowArgs.curry Hello World!
  ```

  which produces the output

  ```
  ["Hello","World!"]
  ```

- One can also execute a Curry program whose program text comes from the standard input. Thus, one can either "pipe" the program text into this command or type the program text on the keyboard. For instance, if we type

  ```
  > runcurry
  main = putStr . unlines . map show . take 8 $ [1..]
  ```

  (followed by the end-of-file marker Ctrl-D), the output

```
1
2
3
4
5
6
7
8
```

is produced.

- One can also write the program text in a script file to be executed like a shell script. In this case, the script must start with the line

  ```
  #!/usr/bin/env runcurry
  ```

  followed by the source text of the Curry program. If the name of the script file has a suffix, it must be different from `.curry` and `.lcurry`.

  For instance, we can write a simple Curry script to count the number of code lines in a Curry program by removing all blank and comment lines and counting the remaining lines:

  ```
  #!/usr/bin/env runcurry

  import Char(isSpace)
  import System(getArgs)

  -- count number of program lines in a file:
  countCLines :: String → IO Int
  countCLines f =
    readFile f >>=
    return . length . filter (not . isEmptyLine) . map stripSpaces . lines
   where
    stripSpaces = reverse . dropWhile isSpace . reverse . dropWhile isSpace

    isEmptyLine []        = True
    isEmptyLine [_]       = False
    isEmptyLine (c1:c2:_) = c1=='-' && c2=='-'

  -- The main program reads Curry file names from arguments:
  main = do
    args <- getArgs
    mapIO_ (\f → do ls <- countCLines f
                    putStrLn $ "Stripped lines of file "++f++": " ++ show ls)
           args
  ```

  If this script is stored in the (executable) file "`codelines.sh`", we can count the code lines of the file `Prog.curry` by the shell command

  ```
  > ./codelines.sh Prog.curry
  ```

When this command is executed, the command `runcurry` compiles the program and evaluates the expression `main`. Since the compilation might take some time in more complex scripts, one can also save the result of the compilation in a binary file. To obtain this behavior, one has to insert the line

```
#jit
```

in the script file, e.g., in the second line. With this option, a binary of the compiled program is saved (in the same directory as the script). Now, when the same script is executed the next time, the stored binary file is executed (provided that it is still newer than the script file itself, otherwise it will be recompiled). This feature combines easy scripting with Curry together with fast execution.

13 CASS: A Generic Curry Analysis Server System

CASS (Curry Analysis Server System) is a tool for the analysis of Curry programs. CASS is generic so that various kinds of analyses (e.g., groundness, non-determinism, demanded arguments) can be easily integrated into CASS. In order to analyze larger applications consisting of dozens or hundreds of modules, CASS supports a modular and incremental analysis of programs. Moreover, it can be used by different programming tools, like documentation generators, analysis environments, program optimizers, as well as Eclipse-based development environments. For this purpose, CASS can also be invoked as a server system to get a language-independent access to its functionality. CASS is completely implemented Curry as a master/worker architecture to exploit parallel or distributed execution environments. The general design and architecture of CASS is described in [25]. In the following, CASS is presented from a perspective of a programmer who is interested to analyze Curry programs.

13.1 Installation

The current implementation of CASS is a package managed by the Curry Package Manager CPM. Thus, to install the newest version of CASS, use the following commands:

```
> cypm update
> cypm install cass
```

This downloads the newest package, compiles it, and places the executable `cass` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to execute CASS as described below.

13.2 Using CASS to Analyze Programs

CASS is intended to analyze various operational properties of Curry programs. Currently, it contains more than a dozen program analyses for various properties. Since most of these analyses are based on abstract interpretations, they usually approximate program properties. To see the list of all available analyses, use the help option of CASS:

```
> cass -h
Usage: ...
⋮
Registered analyses names:
...
Demand          : Demanded arguments
Deterministic   : Deterministic operations
⋮
```

More information about the meaning of the various analyses can be obtained by adding the short name of the analysis:

```
> cass -h Deterministic
...
```

For instance, consider the following Curry module `Rev.curry`:

```
append :: [a] → [a] → [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys

rev :: [a] → [a]
rev []     = []
rev (x:xs) = append (rev xs) [x]

main :: Int → Int → [Int]
main x y = rev [x .. y]
```

CASS supports three different usage modes to analyze this program.

13.2.1 Batch Mode

In the batch mode, CASS is started as a separate application via the shell command `cass`, where the analysis name and the name of the module to be analyzed must be provided:[10]

```
> cass Demand Rev
append : demanded arguments: 1
main : demanded arguments: 1,2
rev : demanded arguments: 1
```

The `Demand` analysis shows the list of argument positions (e.g., 1 for the first argument) which are demanded in order to reduce an application of the operation to some constructor-rooted value. Here we can see that both arguments of `main` are demanded whereas only the first argument of `append` is demanded. This information could be used in a Curry compiler to produce more efficient target code.

The batch mode is useful to test a new analysis and get the information in human-readable form so that one can experiment with different abstractions or analysis methods.

13.2.2 API Mode

The API mode is intended to use analysis information in some application implemented in Curry. Since CASS is implemented in Curry, one can import the modules of the CASS implementation and use the CASS interface operations to start an analysis and use the computed results. For instance, CASS provides an operation (defined in the module `CASS.Server`)

```
analyzeGeneric :: Analysis a → String → IO (Either (ProgInfo a) String)
```

to apply an analysis (first argument) to some module (whose name is given in the second argument). The result is either the analysis information computed for this module or an error message in case of some execution error.

In order to use CASS via the API mode in a Curry program, one has to use the package `cass` by the Curry package manager CPM (the subsequent explanation assumes familiarity with the basic features of CPM):

[10] More output is generated when the property `debugLevel` is changed in the configuration file `.curryanalysisrc` which is installed in the user's home directory when CASS is started for the first time.

1. Add the dependency on package `cass` and also on package `cass-analysis`, which contains some base definitions, in the package specification file `package.json`.

2. Install these dependencies by "`cypm install`".

Then you can import in your application the modules provided by CASS.

The module `Analysis.ProgInfo` (from package `cass-analysis`) contains operations to access the analysis information computed by CASS. For instance, the operation

```
lookupProgInfo:: QName → ProgInfo a → Maybe a
```

returns the information about a given qualified name in the analysis information, if it exists. As a simple example, consider the demand analysis which is implemented in the module `Analysis.Demandedness` by the following operation:

```
demandAnalysis :: Analysis DemandedArgs
```

`DemendedArgs` is just a type synonym for `[Int]`. We can use this analysis in the following simple program:

```
import CASS.Server           ( analyzeGeneric )
import Analysis.ProgInfo     ( lookupProgInfo )
import Analysis.Demandedness ( demandAnalysis )

demandedArgumentsOf :: String → String → IO [Int]
demandedArgumentsOf modname fname = do
  deminfo <- analyzeGeneric demandAnalysis modname >>= return . either id error
  return $ maybe [] id (lookupProgInfo (modname,fname) deminfo)
```

Of course, in a realistic program, the program analysis is performed only once and the computed information `deminfo` is passed around to access it several times. Nevertheless, we can use this simple program to compute the demanded arguments of `Rev.main`:

```
...> demandedArgumentsOf "Rev" "main"
[1,2]
```

13.2.3 Server Mode

The server mode of CASS can be used in an application implemented in some language that does not have a direct interface to Curry. In this case, one can connect to CASS via some socket using a simple communication protocol that is specified in the file `Protocol.txt` (in package `cass`) and sketched below.

To start CASS in the server mode, one has to execute the command

```
> cass --server [ -p <port> ]
```

where an optional port number for the communication can be provided. Otherwise, a free port number is chosen and shown. In the server mode, CASS understands the following commands:

```
GetAnalysis
SetCurryPath <dir1>:<dir2>:...
AnalyzeModule          <analysis name> <output type> <module name>
```

```
AnalyzeInterface       <analysis name> <output type> <module name>
AnalyzeFunction        <analysis name> <output type> <module name> <function name>
AnalyzeDataConstructor <analysis name> <output type> <module name> <constructor name>
AnalyzeTypeConstructor <analysis name> <output type> <module name> <type name>
StopServer
```

The output type can be `Text`, `CurryTerm`, or `XML`. The answer to each request can have two formats:

```
error <error message>
```

if an execution error occured, or

```
ok <n>
<result text>
```

where `<n>` is the number of lines of the result text. For instance, the answer to the command `GetAnalysis` is a list of all available analyses. The list has the form

```
<analysis name> <output type>
```

For instance, a communication could be:

```
> GetAnalysis
< ok 5
< Deterministic CurryTerm
< Deterministic Text
< Deterministic XML
< HigherOrder   CurryTerm
< DependsOn     CurryTerm
```

The command `SetCurryPath` instructs CASS to use the given directories to search for modules to be analyzed. This is necessary since the CASS server might be started in a different location than its client.

Complete modules are analyzed by `AnalyzeModule`, whereas `AnalyzeInterface` returns only the analysis information of exported entities. Furthermore, the analysis results of individual functions, data or type constructors are returned with the remaining analysis commands. Finally, `StopServer` terminates the CASS server.

For instance, if we start CASS by

```
> cass --server -p 12345
```

we can communicate with CASS as follows (user inputs are prefixed by "`>`");

```
> telnet localhost 12345
Connected to localhost.
> GetAnalysis
ok 57
Overlapping XML
Overlapping CurryTerm
Overlapping Text
Deterministic XML
...
> AnalyzeModule Demand Text Rev
```

```
ok 3
append : demanded arguments: 1
main : demanded arguments: 1,2
rev : demanded arguments: 1
> AnalyzeModule Demand CurryTerm Rev
ok 1
[(("Rev","append"),"demanded arguments: 1"),(("Rev","main"),"demanded arguments: 1,2"),(("Rev","re
> AnalyzeModule Demand XML Rev
ok 19
<?xml version="1.0" standalone="yes"?>

<results>
  <operation>
    <module>Rev</module>
    <name>append</name>
    <result>demanded arguments: 1</result>
  </operation>
  <operation>
    <module>Rev</module>
    <name>main</name>
    <result>demanded arguments: 1,2</result>
  </operation>
  <operation>
    <module>Rev</module>
    <name>rev</name>
    <result>demanded arguments: 1</result>
  </operation>
</results>
> StopServer
ok 0
Connection closed by foreign host.
```

13.3 Implementing Program Analyses

This section explains the implementation of program analyses available in CASS. Since CASS is implemented in Curry, a program analysis must also be implemented in Curry and added to the source code of CASS. Therefore, one has to download the source code which is easily done by the command

```
> cypm checkout cass
```

This downloads the most recent version of CASS as a Curry package into the directory `cass`.

Each program analysis accessible by CASS must be registered in the CASS module `CASS.Registry`. Such an analysis must contain an operation of type

```
Analysis a
```

where "`a`" denotes the type of analysis results. Furthermore, the analysis must also contain a "show" operation of type

```
AOutFormat → a → String
```

intended to show the analysis results in various formats. The type `AOutFormat` is defined in module `Analysis.Types` of package `cass-analysis` as

```
data AOutFormat = AText | ANote
```

It is intended to specify the desired kind of output, e.g., `AText` for a longer standard textual representation or `ANote` for a short note (e.g., in the Curry Browser).

Thus, in order to add a new analysis to CASS, one has to do the following steps:

1. Implement a corresponding analysis operation and show operation.
2. Registering it in the module `CASS.Registry` (in the constant `registeredAnalysis`).
3. Compile/install the modified CASS implementation.

In the following, we explain these steps by some examples. For instance, the `Overlapping` analysis should indicate whether a Curry operation is defined by overlapping rules. This analysis can be implemented as a function

```
overlapAnalysis :: Analysis Bool
```

so that the analysis result is `False` if the analyzed operation is not defined by overlapping rules.

In general, an analysis is implemented as a mapping from Curry operations, represented in FlatCurry, into the analysis result. Hence, to implement the `Overlapping` analysis, we define the following operation on function declarations in FlatCurry format:

```
import FlatCurry.Types
...
isOverlappingFunction :: FuncDecl → Bool
isOverlappingFunction (Func _ _ _ _ (Rule _ e))   = orInExpr e
isOverlappingFunction (Func f _ _ _ (External _)) = f == ("Prelude","?")

-- Check an expression for occurrences of Or:
orInExpr :: Expr → Bool
orInExpr (Var _)       = False
orInExpr (Lit _)       = False
orInExpr (Comb _ f es) = f == ("Prelude","?") || any orInExpr es
orInExpr (Free _ e)    = orInExpr e
orInExpr (Let bs e)    = any orInExpr (map snd bs) || orInExpr e
orInExpr (Or _ _)      = True
orInExpr (Case _ e bs) = orInExpr e || any orInBranch bs
 where orInBranch (Branch _ be) = orInExpr be
orInExpr (Typed e _)   = orInExpr e
```

In order to support the inclusion of different kinds of analyses in CASS, CASS offers several constructor operations for the abstract type "`Analysis a`" (which is defined in module `Analysis.Types`). Each analysis has a name provided as a first argument to these constructors. The name is used to store the analysis information persistently and to pass specific analysis tasks to analysis workers. For instance, a simple function analysis which depends only on a given function definition can be defined by the analysis constructor

```
simpleFuncAnalysis :: String → (FuncDecl → a) → Analysis a
```

The arguments are the analysis name and the actual analysis function. Hence, the "overlapping rules" analysis can be specified as

```
import Analysis.Types
...
overlapAnalysis :: Analysis Bool
overlapAnalysis = simpleFuncAnalysis "Overlapping" isOverlappingFunction
```

In order to integrate this analysis into CASS, we also have to define an operation to show the analysis results in a human-readable form:

```
showOverlap :: AOutFormat → Bool → String
showOverlap _     True  = "overlapping"
showOverlap AText False = "non-overlapping"
showOverlap ANote False = ""
```

Here, the typical case of non-overlapping rules is not printed in case of short notes.

Now we have all elements available in order to add this analysis to CASS. To support this easily, there is an operation

```
cassAnalysis :: (Read a, Show a, Eq a)
             => String → Analysis a → (AOutFormat → a → String)
              → RegisteredAnalysis
```

to transform an analysis with some title, an analysis operation, and a "show" operation into an analysis ready to be registered in CASS. The actually registered analyses are specified by the constant

```
registeredAnalysis :: [RegisteredAnalysis]
```

defined in module `CASS.Registry`. Hence, the `Overlapping` can be integrated into CASS by adding it to the definition of `registeredAnalysis`, e.g.,

```
registeredAnalysis :: [RegisteredAnalysis]
registeredAnalysis =
  [
  ⋮
  cassAnalysis "Overlapping rules" overlapAnalysis showOverlap
  ⋮
  ]
```

As a final step, we have to compile and install this extended version of CASS by executing

```
> cypm install
```

in the downloaded package. After this step, one can executed

```
> cass --help
```

to check whether the `Overlapping` analysis occurs in the list of registered analyses names.

To show an example of a more complex kind of analysis, we consider a determinism analysis.

Such an analysis could be based on an abstract domain described by the data type

```
data Deterministic = NDet | Det
```

Here, `Det` is interpreted as "the operation always evaluates in a deterministic manner on ground constructor terms." However, `NDet` is interpreted as "the operation *might* evaluate in different ways for given ground constructor terms." The apparent imprecision is due to the approximation of the analysis. For instance, if the function `f` is defined by overlapping rules and the function `g` *might* call `f`, then `g` is judged as non-deterministic (since it is generally undecidable whether `f` is actually called by `g` in some run of the program).

The determinism analysis requires to examine the current function as well as all directly or indirectly called functions for overlapping rules. Due to recursive function definitions, this analysis cannot be done in one shot for a given function—it requires a fixpoint computation. CASS provides such fixpoint computations and simplifies its implementation by requiring only the implementation of an operation of type

```
FuncDecl → [(QName,a)] → a
```

where "`a`" denotes the type of abstract values. The second argument of type `[(QName,a)]` represents the currently known analysis values for the functions *directly* used in this function declaration. Hence, in the implementation one can assume that the analysis results of all functions occurring in the definition of the function to be analyzed are already known, although they will be approximated by a fixpoint computation performed by CASS. Technically, the abstract values must be a domain with some bottom element and the analysis operation must be monotone. Since this is not checked by CASS, we omit these details.

In our example, the determinism analysis can be implemented by the following operation:

```
detFunc :: FuncDecl → [(QName,Deterministic)] → Deterministic
detFunc (Func f _ _ _ (External _)) _           = f == ("Prelude","?")
detFunc (Func f _ _ _ (Rule _ e))   calledFuncs =
  if orInExpr e || freeVarInExpr e || any (==NDet) (map snd calledFuncs)
    then NDet
    else Det
```

Thus, it computes the abstract value `NDet` if the function itself is defined by overlapping rules or contains free variables that might cause non-deterministic guessing (we omit the definition of `freeVarInExpr` since it is quite similar to `orInExpr`), or if it depends on some non-deterministic function.

To support the integration of such fixpoint analyses in CASS, there exists the following analysis constructor:

```
dependencyFuncAnalysis :: String → a → (FuncDecl → [(QName,a)] → a)
                        → Analysis a
```

Here, the second argument specifies the start value of the fixpoint analysis, i.e., the bottom element of the abstract domain. Hence, the complete determinism analysis can be specified as

```
detAnalysis :: Analysis Deterministic
detAnalysis = dependencyFuncAnalysis "Deterministic" Det detFunc
```

In order to register this analysis, we define a show function

```
showDet :: AOutFormat  → Deterministic  → String
showDet _      NDet = "non-deterministic"
showDet AText Det  = "deterministic"
showDet ANote Det  = ""
```

extend the definiton of `registeredAnalysis` by the line

```
cassAnalysis "Deterministic operations" detAnalysis showDet
```

and compile and install the package.

This simple definition is sufficient to execute this analysis with CASS, since the analysis system takes care of computing fixpoints, calling the analysis functions with appropriate values, analyzing imported modules, caching analysis results, etc. The actual analysis time depends on the size of modules and their imports, the size of the dependencies, and the number of fixpoint iterations (which depends also on the depth of the abstract domain).[11] Beyond the analysis time, it is also important that the analysis terminates, which is not ensured in general fixpoint computations. Termination can be achieved by using an abstract domain with finitely many values and defining the analysis function so that it is monotone w.r.t. some ordering on the abstract values.

[11] CASS supports different methods to compute fixpoints, see the property `fixpoint` in the configuration file `.curryanalysisrc` which is installed in the user's home directory when CASS is started for the first time. This property can also be set in the command to invoke CASS.

14 ERD2Curry: A Tool to Generate Programs from ER Specifications

ERD2Curry is a tool to generate Curry code to access and manipulate data persistently stored in relational databases. The Curry code is generated from a description of the logical model of the database in form of an entity relationship diagram. The idea of this tool is described in detail in [11]. Thus, we describe only the basic steps to use this tool.

14.1 Installation

The current implementation of ERD2Curry is a package managed by the Curry Package Manager CPM (see also Section 6). Thus, to install the newest version of ERD2Curry, use the following commands:

```
> cypm update
> cypm install ertools
```

This downloads the newest package, compiles it, and places the executable `erd2curry` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to execute ERD2Curry as described below.

14.2 Basic Usage

If one creates an entity relationship diagram (ERD) with the Umbrello UML Modeller, one has to store its XML description in XMI format (as offered by Umbrello) in a file, e.g., "`myerd.xmi`". This description can be compiled into a Curry program by the command

```
erd2curry -x myerd.xmi
```

If `MyData` is the name of the ERD, the Curry program file "`MyData.curry`" is generated containing all the necessary database access code as described in [11]. In addition to the generated Curry program file, two auxiliary program files `ERDGeneric.curry` and `KeyDatabase.curry` are created in the same directory.

If one does not want to use the Umbrello UML Modeller, which might be the preferred method since the interface to the Umbrello UML Modeller is no longer actively supported, one can also define an ERD in a Curry program as a (exported!) top-level operation of type `ERD` (w.r.t. the type definition given in the library *kics2home*`/lib/Database/ERD.curry`). The directory `examples` in the package `ertools`[12] contains two examples for such ERD program files:

`BlogERD.curry`: This is a simple ERD model for a blog with entries, comments, and tags.

`UniERD.curry`: This is an ERD model for university lectures as presented in the paper [11].

Figure 6 shows the ER specification stored in the Curry program file "`BlogERD.curry`". This ER specification can be compiled into a Curry program by the command

```
erd2curry BlogERD.curry
```

[12] If you installed ERD2Curry as described above, the downloaded `ertools` package is located in the directory `$HOME/.cpm/bin_packages/ertools`.

```
import Database.ERD

blogERD :: ERD
blogERD =
 ERD "Blog"
     [Entity "Entry"
          [Attribute "Title"  (StringDom Nothing) Unique False,
           Attribute "Text"   (StringDom Nothing) NoKey  False,
           Attribute "Author" (StringDom Nothing) NoKey  False,
           Attribute "Date"   (DateDom   Nothing) NoKey  False],
      Entity "Comment"
        [Attribute "Text"   (StringDom Nothing) NoKey False,
         Attribute "Author" (StringDom Nothing) NoKey False,
         Attribute "Date"   (DateDom   Nothing) NoKey False],
      Entity "Tag"
        [Attribute "Name" (StringDom Nothing) Unique False]
     ]
     [Relationship "Commenting"
       [REnd "Entry"   "commentsOn"    (Exactly 1),
        REnd "Comment" "isCommentedBy" (Between 0 Infinite)],
      Relationship "Tagging"
       [REnd "Entry" "tags" (Between 0 Infinite),
        REnd "Tag" "tagged" (Between 0 Infinite)]
     ]
```

Figure 6: The Curry program `BlogERD.curry`

There is also the possibility to visualize an ER specification as a graph with the graph visualization program `dotty` (for this purpose, it might be necessary to adapt the definition of `dotviewcommand` in your "`.kics2rc`" file, see Section 2.7, according to your local environment). The visualization can be performed by the command

```
erd2curry -v BlogERD.curry
```

15 Spicey: An ER-based Web Framework

Spicey is a framework to support the implementation of web-based systems in Curry. Spicey generates an initial implementation from an entity-relationship (ER) description of the underlying data. The generated implementation contains operations to create and manipulate entities of the data model, supports authentication, authorization, session handling, and the composition of individual operations to user processes. Furthermore, the implementation ensures the consistency of the database w.r.t. the data dependencies specified in the ER model, i.e., updates initiated by the user cannot lead to an inconsistent state of the database.

15.1 Installation

The actual implementation of Spicey is a package managed by the Curry Package Manager CPM. Thus, to install the newest version of Spicey, use the following commands:

```
> cypm update
> cypm install spicey
```

This downloads the newest package, compiles it, and places the executable `spiceup` into the directory `$HOME/.cpm/bin`. Hence it is recommended to add this directory to your path in order to execute Spicey as described below.

15.2 Basic usage

The idea of this tool, which is part of the distribution of KiCS2, is described in detail in [23]. Thus, we summarize only the basic steps to use this tool in order to generate a web application.

First, one has to create a textual description of the entity-relationship model in a Curry program file as an (exported!) top-level operation type `ERD` (w.r.t. the type definitions defined in the module `Database.ERD` of the package `cdbi`) and store it in some program file, e.g., "`MyERD.curry`". The directory `examples` in the package `spicey`[13] contains two examples for such ERD program files:

`BlogERD.curry`: This is a simple ER model for a blog with entries, comments, and tags, as presented in the paper [23].

`UniERD.curry`: This is an ER model for university lectures as presented in the paper [11].

Then you can generate the sources of your web application by the command

```
> spiceup MyERD.curry
```

with the ERD program as a parameter. You can also provide a file name for the SQLite3 database used by the application generated by Spicey, e.g.,

```
> spiceup --db MyData.db MyERD.curry
```

If the parameter "`--db DBFILE`" is not provided, then `DBDFILE` is set to the default name "*ERD*`.db`" (where *ERD* is the name of the specified ER model). Since this specification will be used in the *generated* web programs, a relative database file name will be relative to the place where the web

[13] If you installed Spicey as described above, the downloaded `spicey` package is located in the directory `$HOME/.cpm/app_packages/spicey`.

programs are stored. In order to avoid such confusion, it might be better to specify an absolute path name for the database file. This path could also be set in the definition of the constant `sqliteDBFile` in the generated Curry program `Model/`*ERD*`.curry`.

Spicey generates the web application as a Curry package in a new directory. Thus, change into this directory (e.g., `cd` *ERD*) and install all required packages by the command

```
> make install
```

The generated file `README.txt` contains some information about the generated project structure. One can compile the generated programs by

```
> make compile
```

In order to generate the executable web application, configure the generated `Makefile` by adapting the variable `WEBSERVERDIR` to the location where the compiled cgi programs should be stored, and run

```
> make deploy
```

After the successful compilation and deployment of all files, the application is executable in a web browser by selecting the URL `<URL of web dir>/spicey.cgi`.

15.3 Further remarks

The application generated by Spicey is a schematic initial implementation. It provides an appropriate basic programming structure but it can be extended in various ways. In particular, one can also use embedded SQL statements (see [24] for details) when further developing the Curry code, since the underlying database access operations are generated with the `cdbi` package. The syntax and use of such embedded SQL statements is sketched in [24] and described in the Curry preprocessor.

16 Technical Problems

One can implement distributed systems with KiCS2 by the use of the library `Network.NamedSocket` (available in package `cpns`) that supports a socket communication with symbolic names rather than natural numbers. For instance, this library is the basis of programming dynamic web pages with the Curry package `html`. However, it might be possible that some technical problems arise due to the use of named sockets. Therefore, this section gives some information about the technical requirements of KiCS2 and how to solve problems due to these requirements.

There is one fixed port that is used by the implementation of KiCS2:

Port 8769: This port is used by the **Curry Port Name Server** (CPNS) to implement symbolic names for named sockets in Curry. If some other process uses this port on the machine, the distribution facilities defined in the module `Network.NamedSocket` cannot be used.

If these features do not work, you can try to find out whether this port is in use by the shell command "`netstat -a | grep 8769`" (or similar).

The CPNS is implemented as a demon listening on its port 8769 in order to serve requests about registering a new symbolic name for a named socket or asking the physical port number of an registered named socket. The demon will be automatically started for the first time on a machine when a user runs a program using named sockets. It can also be manually started and terminated by the command `curry-cpnsd` (which is available by installing the package `cpns`, e.g., by the command "`cypm install cpnsd`") If the demon is already running, the command "`curry-cpnsd start`" does nothing (so it can be always executed before invoking a Curry program using named sockets).

If you detect any further technical problem, please write to

`kics2@curry-language.org`

References

[1] S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOP-STR'05)*, pages 6–22. Springer LNCS 3901, 2005.

[2] S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.

[3] S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012.

[4] S. Antoy and M. Hanus. From boolean equalities to constraints. In *Proceedings of the 25th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2015)*, pages 73–88. Springer LNCS 9527, 2015.

[5] S. Antoy and M. Hanus. Default rules for Curry. *Theory and Practice of Logic Programming*, 17(2):121–147, 2017.

[6] S. Antoy and M. Hanus. Transforming boolean equalities into constraints. *Formal Aspects of Computing*, 29(3):475–494, 2017.

[7] S. Antoy and M. Hanus. Equivalence checking of non-deterministic operations. In *Proc. of the 14th International Symposium on Functional and Logic Programming (FLOPS 2018)*, pages 149–165. Springer LNCS 10818, 2018.

[8] S. Antoy, M. Hanus, and S. Libby. Proving non-deterministic computations in Agda. In *Proc. of the 24th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2016)*, volume 234 of *Electronic Proceedings in Theoretical Computer Science*, pages 180–195. Open Publishing Association, 2017.

[9] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.

[10] B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

[11] B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

[12] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

[13] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing equational constraints in a functional language. In *Proc. of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL 2013)*, pages 125–140. Springer LNCS 7752, 2013.

[14] J. Christiansen and S. Fischer. EasyCheck - test data for free. In *Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, pages 322–336. Springer LNCS 4989, 2008.

[15] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming (ICFP'00)*, pages 268–279. ACM Press, 2000.

[16] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[17] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[18] M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[19] M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[20] M. Hanus. A generic analysis environment for declarative programs. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 43–48. ACM Press, 2005.

[21] M. Hanus. CurryBrowser: A generic analysis environment for Curry programs. In *Proc. of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE'06)*, pages 61–74, 2006.

[22] M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.

[23] M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. *Theory and Practice of Logic Programming*, 14(3):269–291, 2014.

[24] M. Hanus and J. Krone. A typeful integration of SQL into Curry. In *Proceedings of the 24th International Workshop on Functional and (Constraint) Logic Programming*, volume 234 of *Electronic Proceedings in Theoretical Computer Science*, pages 104–119. Open Publishing Association, 2017.

[25] M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014.

[26] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.

[27] M. Hanus and F. Teegen. Adding `Data` to Curry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*, pages 230–246. Springer LNCS 12057, 2020.

[28] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-language.org, 2016.

[29] T. Johnsson. Lambda lifting: Transforming programs to recursive functions. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer LNCS 201, 1985.

[30] J. Krone. Integration of SQL into Curry. Master's thesis, University of Kiel, 2015.

[31] U. Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP'08)*, pages 230–266. Springer, 2009.

[32] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[33] P. Wadler. Efficient compilation of pattern-matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.

A Libraries of the KiCS2 Distribution

The KiCS2 distribution comes with a set of base libraries and an extensive collection of libraries for application programming that can be downloaded with the Curry Package Manager (see Section 6). The available packages can be found on-line.[14] Below we sketch some packages for meta-programming followed by the complete description of the base libraries with all exported types and functions. For a more detailed online documentation of the base libraries of KiCS2, see `https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/base.html`.

A.1 AbstractCurry and FlatCurry: Meta-Programming in Curry

To support meta-programming, i.e., the manipulation of Curry programs in Curry, there are Curry packages `flatcurry` and `abstractcurry` which define datatypes for the representation of Curry programs. `AbstractCurry.Types` (package `abstractcurry`) is a more direct representation of a Curry program, whereas `FlatCurry.Types` (package `flatcurry`) is a simplified representation where local function definitions are replaced by global definitions (i.e., lambda lifting has been performed) and pattern matching is translated into explicit case/or expressions. Thus, `FlatCurry.Types` can be used for more back-end oriented program manipulations (or, for writing new back ends for Curry), whereas `AbstractCurry.Types` is intended for manipulations of programs that are more oriented towards the source program.

There are predefined I/O actions to read AbstractCurry and FlatCurry programs: `AbstractCurry.Files.readCurry`) and `FlatCurry.Files.readFlatCurry`). These actions parse the corresponding source program and return a data term representing this program (according to the definitions in the modules `AbstractCurry.Types` and `FlatCurry.Types`).

Since all datatypes are explained in detail in these modules, we refer to the online documentation[15] of these packages.

As an example, consider a program file "`test.curry`" containing the following two lines:

```
rev :: [a] → [a]
rev []     = []
rev (x:xs) = (rev xs) ++ [x]
```

Then the I/O action `(FlatCurry.Files.readFlatCurry "test")` returns the following term:

```
Prog "test"
  ["Prelude"]
  []
  [Func ("test","rev") 1 Public
    (ForallType [(0,KStar)] (FuncType (TCons ("Prelude","[]") [TVar 0])
                                      (TCons ("Prelude","[]") [TVar 0])))
    (Rule [1]
       (Case Flex (Var 1)
          [Branch (Pattern ("Prelude","[]") [])
             (Comb ConsCall ("Prelude","[]") []),
           Branch (Pattern ("Prelude",":") [2,3])
```

[14] `https://www-ps.informatik.uni-kiel.de/~cpm/`

[15] `https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/flatcurry.html`
`https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/abstract-curry.html`

```
           (Comb FuncCall ("Prelude","++")
                 [Comb FuncCall ("test","rev") [Var 3],
                  Comb ConsCall ("Prelude",":")
                       [Var 2,Comb ConsCall ("Prelude","[]") []]
                 ])]))]
[]
```

A.2 System Libraries

A.2.1 Library Control.Applicative

Exported functions:

`liftA :: Applicative a ⇒ (c → b) → Apply a c → Apply a b`

Lift a function to actions. This function may be used as a value for `fmap` in a `Functor` instance.

`liftA3 :: Applicative c ⇒ (b → a → e → d) → Apply c b → Apply c a → Apply c e → Apply c d`

Lift a ternary function to actions.

`when :: Applicative a ⇒ Bool → Apply a () → Apply a ()`

`sequenceA :: Applicative b ⇒ [Apply b a] → Apply b [a]`

Evaluate each action in the list from left to right, and collect the results. For a version that ignores the results see codesequenceA_</code>.

`sequenceA_ :: Applicative a ⇒ [Apply a b] → Apply a ()`

Evaluate each action in the structure from left to right, and ignore the results. For a version that doesn't ignore the results see codesequenceA/code.

A.2.2 Library Control.Monad

Exported functions:

`filterM :: Applicative b ⇒ (a → Apply b Bool) → [a] → Apply b [a]`

This generalizes the list-based codefilter/code function.

`(>=>) :: Monad c ⇒ (d → Apply c b) → (b → Apply c a) → d → Apply c a`

Left-to-right composition of Kleisli arrows.

`(<=<) :: Monad c ⇒ (b → Apply c a) → (d → Apply c b) → d → Apply c a`

Right-to-left composition of Kleisli arrows. @(code>=>/code)@, with the arguments flipped.

`forever :: Applicative c ⇒ Apply c b → Apply c a`

Repeat an action indefinitely.

`mapAndUnzipM :: Applicative b ⇒ (a → Apply b (d,c)) → [a] → Apply b ([d],[c])`

The codemapAndUnzipM/code function maps its first argument over a list, returning the result as a pair of lists. This function is mainly used with complicated data structures or a state-transforming monad.

`zipWithM :: Applicative d ⇒ (b → a → Apply d c) → [b] → [a] → Apply d [c]`

The codezipWithM/code function generalizes codezipWith/code to arbitrary applicative functors.

`zipWithM_ :: Applicative d ⇒ (b → a → Apply d c) → [b] → [a] → Apply d ()`

codezipWithM_</code> is the extension of <code>zipWithM</code> which ignores the final result.

`foldM :: Monad a ⇒ (c → b → Apply a c) → c → [b] → Apply a c`

The codefoldM/code function is analogous to codefoldl/code, except that its result is encapsulated in a monad.

`foldM_ :: Monad a ⇒ (c → b → Apply a c) → c → [b] → Apply a ()`

Like codefoldM/code, but discards the result.

`replicateM :: Applicative b ⇒ Int → Apply b a → Apply b [a]`

`replicateM_ :: Applicative b ⇒ Int → Apply b a → Apply b ()`

Like codereplicateM/code, but discards the result.

`unless :: Applicative a ⇒ Bool → Apply a () → Apply a ()`

The reverse of codewhen/code.

`liftM3 :: Monad b ⇒ (e → d → c → a) → Apply b e → Apply b d → Apply b c → Apply b a`

`join :: Monad b ⇒ Apply b (Apply b a) → Apply b a`

Removes one level of monadic structure, i.e. codeflattens/code the monad.

`void :: Functor b ⇒ Apply b a → Apply b ()`

Ignores the result of the evaluation.

A.2.3 Library Curry.Compiler.Distribution

This module contains definition of constants to obtain information concerning the current distribution of the Curry implementation, e.g., compiler version, run-time version, installation directory.

Exported functions:

`curryCompiler :: String`

The name of the Curry compiler (e.g., "pakcs" or "kics2").

`curryCompilerMajorVersion :: Int`

The major version number of the Curry compiler.

`curryCompilerMinorVersion :: Int`

The minor version number of the Curry compiler.

`curryCompilerRevisionVersion :: Int`

The revision version number of the Curry compiler.

`curryRuntime :: String`

The name of the run-time environment (e.g., "sicstus", "swi", or "ghc")

`curryRuntimeMajorVersion :: Int`

The major version number of the Curry run-time environment.

`curryRuntimeMinorVersion :: Int`

The minor version number of the Curry run-time environment.

`baseVersion :: String`

The version number of the base libraries (e.g., "1.0.5").

`installDir :: String`

Path of the main installation directory of the Curry compiler.

A.2.4 Library Data.Char

Library with some useful functions on characters.

Exported functions:

`isAscii :: Char → Bool`

Returns true if the argument is an ASCII character.

`isLatin1 :: Char → Bool`

Returns true if the argument is an Latin-1 character.

`isAsciiLower :: Char → Bool`

Returns true if the argument is an ASCII lowercase letter.

`isAsciiUpper :: Char → Bool`

Returns true if the argument is an ASCII uppercase letter.

`isControl :: Char → Bool`

Returns true if the argument is a control character.

`toUpper :: Char → Char`

Converts lowercase into uppercase letters.

`toLower :: Char → Char`

Converts uppercase into lowercase letters.

`digitToInt :: Char → Int`

Converts a (hexadecimal) digit character into an integer.

`intToDigit :: Int → Char`

Converts an integer into a (hexadecimal) digit character.

A.2.5 Library Data.Either

Library with some useful operations for the `Either` data type.

Exported functions:

`lefts :: [Either b a] → [b]`

Extracts from a list of `Either` all the `Left` elements in order.

`rights :: [Either a b] → [b]`

Extracts from a list of `Either` all the `Right` elements in order.

`isLeft :: Either a b → Bool`

Return `True` if the given value is a `Left`-value, `False` otherwise.

`isRight :: Either a b → Bool`

Return `True` if the given value is a `Right`-value, `False` otherwise.

`fromLeft :: Either b a → b`

Extract the value from a `Left` constructor.

`fromRight :: Either a b → b`

Extract the value from a `Right` constructor.

`partitionEithers :: [Either b a] → ([b],[a])`

Partitions a list of `Either` into two lists. All the `Left` elements are extracted, in order, to the first component of the output. Similarly the `Right` elements are extracted to the second component of the output.

A.2.6 Library Data.Function

This module provides some utility functions for function application.

Exported functions:

`fix :: (a → a) → a`

`fix f` is the least fixed point of the function `f`, i.e. the least defined `x` such that `f x = x`.

`on :: (a → a → b) → (c → a) → c → c → b`

`on f g x y` applies the binary operation `f` to the results of applying operation `g` to two arguments `x` and `y`. Thus, it transforms two inputs and combines the outputs.

```
(*) `on` f = \x y -> f x * f y
```

A typical usage of this operation is:

```
sortBy (compare `on` fst)
```

A.2.7 Library Data.Functor.Compose

This simple module defines the compose functor known from Haskell's base libraries. The compose functor is the composition of two functors which always is a functor too.

Exported types:

`newtype Compose`

Exported constructors:

- `Compose :: (Apply a (Apply b c))` → `Compose a b c`

Exported functions:

`getCompose :: Compose c b a` → `Apply c (Apply b a)`

A.2.8 Library Data.Functor.Const

This simple module defines the const functor known from Haskell's base libraries. It defines a wrapper around a constant value that "ignores" functions mapped over it.

Exported types:

`newtype Const`

Exported constructors:

- `Const :: a → Const a b`

Exported functions:

`getConst :: Const b a → b`

A.2.9 Library Data.Functor.Identity

This simple module defines the identify functor and monad and has been adapted from the same Haskell module (by Andy Gill). It defines a a trivial type constructor `Identity` which can be used with functions parameterized by functor or monad classes or as a simple base to specialize monad transformers.

Exported types:

`newtype Identity`

The `Identity` type constructor with `Functor`, `Applicative`, and `Monad` instances.

Exported constructors:

- `Identity :: a → Identity a`

Exported functions:

`runIdentity :: Identity a → a`

A.2.10 Library Data.IORef

Library with some useful extensions to the IO monad.

Exported types:

`data IORef`

> Mutable variables containing values of some type. The values are not evaluated when they are assigned to an IORef.

Exported constructors:

Exported functions:

`newIORef :: a → IO (IORef a)`

> Creates a new IORef with an initial value.

`readIORef :: IORef a → IO a`

> Reads the current value of an IORef.

`writeIORef :: IORef a → a → IO ()`

> Updates the value of an IORef.

`modifyIORef :: IORef a → (a → a) → IO ()`

> Modify the value of an IORef.

A.2.11 Library Data.List

Library with some useful operations on lists.

Exported functions:

`elemIndex :: Eq a ⇒ a → [a] → Maybe Int`

Returns the index `i` of the first occurrence of an element in a list as `(Just i)`, otherwise `Nothing` is returned.

`elemIndices :: Eq a ⇒ a → [a] → [Int]`

Returns the list of indices of occurrences of an element in a list.

`find :: (a → Bool) → [a] → Maybe a`

Returns the first element `e` of a list satisfying a predicate as `(Just e)`, otherwise `Nothing` is returned.

`findIndex :: (a → Bool) → [a] → Maybe Int`

Returns the index `i` of the first occurrences of a list element satisfying a predicate as `(Just i)`, otherwise `Nothing` is returned.

`findIndices :: (a → Bool) → [a] → [Int]`

Returns the list of indices of list elements satisfying a predicate.

`nub :: Eq a ⇒ [a] → [a]`

Removes all duplicates in the argument list.

`nubBy :: (a → a → Bool) → [a] → [a]`

Removes all duplicates in the argument list according to an equivalence relation.

`delete :: Eq a ⇒ a → [a] → [a]`

Deletes the first occurrence of an element in a list.

`deleteBy :: (a → a → Bool) → a → [a] → [a]`

Deletes the first occurrence of an element in a list according to an equivalence relation.

`(\\) :: Eq a ⇒ [a] → [a] → [a]`

Computes the difference of two lists.

`union :: Eq a ⇒ [a] → [a] → [a]`

Computes the union of two lists.

`unionBy :: (a → a → Bool) → [a] → [a] → [a]`

Computes the union of two lists according to the given equivalence relation

`intersect :: Eq a ⇒ [a] → [a] → [a]`

Computes the intersection of two lists.

`intersectBy :: (a → a → Bool) → [a] → [a] → [a]`

Computes the intersection of two lists according to the given equivalence relation

`intersperse :: a → [a] → [a]`

Puts a separator element between all elements in a list.

Example: `(intersperse 9 [1,2,3,4]) = [1,9,2,9,3,9,4]`

`intercalate :: [a] → [[a]] → [a]`

`intercalate xs xss` is equivalent to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

`transpose :: [[a]] → [[a]]`

Transposes the rows and columns of the argument.

Example: `(transpose [[1,2,3],[4,5,6]]) = [[1,4],[2,5],[3,6]]`

`diagonal :: [[a]] → [a]`

Diagonalization of a list of lists. Fairly merges (possibly infinite) list of (possibly infinite) lists.

`permutations :: [a] → [[a]]`

Returns the list of all permutations of the argument.

`partition :: (a → Bool) → [a] → ([a],[a])`

Partitions a list into a pair of lists where the first list contains those elements that satisfy the predicate argument and the second list contains the remaining arguments.

Example: `(partition (<4) [8,1,5,2,4,3]) = ([1,2,3],[8,5,4])`

`group :: Eq a ⇒ [a] → [[a]]`

Splits the list argument into a list of lists of equal adjacent elements.

Example: `(group [1,2,2,3,3,3,4]) = [[1],[2,2],[3,3,3],[4]]`

`groupBy :: (a → a → Bool) → [a] → [[a]]`

Splits the list argument into a list of lists of related adjacent elements.

`splitOn :: Eq a ⇒ [a] → [a] → [[a]]`

Breaks the second list argument into pieces separated by the first list argument, consuming the delimiter. An empty delimiter is invalid, and will cause an error to be raised.

`split :: (a → Bool) → [a] → [[a]]`

Splits a list into components delimited by separators, where the predicate returns True for a separator element. The resulting components do not contain the separators. Two adjacent separators result in an empty component in the output.

```
split (==<code>a</code>) "aabbaca" == ["","","bb","c",""]
split (==<code>a</code>) ""        == [""]
```

`inits :: [a] → [[a]]`

Returns all initial segments of a list, starting with the shortest. Example: `inits [1,2,3] == [[],[1],[1,2],[1,2,3]]`

`tails :: [a] → [[a]]`

Returns all final segments of a list, starting with the longest. Example: `tails [1,2,3] == [[1,2,3],[2,3],[3],[]]`

`replace :: a → Int → [a] → [a]`

Replaces an element in a list.

`isPrefixOf :: Eq a ⇒ [a] → [a] → Bool`

Checks whether a list is a prefix of another.

`isSuffixOf :: Eq a ⇒ [a] → [a] → Bool`

Checks whether a list is a suffix of another.

`isInfixOf :: Eq a ⇒ [a] → [a] → Bool`

Checks whether a list is contained in another.

`sort :: Ord a ⇒ [a] → [a]`

The default sorting operation, mergeSort, with standard ordering `<=`.

`sortBy :: (a → a → Bool) → [a] → [a]`

Sorts a list w.r.t. an ordering relation by the insertion method.

`insertBy :: (a → a → Bool) → a → [a] → [a]`

Inserts an object into a list according to an ordering relation.

`last :: [a] → a`

Returns the last element of a non-empty list.

`init :: [a] → [a]`

Returns the input list with the last element removed.

`sum :: Num a ⇒ [a] → a`

Returns the sum of a list of integers.

`product :: Num a ⇒ [a] → a`

Returns the product of a list of integers.

`maximum :: Ord a ⇒ [a] → a`

Returns the maximum of a non-empty list.

`maximumBy :: (a → a → Ordering) → [a] → a`

Returns the maximum of a non-empty list according to the given comparison function

`minimum :: Ord a ⇒ [a] → a`

Returns the minimum of a non-empty list.

`minimumBy :: (a → a → Ordering) → [a] → a`

Returns the minimum of a non-empty list according to the given comparison function

`scanl :: (b → a → b) → b → [a] → [b]`

`scanl` is similar to `foldl`, but returns a list of successive reduced values from the left: scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]

`scanl1 :: (a → a → a) → [a] → [a]`

`scanl1` is a variant of `scanl` that has no starting value argument: scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]

`scanr :: (a → b → b) → b → [a] → [b]`

`scanr` is the right-to-left dual of `scanl`.

`scanr1 :: (a → a → a) → [a] → [a]`

`scanr1` is a variant of `scanr` that has no starting value argument.

`mapAccumL :: (c → b → (c,a)) → c → [b] → (c,[a])`

The `mapAccumL` function behaves like a combination of `map` and `foldl`; it applies a function to each element of a list, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new list.

`mapAccumR :: (c → b → (c,a)) → c → [b] → (c,[a])` The `mapAccumR` function behaves like a combination of `map` and `foldr`; it applies a function to each element of a list, passing an accumulating parameter from right to left, and returning a final value of this accumulator together with the new list.

`cycle :: [a] → [a]`

Builds an infinite list from a finite one.

`unfoldr :: (b → Maybe (a,b)) → b → [a]`

Builds a list from a seed value.

A.2.12 Library Data.Maybe

Library with some useful functions on the `Maybe` datatype.

Exported functions:

`isJust :: Maybe a → Bool`

Return `True` iff the argument is of the form `Just _`.

`isNothing :: Maybe a → Bool`

Return `True` iff the argument is of the form `Nothing`.

`fromJust :: Maybe a → a`

Extract the argument from the `Just` constructor and throw an error if the argument is `Nothing`.

`fromMaybe :: a → Maybe a → a`

Extract the argument from the `Just` constructor or return the provided default value if the argument is `Nothing`.

`listToMaybe :: [a] → Maybe a`

Return `Nothing` on an empty list or `Just x` where `x` is the first list element.

`maybeToList :: Maybe a → [a]`

Return an empty list for `Nothing` or a singleton list for `Just x`.

`catMaybes :: [Maybe a] → [a]`

Return the list of all `Just` values.

`mapMaybe :: (a → Maybe b) → [a] → [b]`

Apply a function which may throw out elements using the `Nothing` constructor to a list of elements.

A.2.13 Library Debug.Trace

This library contains some useful operation for debugging programs.

Exported functions:

`trace :: String → a → a`

Prints the first argument as a side effect and behaves as identity on the second argument.

`traceId :: String → String`

Prints the first argument as a side effect and returns it afterwards.

`traceShow :: Show b ⇒ b → a → a`

Prints the first argument using `show` and returns the second argument afterwards.

`traceShowId :: Show a ⇒ a → a`

Prints the first argument using `show` and returns it afterwards.

`traceIO :: String → IO ()`

Output a trace message from the `IO` monad.

`assert :: Bool → String → a → a`

Assert a condition w.r.t. an error message. If the condition is not met it fails with the given error message, otherwise the third argument is returned.

`assertIO :: Bool → String → IO ()`

Assert a condition w.r.t. an error message from the `IO` monad. If the condition is not met it fails with the given error message.

A.2.14 Library Numeric

Library with some functions for reading and converting numeric tokens.

Exported functions:

`readInt :: String → [(Int,String)]`

Read a (possibly negative) integer as a first token in a string. The string might contain leadings blanks and the integer is read up to the first non-digit. On success returns `[(v,s)]`, where `v` is the value of the integer and `s` is the remaing string without the integer token.

`readNat :: String → [(Int,String)]`

Read a natural number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-digit. On success returns `[(v,s)]`, where `v` is the value of the number and s is the remaing string without the number token.

`readHex :: String → [(Int,String)]`

Read a hexadecimal number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-hexadecimal digit. On success returns `[(v,s)]`, where `v` is the value of the number and s is the remaing string without the number token.

`readOct :: String → [(Int,String)]`

Read an octal number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-octal digit. On success returns `[(v,s)]`, where `v` is the value of the number and s is the remaing string without the number token.

`readBin :: String → [(Int,String)]`

Read a binary number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-binary digit. On success returns `[(v,s)]`, where `v` is the value of the number and s is the remaing string without the number token.

A.2.15 Library Prelude

The standard prelude of Curry with type classes. All exported functions, data types, type classes and methods defined in this module are always available in any Curry program.

Exported types:

`data ()`

Exported constructors:

- `() :: ()`

`data (,)`

Exported constructors:

- `(,) :: a → b → (,) a b`

`data (,,)`

Exported constructors:

- `(,,) :: a → b → c → (,,) a b c`

`data (,,,)`

Exported constructors:

- `(,,,) :: a → b → c → d → (,,,) a b c d`

`data (,,,,)`

Exported constructors:

- `(,,,,) :: a → b → c → d → e → (,,,,) a b c d e`

`data (,,,,,)`

Exported constructors:

- (,,,,,) :: a → b → c → d → e → f → (,,,,,) a b c d e f

`data (,,,,,,)`

Exported constructors:

- (,,,,,,) :: a → b → c → d → e → f → g → (,,,,,,) a b c d e f g

`data (,,,,,,,)`

Exported constructors:

- (,,,,,,,) :: a → b → c → d → e → f → g → h → (,,,,,,,) a b c d e f g h

`data (,,,,,,,,)`

Exported constructors:

- (,,,,,,,,) :: a → b → c → d → e → f → g → h → i → (,,,,,,,,) a b c d e f g h i

`data (,,,,,,,,,)`

Exported constructors:

- (,,,,,,,,,) :: a → b → c → d → e → f → g → h → i → j → (,,,,,,,,,) a b c d e f g h i j

`data (,,,,,,,,,,)`

Exported constructors:

- (,,,,,,,,,,) :: a → b → c → d → e → f → g → h → i → j → k → (,,,,,,,,,,) a b c d e f g h i j k

`data (,,,,,,,,,,,)`

Exported constructors:

- (,,,,,,,,,,,) :: a → b → c → d → e → f → g → h → i → j → k → l → (,,,,,,,,,,,) a b c d e f g h i j k l

data (,,,,,,,,,,,,)

Exported constructors:

- (,,,,,,,,,,,,) :: a → b → c → d → e → f → g → h → i → j → k → l → m → (,,,,,,,,,,,,) a b c d e f g h i j k l m

data (,,,,,,,,,,,,,)

Exported constructors:

- (,,,,,,,,,,,,,) :: a → b → c → d → e → f → g → h → i → j → k → l → m → n → (,,,,,,,,,,,,,) a b c d e f g h i j k l m n

data (,,,,,,,,,,,,,,)

Exported constructors:

- (,,,,,,,,,,,,,,) :: a → b → c → d → e → f → g → h → i → j → k → l → m → n → o → (,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o

data (->)

Exported constructors:

data Bool

Exported constructors:

- False :: Bool
- True :: Bool

data Char

Exported constructors:

`data Either`

Exported constructors:

- `Left :: a → Either a b`
- `Right :: b → Either a b`

`data Float`

Exported constructors:

`data IO`

Exported constructors:

`data IOError`

The (abstract) type of error values. Currently, it distinguishes between general I/O errors, user-generated errors (see `code`userError`/code`), failures and non-determinism errors during I/O computations. These errors can be caught by `code`catch`/code`. Each error contains a string shortly explaining the error. This type might be extended in the future to distinguish further error situations.

Exported constructors:

- `IOError :: String → IOError`
- `UserError :: String → IOError`
- `FailError :: String → IOError`
- `NondetError :: String → IOError`

`data Int`

Exported constructors:

`data Maybe`

Exported constructors:

- `Nothing :: Maybe a`
- `Just :: a → Maybe a`

`data Ordering`

Exported constructors:

- `LT :: Ordering`
- `EQ :: Ordering`
- `GT :: Ordering`

`data []`

Exported constructors:

- `[] :: [] a`
- `: :: a → [a] → [] a`

Exported functions:

`(/==) :: Data a ⇒ a → a → Bool`

The negation of strict equality.

`shows :: Show a ⇒ a → String → String`

Converts a showable value to a show function that prepends this value.

`showChar :: Char → String → String`

Converts a character to a show function that prepends the character.

`showString :: String → String → String`

Converts a string to a show function that prepends the string.

`showParen :: Bool → (String → String) → String → String`

If the first argument is `True`, Converts a show function to a show function adding enclosing brackets, otherwise the show function is returned unchanged.

`showTuple :: [String → String] → String → String`

Converts a list of show functions to a show function combining the given show functions to a tuple representation.

`reads :: Read a ⇒ String → [(a,String)]`

`readParen :: Bool → (String → [(a,String)]) → String → [(a,String)]`

`read :: Read a ⇒ String → a`

`lex :: String → [(String,String)]`

`even :: Integral a ⇒ a → Bool`

Returns whether an integer is even.

`odd :: Integral a ⇒ a → Bool`

Returns whether an integer is odd.

`fromIntegral :: Integral a ⇒ Num b ⇒ a → b`

General coercion from integral types.

`realToFrac :: Real a ⇒ Fractional b ⇒ a → b`

General coercion to fractional types.

`(^) :: Num b ⇒ Integral a ⇒ b → a → b`

`(<$>) :: Functor c ⇒ (b → a) → Apply c b → Apply c a`

`liftM2 :: Monad b ⇒ (d → c → a) → Apply b d → Apply b c → Apply b a`

`sequence :: Monad a ⇒ [Apply a b] → Apply a [b]`

Executes a sequence of monadic actions and collects all results in a list.

`sequence_ :: Monad a ⇒ [Apply a b] → Apply a ()`

Executes a sequence of monadic actions and ignores the results.

`mapM :: Monad c ⇒ (a → Apply c b) → [a] → Apply c [b]`

Maps a monadic action function on a list of elements. The results of all monadic actions are collected in a list.

`mapM_ :: Monad c ⇒ (a → Apply c b) → [a] → Apply c ()`

Maps an monadic action function on a list of elements. The results of all monadic actions are ignored.

`isUpper :: Char → Bool`

Returns true if the argument is an uppercase letter.

`isLower :: Char → Bool`

Returns true if the argument is an lowercase letter.

`isAlpha :: Char → Bool`

Returns true if the argument is a letter.

`isDigit :: Char → Bool`

Returns true if the argument is a decimal digit.

`isAlphaNum :: Char → Bool`

Returns true if the argument is a letter or digit.

`isBinDigit :: Char → Bool`

Returns true if the argument is a binary digit.

`isOctDigit :: Char → Bool`

Returns true if the argument is an octal digit.

`isHexDigit :: Char → Bool`

Returns true if the argument is a hexadecimal digit.

`isSpace :: Char → Bool`

Returns true if the argument is a white space.

`ord :: Char → Int`

Converts a character into its ASCII value.

`chr :: Int → Char`

Converts a Unicode value into a character. The conversion is total, i.e., for out-of-bound values, the smallest or largest character is generated.

`lines :: String → [String]`

Breaks a string into a list of lines where a line is terminated at a newline character. The resulting lines do not contain newline characters.

`unlines :: [String] → String`

Concatenates a list of strings with terminating newlines.

`words :: String → [String]`

Breaks a string into a list of words where the words are delimited by white spaces.

`unwords :: [String] → String`

Concatenates a list of strings with a blank between two strings.

`($) :: (b → a) → b → a`

Right-associative application.

`($!) :: (a → b) → a → b`

Right-associative application with strict evaluation of its argument to head normal form.

`($!!) :: (a → b) → a → b`

Right-associative application with strict evaluation of its argument to normal form.

`($#) :: (b → a) → b → a`

Right-associative application with strict evaluation of its argument to a non-variable term.

`($##) :: (a → b) → a → b`

Right-associative application with strict evaluation of its argument to ground normal form.

`seq :: b → a → a`

Evaluates the first argument to head normal form (which could also be a free variable) and returns the second argument.

`ensureNotFree :: a → a`

Evaluates the argument to head normal form and returns it. Suspends until the result is bound to a non-variable term.

`ensureSpine :: [a] → [a]`

Evaluates the argument to spine form and returns it. Suspends until the result is bound to a non-variable spine.

`normalForm :: a → a`

Evaluates the argument to normal form and returns it.

`groundNormalForm :: a → a`

Evaluates the argument to ground normal form and returns it. Suspends as long as the normal form of the argument is not ground.

`(.) :: (a → b) → (c → a) → c → b`

Function composition.

`id :: a → a`

Identity function.

`const :: b → a → b`

Constant function.

`asTypeOf :: a → a → a`

`asTypeOf` is a type-restricted version of `const`. It is usually used as an infix operator, and its typing forces its first argument (which is usually overloaded) to have the same type as the second.

`curry :: ((c,b) → a) → c → b → a`

Converts an uncurried function to a curried function.

`uncurry :: (c → b → a) → (c,b) → a`

Converts an curried function to a function on pairs.

`flip :: (b → c → a) → c → b → a`

`flip f` is identical to `f`, but with the order of arguments reversed.

`until :: (a → Bool) → (a → a) → a → a`

Repeats application of a function until a predicate holds.

`(&&) :: Bool → Bool → Bool`

Sequential conjunction on Booleans.

`(||) :: Bool → Bool → Bool`

Sequential disjunction on Booleans.

`not :: Bool → Bool`

Negation on Booleans.

`otherwise :: Bool`

Useful name for the last condition in a sequence of conditional equations.

`ifThenElse :: Bool → a → a → a`

The standard conditional. It suspends if the condition is a free variable.

`fst :: (b,a) → b`

Selects the first component of a pair.

`snd :: (b,a) → a`

Selects the second component of a pair.

`head :: [a] → a`

Computes the first element of a list.

`tail :: [a] → [a]`

Computes the remaining elements of a list.

`null :: [a] → Bool`

Is a list empty?

`(++) :: [a] → [a] → [a]`

Concatenates two lists. Since it is flexible, it could be also used to split a list into two sublists etc.

`length :: [a] → Int`

Computes the length of a list.

`(!!) :: [a] → Int → a`

List index (subscript) operator, head has index 0.

`map :: (b → a) → [b] → [a]`

Maps a function on all elements of a list.

`foldl :: (b → a → b) → b → [a] → b`

Accumulates all list elements by applying a binary operator from left to right.

`foldl1 :: (a → a → a) → [a] → a`

Accumulates a non-empty list from left to right.

`foldr :: (a → b → b) → b → [a] → b`

Accumulates all list elements by applying a binary operator from right to left.

`foldr1 :: (a → a → a) → [a] → a`

Accumulates a non-empty list from right to left:

`filter :: (a → Bool) → [a] → [a]`

Filters all elements satisfying a given predicate in a list.

`zip :: [b] → [a] → [(b,a)]`

Joins two lists into one list of pairs. If one input list is shorter than the other, the additional elements of the longer list are discarded.

`zip3 :: [c] → [b] → [a] → [(c,b,a)]`

Joins three lists into one list of triples. If one input list is shorter than the other, the additional elements of the longer lists are discarded.

`zipWith :: (c → b → a) → [c] → [b] → [a]`

Joins two lists into one list by applying a combination function to corresponding pairs of elements. Thus `zip = zipWith (,)`

`zipWith3 :: (d → c → b → a) → [d] → [c] → [b] → [a]`

Joins three lists into one list by applying a combination function to corresponding triples of elements. Thus `zip3 = zipWith3` (,,)

`unzip :: [(b,a)] → ([b],[a])`

Transforms a list of pairs into a pair of lists.

`unzip3 :: [(c,b,a)] → ([c],[b],[a])`

Transforms a list of triples into a triple of lists.

`concat :: [[a]] → [a]`

Concatenates a list of lists into one list.

`concatMap :: (a → [b]) → [a] → [b]`

Maps a function from elements to lists and merges the result into one list.

`iterate :: (a → a) → a → [a]`

Infinite list of repeated applications of a function f to an element x. Thus, `iterate f x = [x, f x, f (f x), ...]`.

`repeat :: a → [a]`

Infinite list where all elements have the same value. Thus, `repeat x = [x, x, x, ...]`.

`replicate :: Int → a → [a]`

List of length n where all elements have the same value.

`take :: Int → [a] → [a]`

Returns prefix of length n.

`drop :: Int → [a] → [a]`

Returns suffix without first n elements.

`splitAt :: Int → [a] → ([a],[a])`

`splitAt n xs` is equivalent to `(take n xs, drop n xs)`

`takeWhile :: (a → Bool) → [a] → [a]`

Returns longest prefix with elements satisfying a predicate.

`dropWhile :: (a → Bool) → [a] → [a]`

Returns suffix without takeWhile prefix.

`span :: (a → Bool) → [a] → ([a],[a])`

`span p xs` is equivalent to `(takeWhile p xs, dropWhile p xs)`

`break :: (a → Bool) → [a] → ([a],[a])`

`break p xs` is equivalent to `(takeWhile (not . p) xs, dropWhile (not . p) xs)`. Thus, it breaks a list at the first occurrence of an element satisfying p.

`reverse :: [a] → [a]`

Reverses the order of all elements in a list.

`and :: [Bool] → Bool`

Computes the conjunction of a Boolean list.

`or :: [Bool] → Bool`

Computes the disjunction of a Boolean list.

`any :: (a → Bool) → [a] → Bool`

Is there an element in a list satisfying a given predicate?

`all :: (a → Bool) → [a] → Bool`

Is a given predicate satisfied by all elements in a list?

`elem :: Eq a ⇒ a → [a] → Bool`

Element of a list?

`notElem :: Eq a ⇒ a → [a] → Bool`

Not element of a list?

`lookup :: Eq b ⇒ b → [(b,a)] → Maybe a`

Looks up a key in an association list.

`maybe :: b → (a → b) → Maybe a → b`

`either :: (b → a) → (c → a) → Either b c → a`

`getChar :: IO Char`

An action that reads a character from standard output and returns it.

`getLine :: IO String`

An action that reads a line from standard input and returns it.

`putChar :: Char → IO ()`

An action that puts its character argument on standard output.

`putStr :: String → IO ()`

Action to print a string on standard output.

`putStrLn :: String → IO ()`

Action to print a string with a newline on standard output.

`print :: Show a ⇒ a → IO ()`

Converts a term into a string and prints it.

`readFile :: String → IO String`

An action that (lazily) reads a file and returns its contents.

`writeFile :: String → String → IO ()`

An action that writes a file.

`appendFile :: String → String → IO ()`

An action that appends a string to a file. It behaves like `writeFile` if the file does not exist.

`userError :: String → IOError`

A user error value is created by providing a description of the error situation as a string.

`ioError :: IOError → IO a`

Raises an I/O exception with a given error value.

`catch :: IO a → (IOError → IO a) → IO a`

Catches a possible error or failure during the execution of an I/O action. `catch act errfun` executes the I/O action `act`. If an exception or failure occurs during this I/O action, the function `errfun` is applied to the error value.

`success :: Bool`

The always satisfiable constraint.

`solve :: Bool → Bool`

Enforce a Boolean condition to be true. The computation fails if the argument evaluates to `False`.

`doSolve :: Bool → IO ()`

Solves a constraint as an I/O action. Note: The constraint should be always solvable in a deterministic way.

`(=:=) :: a → a → Bool`

The equational constraint. `(e1 =:= e2)` is satisfiable if both sides `e1` and `e2` can be reduced to a unifiable data term (i.e., a term without defined function symbols).

`(=:<=) :: a → a → Bool`

Non-strict equational constraint. This operation is not intended to be used in source programs but it is used to implement functional patterns. Conceptually, `(e1 =:<= e2)` is satisfiable if `e1` can be evaluated to some pattern (data term) that matches `e2`, i.e., `e2` is an instance of this pattern. The `Data` context is required since the resulting pattern might be non-linear so that it abbreviates some further equational constraints, see Section 7.

`(&) :: Bool → Bool → Bool`

Concurrent conjunction. An expression like `(c1 & c2)` is evaluated by evaluating the `c1` and `c2` in a concurrent manner.

`(&>) :: Bool → a → a`

Conditional expression. An expression like `(c &> e)` is evaluated by evaluating the first argument to `True` and then evaluating `e`. The expression has no value if the condition does not evaluate to `True`.

`(?) :: a → a → a`

Non-deterministic choice *par excellence*. The value of `x ? y` is either `x` or `y`.

`anyOf :: [a] → a`

Returns non-deterministically any element of a list.

`unknown :: Data a ⇒ a`

Evaluates to a fresh free variable.

`failed :: a`

A non-reducible polymorphic function. It is useful to express a failure in a search branch of the execution.

`error :: String → a`

Aborts the execution with an error message.

`apply :: (a → b) → a → b`

`cond :: Bool → a → a`

`PEVAL :: a → a`

Identity function used by the partial evaluator to mark expressions to be partially evaluated.

`(===) :: Data a ⇒ a → a → Bool`

`aValue :: Data a ⇒ a`

`(==) :: Eq a ⇒ a → a → Bool`

`(/=) :: Eq a ⇒ a → a → Bool`

`compare :: Ord a ⇒ a → a → Ordering`

`(<) :: Ord a ⇒ a → a → Bool`

`(>) :: Ord a ⇒ a → a → Bool`

`(<=) :: Ord a ⇒ a → a → Bool`

```
(>=) :: Ord a ⇒ a → a → Bool

min :: Ord a ⇒ a → a → a

max :: Ord a ⇒ a → a → a

show :: Show a ⇒ a → String

showsPrec :: Show a ⇒ Int → a → String → String

showList :: Show a ⇒ [a] → String → String

readsPrec :: Read a ⇒ Int → String → [(a,String)]

readList :: Read a ⇒ String → [([a],String)]

minBound :: Bounded a ⇒ a

maxBound :: Bounded a ⇒ a

succ :: Enum a ⇒ a → a

pred :: Enum a ⇒ a → a

toEnum :: Enum a ⇒ Int → a

fromEnum :: Enum a ⇒ a → Int
```

```
enumFrom :: Enum a ⇒ a → [a]

enumFromThen :: Enum a ⇒ a → a → [a]

enumFromTo :: Enum a ⇒ a → a → [a]

enumFromThenTo :: Enum a ⇒ a → a → a → [a]

(+) :: Num a ⇒ a → a → a

(-) :: Num a ⇒ a → a → a

(*) :: Num a ⇒ a → a → a

negate :: Num a ⇒ a → a

abs :: Num a ⇒ a → a

signum :: Num a ⇒ a → a

fromInt :: Num a ⇒ Int → a

(/) :: Fractional a ⇒ a → a → a

recip :: Fractional a ⇒ a → a

fromFloat :: Fractional a ⇒ Float → a
```

```
toFloat :: Real a ⇒ a → Float

div :: Integral a ⇒ a → a → a

mod :: Integral a ⇒ a → a → a

quot :: Integral a ⇒ a → a → a

rem :: Integral a ⇒ a → a → a

divMod :: Integral a ⇒ a → a → (a,a)

quotRem :: Integral a ⇒ a → a → (a,a)

toInt :: Integral a ⇒ a → Int

properFraction :: RealFrac a ⇒ forall b.Integral b ⇒ a → (b,a)

truncate :: RealFrac a ⇒ forall b.Integral b ⇒ a → b

round :: RealFrac a ⇒ forall b.Integral b ⇒ a → b

ceiling :: RealFrac a ⇒ forall b.Integral b ⇒ a → b

floor :: RealFrac a ⇒ forall b.Integral b ⇒ a → b

pi :: Floating a ⇒ a
```

```
exp :: Floating a ⇒ a → a

log :: Floating a ⇒ a → a

sqrt :: Floating a ⇒ a → a

(**) :: Floating a ⇒ a → a → a

logBase :: Floating a ⇒ a → a → a

sin :: Floating a ⇒ a → a

cos :: Floating a ⇒ a → a

tan :: Floating a ⇒ a → a

asin :: Floating a ⇒ a → a

acos :: Floating a ⇒ a → a

atan :: Floating a ⇒ a → a

sinh :: Floating a ⇒ a → a

cosh :: Floating a ⇒ a → a

tanh :: Floating a ⇒ a → a
```

```
asinh :: Floating a ⇒ a → a

acosh :: Floating a ⇒ a → a

atanh :: Floating a ⇒ a → a

mempty :: Monoid a ⇒ a

mappend :: Monoid a ⇒ a → a → a

mconcat :: Monoid a ⇒ [a] → a

fmap :: Functor a ⇒ forall b c.(c → b) → Apply a c → Apply a b

(<$) :: Functor a ⇒ forall b c.c → Apply a b → Apply a c

pure :: Applicative a ⇒ forall b.b → Apply a b

(<*>) :: Applicative a ⇒ forall b c.Apply a (c → b) → Apply a c → Apply a b

(*>) :: Applicative a ⇒ forall b c.Apply a c → Apply a b → Apply a b

(<*) :: Applicative a ⇒ forall b c.Apply a c → Apply a b → Apply a c

liftA2 :: Applicative a ⇒ forall b c d.(d → c → b) → Apply a d → Apply a c →
Apply a b

empty :: Alternative a ⇒ forall b.Apply a b
```

```
(<|>) :: Alternative a ⇒ forall b.Apply a b → Apply a b → Apply a b

some :: Alternative a ⇒ forall b.Apply a b → Apply a [b]

many :: Alternative a ⇒ forall b.Apply a b → Apply a [b]

(>>=) :: Monad a ⇒ forall b c.Apply a c → (c → Apply a b) → Apply a b

(>>) :: Monad a ⇒ forall b c.Apply a c → Apply a b → Apply a b

return :: Monad a ⇒ forall b.b → Apply a b

fail :: MonadFail a ⇒ forall b.String → Apply a b
```

A.2.16 Library System.Console.GetOpt

This module is a modified version of the module `System.Console.GetOpt` by Sven Panne from the ghc-base package. It has been adapted for Curry by Bjoern Peemoeller

Exported types:

`data ArgOrder`

Exported constructors:

- `RequireOrder :: ArgOrder a`
- `Permute :: ArgOrder a`
- `ReturnInOrder :: (String → a) → ArgOrder a`

`data OptDescr`

Exported constructors:

- `Option :: String → [String] → (ArgDescr a) → String → OptDescr a`

`data ArgDescr`

Exported constructors:

- `NoArg :: a → ArgDescr a`
- `ReqArg :: (String → a) → String → ArgDescr a`
- `OptArg :: (Maybe String → a) → String → ArgDescr a`

Exported functions:

```
usageInfo :: String → [OptDescr a] → String
```

```
getOpt :: ArgOrder a → [OptDescr a] → [String] → ([a],[String],[String])
```

```
getOpt' :: ArgOrder a → [OptDescr a] → [String] → ([a],[String],[String],[String])
```

A.2.17 Library System.CPUTime

Exported functions:

`getCPUTime :: IO Int`

Returns the current cpu time of the process in milliseconds.

`getElapsedTime :: IO Int`

Returns the current elapsed time of the process in milliseconds. This operation is not supported in KiCS2 (there it always returns 0), but only included for compatibility reasons.

A.2.18 Library System.Environment

Library to access parts of the system environment.

Exported functions:

`getArgs :: IO [String]`

Returns the list of the program's command line arguments. The program name is not included.

`getEnv :: String → IO String`

Returns the value of an environment variable. The empty string is returned for undefined environment variables.

`setEnv :: String → String → IO ()`

Set an environment variable to a value. The new value will be passed to subsequent shell commands (see `code`system`/code`) and visible to subsequent calls to `code`getEnv`/code` (but it is not visible in the environment of the process that started the program execution).

`unsetEnv :: String → IO ()`

Removes an environment variable that has been set by `code`setEnv`/code`.

`getHostname :: IO String`

Returns the hostname of the machine running this process.

`getProgName :: IO String`

Returns the name of the current program, i.e., the name of the main module currently executed.

`isPosix :: Bool`

Is the underlying operating system a POSIX system (unix, MacOS)?

`isWindows :: Bool`

Is the underlying operating system a Windows system?

A.2.19 Library System.IO

Library for IO operations like reading and writing files that are not already contained in the prelude.

Exported types:

`data Handle`

Exported constructors:

`data IOMode`

The modes for opening a file.

Exported constructors:

- `ReadMode :: IOMode`
- `WriteMode :: IOMode`
- `AppendMode :: IOMode`

`data SeekMode`

The modes for positioning with `hSeek` in a file.

Exported constructors:

- `AbsoluteSeek :: SeekMode`
- `RelativeSeek :: SeekMode`
- `SeekFromEnd :: SeekMode`

Exported functions:

`stdin :: Handle`

Standard input stream.

`stdout :: Handle`

Standard output stream.

`stderr :: Handle`

Standard error stream.

`openFile :: String → IOMode → IO Handle`

Opens a file in specified mode and returns a handle to it.

`hClose :: Handle → IO ()`

Closes a file handle and flushes the buffer in case of output file.

`hFlush :: Handle → IO ()`

Flushes the buffer associated to handle in case of output file.

`hIsEOF :: Handle → IO Bool`

Is handle at end of file?

`isEOF :: IO Bool`

Is standard input at end of file?

`hSeek :: Handle → SeekMode → Int → IO ()`

Set the position of a handle to a seekable stream (e.g., a file). If the second argument is `AbsoluteSeek`, `SeekFromEnd`, or `RelativeSeek`, the position is set relative to the beginning of the file, to the end of the file, or to the current position, respectively.

`hWaitForInput :: Handle → Int → IO Bool`

Waits until input is available on the given handle. If no input is available within t milliseconds, it returns False, otherwise it returns True.

`hWaitForInputs :: [Handle] → Int → IO Int`

Waits until input is available on some of the given handles. If no input is available within the given milliseconds, it returns `-1`, otherwise it returns the index of the corresponding handle with the available data.

`hReady :: Handle → IO Bool`

Checks whether an input is available on a given handle.

`hGetChar :: Handle → IO Char`

Reads a character from an input handle and returns it. Throws an error if the end of file has been reached.

`hGetLine :: Handle → IO String`

Reads a line from an input handle and returns it. Throws an error if the end of file has been reached while reading the *first* character. If the end of file is reached later in the line, it ist treated as a line terminator and the (partial) line is returned.

`hGetContents :: Handle → IO String`

Reads the complete contents from an input handle and closes the input handle before returning the contents.

`getContents :: IO String`

Reads the complete contents from the standard input stream until EOF.

`hPutChar :: Handle → Char → IO ()`

Puts a character to an output handle.

`hPutStr :: Handle → String → IO ()`

Puts a string to an output handle.

`hPutStrLn :: Handle → String → IO ()`

Puts a string with a newline to an output handle.

`hPrint :: Show a ⇒ Handle → a → IO ()`

Converts a term into a string and puts it to an output handle.

`hIsReadable :: Handle → IO Bool`

Is the handle readable?

`hIsWritable :: Handle → IO Bool`

Is the handle writable?

`hIsTerminalDevice :: Handle → IO Bool`

Is the handle connected to a terminal?

A.2.20 Library System.IO.Unsafe

Library containing *unsafe* operations. These operations should be carefully used (e.g., for testing or debugging). These operations should not be used in application programs!

Exported functions:

`unsafePerformIO :: IO a → a`

Performs and hides an I/O action in a computation (use with care!).

`trace :: String → a → a`

Prints the first argument as a side effect and behaves as identity on the second argument.

`spawnConstraint :: Bool → a → a`

Spawns a constraint and returns the second argument. This function can be considered as defined by `spawnConstraint c x | c = x`. However, the evaluation of the constraint and the right-hand side are performed concurrently, i.e., a suspension of the constraint does not imply a blocking of the right-hand side and the right-hand side might be evaluated before the constraint is successfully solved. Thus, a computation might return a result even if some of the spawned constraints are suspended (use the PAKCS option `+suspend` to show such suspended goals).

`isVar :: Data a ⇒ a → Bool`

Tests whether the first argument evaluates to a currently unbound variable (use with care!).

`identicalVar :: Data a ⇒ a → a → Bool`

Tests whether both arguments evaluate to the identical currently unbound variable (use with care!). For instance,

```
identicalVar (id x) (fst (x,1))  where x free
```

evaluates to `True`, whereas

```
identicalVar x y  where x,y free
```

and

```
let x=1 in identicalVar x x
```

evaluate to `False`

`isGround :: Data a ⇒ a → Bool`

Tests whether the argument evaluates to a ground value (use with care!).

`compareAnyTerm :: a → a → Ordering`

Comparison of any data terms, possibly containing variables. Data constructors are compared in the order of their definition in the datatype declarations and recursively in the arguments. Variables are compared in some internal order.

`showAnyTerm :: a → String`

Transforms the normal form of a term into a string representation in standard prefix notation. Thus, showAnyTerm evaluates its argument to normal form. This function is similar to the function `ReadShowTerm.showTerm` but it also transforms logic variables into a string representation that can be read back by `Unsafe.read(s)AnyUnqualifiedTerm`. Thus, the result depends on the evaluation and binding status of logic variables so that it should be used with care!

`readsAnyUnqualifiedTerm :: [String] → String → [(a,String)]`

Transforms a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The string might contain logical variable encodings produced by showAnyTerm. In case of a successful parse, the result is a one element list containing a pair of the data term and the remaining unparsed string.

`readAnyUnqualifiedTerm :: [String] → String → a`

Transforms a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The string might contain logical variable encodings produced by `showAnyTerm`.

`showAnyExpression :: a → String`

Transforms any expression (even not in normal form) into a string representation in standard prefix notation without module qualifiers. The result depends on the evaluation and binding status of logic variables so that it should be used with care!

A.2.21 Library Test.Prop

This module defines the interface of properties that can be checked with the CurryCheck tool, an automatic property-based test tool based on the EasyCheck library. The ideas behind EasyCheck are described in this paper. CurryCheck automatically tests properties defined with this library. CurryCheck supports the definition of unit tests (also for I/O operations) and property tests parameterized over some arguments. CurryCheck is described in more detail in this paper. Basically, this module is a stub clone of the EasyCheck library which contains only the interface of the operations used to specify properties. Hence, this library does not import any other library. This supports the definition of properties in any other module (execept for the prelude).

Exported functions:

`returns :: Eq a ⇒ Show a ⇒ IO a → a → PropIO`

The property `returns a x` is satisfied if the execution of the I/O action `a` returns the value `x`.

`sameReturns :: Eq a ⇒ Show a ⇒ IO a → IO a → PropIO`

The property `sameReturns a1 a2` is satisfied if the execution of the I/O actions `a1` and `a2` return identical values.

`toError :: a → PropIO`

The property `toError a` is satisfied if the evaluation of the argument to normal form yields an exception.

`toIOError :: IO a → PropIO`

The property `toIOError a` is satisfied if the execution of the I/O action `a` causes an exception.

`(-=-) :: Eq a ⇒ Show a ⇒ a → a → Prop`

The property `x -=- y` is satisfied if `x` and `y` have deterministic values that are equal.

`(<~>) :: Eq a ⇒ Show a ⇒ a → a → Prop`

The property x <~> y is satisfied if the sets of the values of `x` and `y` are equal.

`(~>) :: Eq a ⇒ Show a ⇒ a → a → Prop`

The property x ~> y is satisfied if `x` evaluates to every value of `y`. Thus, the set of values of `y` must be a subset of the set of values of `x`.

`(<~) :: Eq a ⇒ Show a ⇒ a → a → Prop`

The property x <~ y is satisfied if `y` evaluates to every value of `x`. Thus, the set of values of `x` must be a subset of the set of values of `y`.

`(<~~>) :: Eq a ⇒ Show a ⇒ a → a → Prop`

The property x <~~> y is satisfied if the multisets of the values of x and y are equal.

(==>) :: Bool → Prop → Prop

A conditional property is tested if the condition evaluates to True.

solutionOf :: Data a ⇒ (a → Bool) → a

solutionOf p returns (non-deterministically) a solution of predicate p. This operation is useful to test solutions of predicates.

is :: Show a ⇒ a → (a → Bool) → Prop

The property is x p is satisfied if x has a deterministic value which satisfies p.

isAlways :: Show a ⇒ a → (a → Bool) → Prop

The property isAlways x p is satisfied if all values of x satisfy p.

isEventually :: Show a ⇒ a → (a → Bool) → Prop

The property isEventually x p is satisfied if some value of x satisfies p.

uniquely :: Bool → Prop

The property uniquely x is satisfied if x has a deterministic value which is true.

always :: Bool → Prop

The property always x is satisfied if all values of x are true.

eventually :: Bool → Prop

The property eventually x is satisfied if some value of x is true.

failing :: Show a ⇒ a → Prop

The property failing x is satisfied if x has no value.

successful :: Show a ⇒ a → Prop

The property successful x is satisfied if x has at least one value.

deterministic :: Show a ⇒ a → Prop

The property deterministic x is satisfied if x has exactly one value.

(#) :: Eq a ⇒ Show a ⇒ a → Int → Prop

The property x # n is satisfied if x has n values.

(#<) :: Eq a ⇒ Show a ⇒ a → Int → Prop

The property x #< n is satisfied if x has less than n values.

(#>) :: Eq a ⇒ Show a ⇒ a → Int → Prop

The property x #> n is satisfied if x has more than n values.

for :: Show a ⇒ a → (a → Prop) → Prop

The property for x p is satisfied if all values y of x satisfy property p y.

forAll :: Show a ⇒ [a] → (a → Prop) → Prop

The property forAll xs p is satisfied if all values x of the list xs satisfy property p x.

(<=>) :: a → a → Prop

The property f <=> g is satisfied if f and g are equivalent operations, i.e., they can be replaced in any context without changing the computed results.

label :: String → Prop → Prop

Assign a label to a property. All labeled tests are counted and shown at the end.

classify :: Bool → String → Prop → Prop

Assign a label to a property if the first argument is True. All labeled tests are counted and shown at the end. Hence, this combinator can be used to classify tests:

```
multIsComm x y = classify (x<0 || y<0) "Negative" $ x*y -=- y*x
```

trivial :: Bool → Prop → Prop

Assign the label "trivial" to a property if the first argument is True. All labeled tests are counted and shown at the end.

collect :: Show a ⇒ a → Prop → Prop

Assign a label showing the given argument to a property. All labeled tests are counted and shown at the end.

collectAs :: Show a ⇒ String → a → Prop → Prop

Assign a label showing a given name and the given argument to a property. All labeled tests are counted and shown at the end.

valuesOf :: a → [a]

Computes the list of all values of the given argument according to a given strategy (here: randomized diagonalization of levels with flattening).

A.2.22 Library Test.Prop.Types

This module defines some types used by the EasyCheck libraries.

Exported types:

`data PropIO`

Abstract type to represent properties involving IO actions.

Exported constructors:

- `PropIO :: (Bool → String → IO (Maybe String)) → PropIO`

`data Prop`

Abstract type to represent standard properties to be checked. Basically, it contains all tests to be executed to check the property.

Exported constructors:

- `Prop :: [Test] → Prop`

`data Test`

Abstract type to represent a single test for a property to be checked. A test consists of the result computed for this test, the arguments used for this test, and the labels possibly assigned to this test by annotating properties.

Exported constructors:

- `Test :: Result → [String] → [String] → Test`

`data Result`

Data type to represent the result of checking a property.

Exported constructors:

- `Undef :: Result`
- `Ok :: Result`
- `Falsified :: [String] → Result`
- `Ambigious :: [Bool] → [String] → Result`

A.2.23 Library Text.Show

This library provides a type and combinators for show functions using functional lists.

Exported functions:

`showString :: String → String → String`

> Prepend a string

`showChar :: Char → String → String`

> Prepend a single character

`showParen :: Bool → (String → String) → String → String`

> Surround the inner show function with parentheses if the first argument evaluates to `True`.

`shows :: Show a ⇒ a → String → String`

> Convert a value to `ShowS` using the standard show function.

B SQL Syntax Supported by CurryPP

This section contains a grammar in EBNF which specifies the SQL syntax recognized by the Curry preprocessor in integrated SQL code (see Sect. 11.4). The grammar satisfies the LL(1) property and is influenced by the SQLite dialect.[16]

```
 --------------type of statements--------------------------------

  statement ::= queryStatement | transactionStatement
  queryStatement ::= ( deleteStatement
                     | insertStatement
                     | selectStatement
                     | updateStatement )
                     ';'

  ------------- transaction ------------------------------------

  transactionStatement ::= (BEGIN
                           |IN TRANSACTION '(' queryStatement
                                            { queryStatement }')'
                           |COMMIT
                           |ROLLBACK ) ';'

 -------------- delete -----------------------------------------

  deleteStatement ::= DELETE FROM tableSpecification
                                 [ WHERE condition ]

  -------------insert ------------------------------------------

  insertStatement ::= INSERT INTO tableSpecification
                                 insertSpecification

  insertSpecification ::= ['(' columnNameList ')' ] valuesClause

  valuesClause ::= VALUES valueList

   ------------update-------------------------------------------

  updateStatement ::= UPDATE tableSpecification
                      SET (columnAssignment {',' columnAssignment}
                                               [ WHERE condition ]
                          | embeddedCurryExpression )

  columnAssignment ::= columnName '=' literal


  -------------select statement ---------------------------------
```

[16] https://sqlite.org/lang.html

```
selectStatement ::= selectHead { setOperator selectHead }
                              [ orderByClause ]
                              [ limitClause ]
selectHead ::= selectClause fromClause
                           [ WHERE condition ]
                           [ groupByClause [ havingClause ]]

setOperator ::= UNION | INTERSECT | EXCEPT

selectClause ::= SELECT [( DISTINCT | ALL )]
                       ( selectElementList | '*' )

selectElementList ::=  selectElement { ',' selectElement }

selectElement ::= [ tableIdentifier'.' ] columnName
                 | aggregation
                 | caseExpression

aggregation ::= function '(' [ DISTINCT ] columnReference ')'

caseExpression ::= CASE WHEN condition THEN operand
                                       ELSE operand END

function ::= COUNT | MIN | MAX | AVG | SUM

fromClause ::= FROM tableReference  { ',' tableReference }

groupByClause ::= GROUP BY columnList

havingClause ::= HAVING conditionWithAggregation

orderByClause ::= ORDER BY columnReference [ sortDirection ]
                          {',' columnReference
                               [ sortDirection ] }

sortDirection ::= ASC | DESC

limitClause = LIMIT integerExpression

-------------common elements-----------------------------------

columnList ::=  columnReference  { ',' columnReference }

columnReference ::= [ tableIdentifier'.' ] columnName

columnNameList ::= columnName  { ',' columnName}

tableReference ::=  tableSpecification [ AS tablePseudonym ]
```

```
                                        [ joinSpecification ]
tableSpecification ::= tableName

condition ::=   operand operatorExpression
                        [logicalOperator condition]
              | EXISTS subquery [logicalOperator condition]
              | NOT condition
              | '(' condition  ')'
              | satConstraint [logicalOperator condition]

operand ::=  columnReference
            | literal


subquery ::= '(' selectStatement ')'

operatorExpression ::=  IS NULL
                      | NOT NULL
                      | binaryOperator operand
                      | IN setSpecification
                      | BETWEEN operand operand
                      | LIKE quotes pattern quotes

setSpecification ::=  literalList

binaryOperator ::= '>'| '<' | '>=' | '<=' | '=' | '!='

logicalOperator ::= AND | OR

conditionWithAggregation ::=
        aggregation [logicalOperator disaggregation]
      | '(' conditionWithAggregation  ')'
      | operand operatorExpression
               [logicalOperator conditionWithAggregation]
      | NOT conditionWithAggregation
      | EXISTS subquery
               [logicalOperator conditionWithAggregation]
      | satConstraint
               [logicalOperator conditionWithAggregation]

aggregation ::= function '('(ALL | DISTINCT) columnReference')'
                          binaryOperator
                          operand

satConstraint ::= SATISFIES tablePseudonym
                            relation
                            tablePseudonym

joinSpecification ::=  joinType tableSpecification
```

```
                              [ AS tablePseudonym ]
                              [ joinCondition ]
                              [ joinSpecification ]

joinType ::= CROSS JOIN | INNER JOIN

joinCondition ::= ON condition

-------------identifier and datatypes-------------------------

valueList ::= ( embeddedCurryExpression | literalList )
               {',' ( embeddedCurryExpression | literalList )}

literalList ::= '(' literal  { ',' literal }  ')'

literal ::=   numericalLiteral
            | quotes alphaNumericalLiteral quotes
            | dateLiteral
            | booleanLiteral
            | embeddedCurryExpression
            | NULL

numericalLiteral ::=  integerExpression
                     |floatExpression

integerExpression ::= [ - ] digit { digit }

floatExpression := [ - ] digit { digit } '.' digit { digit }

alphaNumericalLiteral ::=  character { character }
character ::= digit | letter

dateLiteral ::=  year ':' month ':' day ':'
                 hours ':' minutes ':' seconds

month ::= digit digit
day ::= digit digit
hours ::= digit digit
minutes ::= digit digit
seconds ::= digit digit
year ::= digit digit digit digit

booleanLiteral ::= TRUE | FALSE

embeddedCurryExpression ::= '{' curryExpression '}'

pattern ::= ( character | specialCharacter )
             {( character | specialCharacter )}
specialCharacter ::= '%' | '_'
```

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

letter ::= (a...z) | (A...Z)

tableIdentifier ::= tablePseudonym | tableName
columnName ::= letter [alphanumericalLiteral]
tableName ::= letter [alphanumericalLiteral]
tablePseudonym ::= letter
relation ::= letter [[alphanumericalLiteral] | '_' ]
quotes ::= ('"'|''')
```

C Auxiliary Files

During the translation and execution of a Curry program with KiCS2, various intermediate representations of the source program are created and stored in different files which are shortly explained in this section. If you use KiCS2, it is not necessary to know about these auxiliary files because they are automatically generated and updated. You should only remember the command for deleting all auxiliary files ("`cleancurry`", see Section 1.2) to clean up your directories.

Usually, the auxiliary files are invisible: if the Curry module M is stored in directory *dir*, the corresponding auxiliary files are stored in directory "*dir*`/.curry/kics2-`*v*" where v is the version of KiCS2. Thus, the auxiliary files produced by different versions of KiCS2 causes no conflicts. This scheme is also used for hierarchical module names: if the module $D1.D2.M$ is stored in directory *dir* (i.e., the module is actually stored in $dir/D1/D2/M.curry$), then the corresponding Prolog program is stored in directory "*dir*`/.curry/kics2-`*v*`/D1/D2`".

The various components of KiCS2 create the following auxiliary files.

`prog.fcy`: This file contains the Curry program in the so-called "FlatCurry" representation where all functions are global (i.e., lambda lifting has been performed) and pattern matching is translated into explicit case/or expressions (compare Appendix A.1). This representation might be useful for other back ends and compilers for Curry and is the basis doing metaprogramming in Curry. This file is implicitly generated when a program is compiled with KiCS2. It can be also explicitly generated by the front end of KiCS2:

```
kics2 frontend --flat -ikics2home/lib prog
```

The FlatCurry representation of a Curry program is usually generated by the front-end after parsing, type checking and eliminating local declarations.

`prog.fint`: This file contains the interface of the program in the so-called "FlatCurry" representation, i.e., it is similar to `prog.fcy` but contains only exported entities and the bodies of all functions omitted (i.e., "external"). This representation is useful for providing a fast access to module interfaces. This file is implicitly generated when a program is compiled with KiCS2 and stored in the same directory as `prog.fcy`.

`Curry_prog.hs`: This file contains a Haskell program as the result of translating the Curry program with the KiCS2 compiler.

If the Curry module M is stored in the directory *dir*, the corresponding Haskell program is stored in the directory "*dir*`/.curry/kics2`". This is also the case for hierarchical module names: if the module $D1.D2.M$ is stored in the directory *dir* (i.e., the module is actually stored in $dir/D1/D2/M.curry$), then the corresponding Haskell program is stored in "*dir*`/.curry/kics2/D1/D2/Curry_prog.hs`".

`Curry_prog.hi`: This file contains the interface of the Haskell program `Curry_prog.hs` when the latter program is compiled in order to execute it. This file is stored in the same directory as `Curry_prog.hs`.

`Curry_prog.o`: This file contains the object code of the Haskell program `Curry_prog.hs` when the latter program is compiled in order to execute it. This file is stored in the same directory as `Curry_prog.hs`.

`Curry_prog.nda`: This file contains some information about the determinism behavior of operations that is used by the KiCS2 compiler (see [12] for more details about the use of this information). This file is stored in the same directory as `Curry_prog.hs`.

`Curry_prog.info`: This file contains some information about the top-level functions of module `prog` that are used by the interactive environment, like determinism behavior or IO status. This file is stored in the same directory as `Curry_prog.hs`.

`prog`: This file contains the executable after compiling and saving a program with KiCS2 (see command "`:save`" in Section 2.2).

D External Operations

Currently, KiCS2 has no general interface to external operations, i.e., operations whose semantics is not defined by program rules in a Curry program but by some code written in another programming language. Thus, if an external operation should be added to the system, this operation must be declared as `external` in the Curry source code and an implementation for this external operation must be provided for the run-time system. An external operation is defined as follows in the Curry source code:

1. Add a type declaration for the external operation somewhere in a module defining this operation (usually, the prelude or some system module).

2. For external operations it is not allowed to define any rule since their semantics is determined by an external implementation. Instead of the defining rules, you have to write

   ```
   f external
   ```

   below the type declaration for the external operation `f`.

Furthermore, an implementation of the external operation must be provided in the target language of the KiCS2 compiler, i.e., in Haskell, and inserted in the compiled code. In order to simplify this task, KiCS2 follows some code conventions that are described in the following.

Assume you want to implement your own concatenation for strings in a module `String`. The name and type of this string concatenation should be

```
sconc :: String → String → String
```

Since the primitive Haskell implementation of this operation does not now anything about the operational mechanism of Curry (e.g., needed narrowing, non-deterministic rewriting), the arguments need to be completely evaluated before the primitive implementation is called. This can be easily obtained by the prelude operation `($##)` that applies an operation to the *normal form* of the given argument, i.e., this operation evaluates the argument to its normal form before applying the operation to it.[17] Thus, we define `sconc` by

```
sconc :: String → String → String
sconc s1 s2 = (prim_sconc $## s1) $## s2

prim_sconc :: String → String → String
prim_sconc external
```

so that it is ensured that the external operation `prim_sconc` is always called with complete evaluated arguments.

In order to define the Haskell code implementing `prim_sconc`, one has to satisfy the naming conventions of KiCS2. The KiCS2 compiler generates the following code for the external operation `prim_sconc` (note that the generated Haskell code for the module `String` is stored in the file `.curry/kics2/Curry_String.hs`):

[17] There is also a similar prelude operation `($#)` which evaluates the argument only to head-normal form. This is a bit more efficient and can be used for unstructured types like `Bool`.

```
d_C_prim_sconc :: Curry_Prelude.OP_List Curry_Prelude.C_Char
               → Curry_Prelude.OP_List Curry_Prelude.C_Char
               → ConstStore
               → Curry_Prelude.OP_List Curry_Prelude.C_Char
d_C_prim_sconc x1 x2 x3500 = external_d_C_prim_sconc x1 x2 x3500
```

The type constructors `OP_List` and `C_Char` of the prelude `Curry_Prelude`[18] correspond to the Curry type constructors for lists and characters. The Haskell operation `external_d_C_prim_sconc` is the external operation to be implemented in Haskell by the programmer. The additional argument of type `ConstStore` represents the current set of constraints when this operation is called. This argument is intended to provide a more efficient access to binding constraints and can be ignored in standard operations.

If `String.curry` contains the code of the Curry function `sconc` described above, the Haskell code implementing the external operations occurring in the module `String` must be in the file `External_String.hs` which is located in the same directory as the file `String.curry`. The KiCS2 compiler appends the code contained in `External_String.hs` to the generated code stored in the file `.curry/kics2/Curry_String.hs`.[19]

In order to complete our example, we have to write into the file `External_String.hs` a definition of the Haskell function `external_d_C_prim_sconc`. Thus, we start with the following definitions:

```
import qualified Curry_Prelude as CP

external_d_C_prim_sconc :: CP.OP_List CP.C_Char  → CP.OP_List CP.C_Char
                        → ConstStore  → CP.OP_List CP.C_Char
```

First, we import the standard prelude with the name `CP` in order to shorten the writing of type declarations. In order to write the final code of this operation, we have to convert the Curry-related types (like `C_Char`) into the corresponding Haskell types (like `Char`). Note that the Curry-related types contain information about non-deterministic or constrained values (see [12, 13]) that are meaningless in Haskell. To solve this conversion problem, the implementation of KiCS2 provides a family of operations to perform these conversions for the predefined types occurring in the standard prelude. For instance, `fromCurry` converts a Curry type into the corresponding Haskell type, and `toCurry` converts the Haskell type into the corresponding Curry type. Thus, we complete our example with the definition (note that we simply ignore the final argument representing the constraint store)

```
external_d_C_prim_sconc s1 s2 _ =
  toCurry ((fromCurry s1 ++ fromCurry s2) :: String)
```

Here, we use Haskell's concatenation operation "`++`" to concatenate the string arguments. The type annotation "`:: String`" is necessary because "`++`" is a polymorphic function so that the type inference system of Haskell has problems to determine the right instance of the conversion function.

The conversion between Curry types and Haskell types, i.e., the family of conversion operation `fromCurry` and `toCurry`, is defined in the KiCS2 implementation for all standard data types. In particular, it is also defined on function types so that one can easily implement external Curry I/O

[18]Note that all translated Curry modules are imported in the Haskell code fully qualified in order to avoid name conflicts.

[19]If the file `External_String.hs` contains also some import declarations at the beginning, these import declarations are put after the generated import declarations.

actions by using Haskell I/O actions. For instance, if we want to implement an external operation to print some string as an output line, we start by declaring the external operations in the Curry module `String`:

```
printString :: String → IO ()
printString s = prim_printString $## s

prim_printString :: String → IO ()
prim_printString external
```

Next we add the corresponding implementation in the file `External_String.hs` (where `C_IO` and `OP_Unit` are the names of the Haskell representation of the Curry type constructor `IO` and the Curry data type "`()`", respectively):

```
external_d_C_prim_printString :: CP.OP_List CP.C_Char → ConstStore
                              → CP.C_IO CP.OP_Unit
external_d_C_prim_printString s _ = toCurry putStrLn s
```

Here, Haskell's I/O action `putStrLn` of type "`String -> IO ()`" is transformed into a Curry I/O action "`toCurry putStrLn`" which has the type

```
CP.OP_List CP.C_Char → CP.C_IO CP.OP_Unit
```

When we compile the Curry module `String`, KiCS2 combines these definitions in the target program so that we can immediately use the externally defined operation `printString` in Curry programs.

As we have seen, KiCS2 transforms a name like `primOP` of an external operation into the name `external_d_C_primOP` for the Haskell operation to be implemented, i.e., only a specific prefix is added. However, this is only valid if no special characters occur in the Curry names. Otherwise (in order to generate a correct Haskell program), special characters are translated into specific names prefixed by "`OP_`". For instance, if we declare the external operation

```
(<&>) :: Int → Int → Int
(<&>) external
```

the generated Haskell module contains the code

```
d_OP_lt_ampersand_gt :: Curry_Prelude.C_Int → Curry_Prelude.C_Int
                     → ConstStore → Curry_Prelude.C_Int
d_OP_lt_ampersand_gt x1 x2 x3500 = external_d_OP_lt_ampersand_gt x1 x2 x3500
```

so that one has to implement the operation `external_d_OP_lt_ampersand_gt` in Haskell. If in doubt, one should look into the generated Haskell code about the names and types of the operations to be implemented.

Finally, note that this method to connect functions implemented in Haskell to Curry programs provides the opportunity to connect also operations written in other programming languages to Curry via Haskell's foreign function interface.

Index