KiCS2

The Kiel Curry System (Version 2)

User Manual

Version 0.5.1 of 2017-04-10

Michael Hanus[1] [editor]

Additional Contributors:

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

(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`
(4) University of Kiel, Germany, `jrt@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, 27] 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 [27]. 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].

Acknowledgements

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 D 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 6).

- 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 `AllSolutions` (Section A.2.1) defines the operation

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

  to compute all values of a given argument expression. There is also the library `SearchTree` (Section A.3.8) 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* [3] which compute the set of all results but do not encapsulate non-determinism in the actual arguments. See the library `SetFunctions` (Section A.2.36) 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` (Section A.2.39). 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.

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

$\boxed{\textbf{:quit}}$ Exit the system.

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

$\boxed{\textbf{:type } expr}$ Show the type of the expression $expr$.

$\boxed{\textbf{:programs}}$ Show the list of all Curry programs that are available in the load path.

$\boxed{\textbf{:cd } dir}$ Change the current working directory to dir.

$\boxed{\textbf{:edit}}$ 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.

$\boxed{\textbf{:edit } m}$ 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".

$\boxed{\textbf{:show}}$ 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.

$\boxed{\textbf{:show } m}$ Show the source text of module m which must be accessible via the current load path if no path specification is given.

$\boxed{\textbf{:source } f}$ Show the source code of function f (which must be visible in the currently loaded module) in a separate window.

$\boxed{\textbf{:source } m.f}$ Show the source code of function f defined in module m in a separate window.

$\boxed{\textbf{:browse}}$ Start the CurryBrowser to analyze the currently loaded module together with all its imported modules (see Section 6 for more details).

$\boxed{\textbf{:interface}}$ 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.

$\boxed{\textbf{:interface } m}$ Similar to ":interface" but shows the interface of the module m which must be in the load path of KiCS2.

$\boxed{\textbf{:usedimports}}$ Show all calls to imported functions in the currently loaded module. This might be useful to see which import declarations are really necessary.

$\boxed{\textbf{:set } option}$ 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.

$\boxed{\texttt{prdfs}}$ 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.

$\boxed{\texttt{choices } n}$ 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.

$\boxed{\texttt{supply } i}$ (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.

$\boxed{\texttt{v}n}$ 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.

$\boxed{\texttt{prompt } p}$ 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> "
```

$\boxed{\texttt{+/-interactive}}$ 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
```

$\boxed{\texttt{parser } opts}$ 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.

$\boxed{\texttt{cmp } opts}$ 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.

$\boxed{\texttt{ghc } opts}$ 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
```

$\boxed{\texttt{rts } opts}$ 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 in-
stance, setting the option

```
:set rts -H512m
```

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

 `args` *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` (Section A.2.39)) 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
>
```

17

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 [2] 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 [2] 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 [2] 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.

4 Recognized Syntax of Curry

The KiCS2 Curry compiler accepts a slightly extended version of the grammar specified in the Curry Report [27]. 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:

$$
\begin{array}{rl}
NonTerm ::= \alpha & \text{production} \\
NonTerm & \text{nonterminal symbol} \\
\texttt{Term} & \text{terminal symbol} \\
[\alpha] & \text{optional} \\
\{\alpha\} & \text{zero or more repetitions} \\
(\alpha) & \text{grouping} \\
\alpha \mid \beta & \text{alternative} \\
\alpha_{\langle\beta\rangle} & \text{difference – elements generated by } \alpha \\
& \text{without those generated by } \beta
\end{array}
$$

The Curry files are expected to be encoded in UTF8. 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.

$$
\begin{array}{rl}
Letter ::= & \text{any ASCII letter} \\
Dashes ::= & \texttt{--} \ \{\texttt{-}\} \\[4pt]
Ident ::= & (Letter \ \{Letter \mid Digit \mid \texttt{\_} \mid \texttt{'}\})_{\langle ReservedID \rangle} \\
Symbol ::= & \texttt{\textasciitilde} \mid \texttt{!} \mid \texttt{@} \mid \texttt{\#} \mid \texttt{\$} \mid \texttt{\%} \mid \texttt{\^{}} \mid \texttt{\&} \mid \texttt{*} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{?} \mid \texttt{.} \mid \texttt{/} \mid \texttt{|} \mid \texttt{\textbackslash} \mid \texttt{:} \\[4pt]
ModuleID ::= & \{Ident \ \texttt{.}\} \ Ident \\
TypeConstrID ::= & Ident \\
TypeVarID ::= & Ident \mid \texttt{\_} \\
DataConstrID ::= & Ident
\end{array}
$$

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

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

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

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

Note that the identifiers `as`, `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.

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

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:

$$\begin{aligned}
\textit{Char} ::=\ & \texttt{'}\ (\ \textit{Graphic}_{\langle\texttt{\textbackslash}\rangle} \mid \textit{Space} \mid \textit{Escape}_{\langle\texttt{\textbackslash\&}\rangle}\ )\ \texttt{'} \\
\textit{String} ::=\ & \texttt{"}\ \{\ \textit{Graphic}_{\langle\texttt{"}\ |\ \texttt{\textbackslash}\rangle} \mid \textit{Space} \mid \textit{Escape} \mid \textit{Gap}\ \}\ \texttt{"} \\
\textit{Escape} ::=\ & \texttt{\textbackslash}\ (\ \textit{CharEsc} \mid \textit{AsciiEsc} \mid \textit{Decimal} \mid \texttt{o}\ \textit{Octal} \mid \texttt{x}\ \textit{Hexadecimal}\ ) \\
\textit{CharEsc} ::=\ & \texttt{a} \mid \texttt{b} \mid \texttt{f} \mid \texttt{n} \mid \texttt{r} \mid \texttt{t} \mid \texttt{v} \mid \texttt{\textbackslash} \mid \texttt{"} \mid \texttt{'} \mid \texttt{\&} \\
\textit{AsciiAsc} ::=\ & \texttt{\textasciicircum}\ \textit{Cntrl} \mid \texttt{NUL} \mid \texttt{SOH} \mid \texttt{STX} \mid \texttt{ETX} \mid \texttt{EOT} \mid \texttt{ENQ} \mid \texttt{ACK}
\end{aligned}$$

```
           |  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.[2]

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.

4.4 Context-Free Grammar

$$Module ::= \texttt{module } ModuleID \ [Exports] \ \texttt{where } Block$$
$$\qquad\quad | \quad Block$$

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

---

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

$$
\begin{array}{rcll}
\textit{Exports} & ::= & \texttt{(} \ \textit{Export}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{Export}_n \ \texttt{)} & (n \geq 0) \\
\textit{Export} & ::= & \textit{QFunction} & \\
& | & \textit{QTypeConstrID} \ [\texttt{(} \ \textit{ConsLabel}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{ConsLabel}_n \ \texttt{)}] & (n \geq 0) \\
& | & \textit{QTypeConstrID} \ \texttt{(..)} & \\
& | & \texttt{module} \ \textit{ModuleID} & \\
\textit{ConsLabel} & ::= & \textit{DataConstr} \ | \ \textit{Label} & \\
\end{array}
$$

$$
\begin{array}{rcll}
\textit{ImportDecls} & ::= & \textit{ImportDecl}_1 \ \texttt{;} \ \dots \ \texttt{;} \ \textit{ImportDecl}_n & (n \geq 1) \\
\textit{ImportDecl} & ::= & \texttt{import} \ [\texttt{qualified}] \ \textit{ModuleID} \ [\texttt{as} \ \textit{ModuleID}] \ [\textit{ImportSpec}] & \\
\textit{ImportSpec} & ::= & \texttt{(} \ \textit{Import}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{Import}_n \ \texttt{)} & (n \geq 0) \\
& | & \texttt{hiding} \ \texttt{(} \ \textit{Import}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{Import}_n \ \texttt{)} & (n \geq 0) \\
\textit{Import} & ::= & \textit{Function} & \\
& | & \textit{TypeConstrID} \ [\texttt{(} \ \textit{ConsLabel}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{ConsLabel}_n \ \texttt{)}] & (n \geq 0) \\
& | & \textit{TypeConstrID} \ \texttt{(..)} & \\
\end{array}
$$

$$
\begin{array}{rcll}
\textit{BlockDecl} & ::= & \textit{TypeSynDecl} & \\
& | & \textit{DataDecl} & \\
& | & \textit{NewtypeDecl} & \\
& | & \textit{FixityDecl} & \\
& | & \textit{FunctionDecl} & \\
\end{array}
$$

$$
\begin{array}{rcll}
\textit{TypeSynDecl} & ::= & \texttt{type} \ \textit{SimpleType} \ \texttt{=} \ \textit{TypeExpr} & \\
\textit{SimpleType} & ::= & \textit{TypeConstrID} \ \textit{TypeVarID}_1 \ \dots \ \textit{TypeVarID}_n & (n \geq 0) \\
\end{array}
$$

$$
\begin{array}{rcll}
\textit{DataDecl} & ::= & \texttt{data} \ \textit{SimpleType} & (\textit{external data type}) \\
& | & \texttt{data} \ \textit{SimpleType} \ \texttt{=} \ \textit{ConstrDecl}_1 \ \texttt{|} \ \dots \ \texttt{|} \ \textit{ConstrDecl}_n & (n \geq 1) \\
\textit{ConstrDecl} & ::= & \textit{DataConstr} \ \textit{SimpleTypeExpr}_1 \ \dots \ \textit{SimpleTypeExpr}_n & (n \geq 0) \\
& | & \textit{TypeConsExpr} \ \textit{ConOp} \ \textit{TypeConsExpr} & (\textit{infix data constructor}) \\
& | & \textit{DataConstr} \ \texttt{\{} \ \textit{FieldDecl}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{FieldDecl}_n \ \texttt{\}} & (n \geq 0) \\
\textit{FieldDecl} & ::= & \textit{Label}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{Label}_n \ \texttt{::} \ \textit{TypeExpr} & (n \geq 1) \\
\end{array}
$$

$$
\begin{array}{rcll}
\textit{NewtypeDecl} & ::= & \texttt{newtype} \ \textit{SimpleType} \ \texttt{=} \ \textit{NewConstrDecl} & \\
\textit{NewConstrDecl} & ::= & \textit{DataConstr} \ \textit{SimpleTypeExpr} & \\
& | & \textit{DataConstr} \ \texttt{\{} \ \textit{Label} \ \texttt{::} \ \textit{TypeExpr} \ \texttt{\}} & \\
\end{array}
$$

$$
\begin{array}{rcll}
\textit{TypeExpr} & ::= & \textit{TypeConsExpr} \ [\texttt{->} \ \textit{TypeExpr}] & \\
\textit{TypeConsExpr} & ::= & \textit{QTypeConstrID} \ \textit{SimpleTypeExpr}_1 \ \dots \ \textit{SimpleTypeExpr}_n & (n \geq 1) \\
& | & \textit{SimpleTypeExpr} & \\
\textit{SimpleTypeExpr} & ::= & \textit{TypeVarID} & \\
& | & \textit{QTypeConstrID} & \\
& | & \texttt{()} & (\textit{unit type}) \\
& | & \texttt{(} \ \textit{TypeExpr}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{TypeExpr}_n \ \texttt{)} & (\textit{tuple type, } n \geq 2) \\
& | & \texttt{[} \ \textit{TypeExpr} \ \texttt{]} & (\textit{list type}) \\
& | & \texttt{(} \ \textit{TypeExpr} \ \texttt{)} & (\textit{parenthesized type}) \\
\end{array}
$$

$$
\begin{array}{rcll}
\textit{FixityDecl} & ::= & \textit{Fixity} \ [\textit{Int}] \ \textit{Op}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{Op}_n & (n \geq 1) \\
\textit{Fixity} & ::= & \texttt{infixl} \ | \ \texttt{infixr} \ | \ \texttt{infix} & \\
\end{array}
$$

$$
\begin{array}{rcll}
\textit{FunctionDecl} & ::= & \textit{Signature} \ | \ \textit{ExternalDecl} \ | \ \textit{Equation} & \\
\textit{Signature} & ::= & \textit{Functions} \ \texttt{::} \ \textit{TypeExpr} & \\
\textit{ExternalDecl} & ::= & \textit{Functions} \ \texttt{external} & (\textit{externally defined functions}) \\
\textit{Functions} & ::= & \textit{Function}_1 \ \texttt{,} \ \dots \ \texttt{,} \ \textit{Function}_n & (n \geq 1) \\
\textit{Equation} & ::= & \textit{FunLhs} \ \textit{Rhs} & \\
\textit{FunLhs} & ::= & \textit{Function} \ \textit{SimplePat}_1 \ \dots \ \textit{SimplePat}_n & (n \geq 0) \\
\end{array}
$$

$$
\begin{aligned}
&\qquad\qquad\quad | \;\; \textit{ConsPattern FunOp ConsPattern} \\
&\qquad\qquad\quad | \;\; (\;\textit{FunLhs}\;)\;\textit{SimplePat}_1\;\ldots\;\textit{SimplePat}_n \qquad\qquad\qquad\qquad (n \geq 1)
\end{aligned}
$$

$$
\begin{aligned}
\textit{Rhs} ::=&\; \texttt{=}\;\textit{Expr}\;[\texttt{where}\;\textit{LocalDecls}] \\
|&\;\; \textit{CondExprs}\;[\texttt{where}\;\textit{LocalDecls}]
\end{aligned}
$$

$$
\textit{CondExprs} ::= \texttt{|}\;\textit{InfixExpr}\;\texttt{=}\;\textit{Expr}\;[\textit{CondExprs}]
$$

$$
\textit{LocalDecls} ::= \{\;\textit{LocalDecl}_1\;\texttt{;}\;\ldots\;\texttt{;}\;\textit{LocalDecl}_n\;\} \qquad\qquad\qquad\qquad (n \geq 0)
$$

$$
\begin{aligned}
\textit{LocalDecl} ::=&\; \textit{FunctionDecl} \\
|&\;\; \textit{PatternDecl} \\
|&\;\; \textit{Variable}_1\;\texttt{,}\;\ldots\;\texttt{,}\;\textit{Variable}_n\;\texttt{free} \qquad\qquad\qquad\qquad (n \geq 1) \\
|&\;\; \textit{FixityDecl}
\end{aligned}
$$

$$
\textit{PatternDecl} ::= \textit{Pattern Rhs}
$$

$$
\textit{Pattern} ::= \textit{ConsPattern}\;[\textit{QConOp Pattern}] \qquad\qquad\qquad (\textit{infix constructor pattern})
$$

$$
\begin{aligned}
\textit{ConsPattern} ::=&\; \textit{GDataConstr SimplePat}_1\;\ldots\;\textit{SimplePat}_n \qquad (\textit{constructor pattern, } n \geq 1) \\
|&\;\; \texttt{-}\;\textit{Int} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\textit{negative integer pattern}) \\
|&\;\; \texttt{-.}\;\textit{Float} \qquad\qquad\qquad\qquad\qquad\qquad\quad (\textit{negative float pattern}) \\
|&\;\; \textit{SimplePat}
\end{aligned}
$$

$$
\begin{aligned}
\textit{SimplePat} ::=&\; \textit{Variable} \\
|&\;\; \texttt{\_} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textit{wildcard}) \\
|&\;\; \textit{GDataConstr} \qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textit{constructor}) \\
|&\;\; \textit{Literal} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\textit{literal}) \\
|&\;\; (\;\textit{Pattern}\;) \qquad\qquad\qquad\qquad\qquad (\textit{parenthesized pattern}) \\
|&\;\; (\;\textit{Pattern}_1\;\texttt{,}\;\ldots\;\texttt{,}\;\textit{Pattern}_n\;) \qquad\qquad (\textit{tuple pattern, } n \geq 2) \\
|&\;\; [\;\textit{Pattern}_1\;\texttt{,}\;\ldots\;\texttt{,}\;\textit{Pattern}_n\;] \qquad\qquad (\textit{list pattern, } n \geq 1) \\
|&\;\; \textit{Variable}\;\texttt{@}\;\textit{SimplePat} \qquad\qquad\qquad\qquad\qquad (\textit{as-pattern}) \\
|&\;\; \texttt{\textasciitilde}\;\textit{SimplePat} \qquad\qquad\qquad\qquad\qquad\qquad (\textit{irrefutable pattern}) \\
|&\;\; (\;\textit{QFunction SimplePat}_1\;\ldots\;\textit{SimplePat}_n\;) \quad (\textit{functional pattern, } n \geq 1) \\
|&\;\; (\;\textit{ConsPattern QFunOp Pattern}\;) \qquad (\textit{infix functional pattern}) \\
|&\;\; \textit{QDataConstr}\;\{\;\textit{FieldPat}_1\;\texttt{,}\;\ldots\;\texttt{,}\;\textit{FieldPat}_n\;\} \quad (\textit{labeled pattern, } n \geq 0)
\end{aligned}
$$

$$
\textit{FieldPat} ::= \textit{QLabel}\;\texttt{=}\;\textit{Pattern}
$$

$$
\begin{aligned}
\textit{Expr} ::=&\; \textit{InfixExpr}\;\texttt{::}\;\textit{TypeExpr} \qquad (\textit{expression with type signature}) \\
|&\;\; \textit{InfixExpr}
\end{aligned}
$$

$$
\begin{aligned}
\textit{InfixExpr} ::=&\; \textit{NoOpExpr QOp InfixExpr} \qquad (\textit{infix operator application}) \\
|&\;\; \texttt{-}\;\textit{InfixExpr} \qquad\qquad\qquad\qquad\qquad\qquad (\textit{unary int minus}) \\
|&\;\; \texttt{-.}\;\textit{InfixExpr} \qquad\qquad\qquad\qquad\qquad\quad (\textit{unary float minus}) \\
|&\;\; \textit{NoOpExpr}
\end{aligned}
$$

$$
\begin{aligned}
\textit{NoOpExpr} ::=&\; \texttt{\textbackslash}\;\textit{SimplePat}_1\;\ldots\;\textit{SimplePat}_n\;\texttt{->}\;\textit{Expr} \quad (\textit{lambda expression, } n \geq 1) \\
|&\;\; \texttt{let}\;\textit{LocalDecls}\;\texttt{in}\;\textit{Expr} \qquad\qquad\qquad\qquad (\textit{let expression}) \\
|&\;\; \texttt{if}\;\textit{Expr}\;\texttt{then}\;\textit{Expr}\;\texttt{else}\;\textit{Expr} \qquad\qquad\qquad (\textit{conditional}) \\
|&\;\; \texttt{case}\;\textit{Expr}\;\texttt{of}\;\{\;\textit{Alt}_1\;\texttt{;}\;\ldots\;\texttt{;}\;\textit{Alt}_n\;\} \quad (\textit{case expression, } n \geq 1) \\
|&\;\; \texttt{fcase}\;\textit{Expr}\;\texttt{of}\;\{\;\textit{Alt}_1\;\texttt{;}\;\ldots\;\texttt{;}\;\textit{Alt}_n\;\} \quad (\textit{fcase expression, } n \geq 1) \\
|&\;\; \texttt{do}\;\{\;\textit{Stmt}_1\;\texttt{;}\;\ldots\;\texttt{;}\;\textit{Stmt}_n\;\texttt{;}\;\textit{Expr}\;\} \quad (\textit{do expression, } n \geq 0) \\
|&\;\; \textit{FuncExpr}
\end{aligned}
$$

$$
\textit{FuncExpr} ::= [\textit{FuncExpr}]\;\textit{BasicExpr} \qquad\qquad\qquad\qquad\qquad (\textit{application})
$$

$$
\begin{aligned}
\textit{BasicExpr} ::=&\; \textit{Variable} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textit{variable}) \\
|&\;\; \texttt{\_} \qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textit{anonymous free variable}) \\
|&\;\; \textit{QFunction} \qquad\qquad\qquad\qquad\qquad\qquad (\textit{qualified function}) \\
|&\;\; \textit{GDataConstr} \qquad\qquad\qquad\qquad\qquad (\textit{general constructor}) \\
|&\;\; \textit{Literal} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\textit{literal})
\end{aligned}
$$

$$
\begin{aligned}
&\quad | \quad (\ \textit{Expr}\ ) &&(\textit{parenthesized expression})\\
&\quad | \quad (\ \textit{Expr}_1\ ,\ \ldots\ ,\ \textit{Expr}_n\ ) &&(\textit{tuple},\ n \geq 2)\\
&\quad | \quad [\ \textit{Expr}_1\ ,\ \ldots\ ,\ \textit{Expr}_n\ ] &&(\textit{finite list},\ n \geq 1)\\
&\quad | \quad [\ \textit{Expr}\ [,\ \textit{Expr}]\ ..\ [\textit{Expr}]\ ] &&(\textit{arithmetic sequence})\\
&\quad | \quad [\ \textit{Expr}\ |\ \textit{Qual}_1\ ,\ \ldots\ ,\ \textit{Qual}_n\ ] &&(\textit{list comprehension},\ n \geq 1)\\
&\quad | \quad (\ \textit{InfixExpr QOp}\ ) &&(\textit{left section})\\
&\quad | \quad (\ \textit{QOp}_{\langle -,\ -.\rangle}\ \textit{InfixExpr}\ ) &&(\textit{right section})\\
&\quad | \quad \textit{QDataConstr}\ \{\ \textit{FBind}_1\ ,\ \ldots\ ,\ \textit{FBind}_n\ \} &&(\textit{record construction},\ n \geq 0)\\
&\quad | \quad \textit{BasicExpr}_{\langle QDataConstr\rangle}\ \{\ \textit{FBind}_1\ ,\ \ldots\ ,\ \textit{FBind}_n\ \} &&(\textit{record update},\ n \geq 1)\\[4pt]
\textit{Alt} ::=\ &\textit{Pattern}\ \verb|->|\ \textit{Expr}\ [\verb|where|\ \textit{LocalDecls}]\\
&\quad | \quad \textit{Pattern GdAlts}\ [\verb|where|\ \textit{LocalDecls}]\\
\textit{GdAlts} ::=\ &|\ \textit{InfixExpr}\ \verb|->|\ \textit{Expr}\ [\textit{GdAlts}]\\[4pt]
\textit{FBind} ::=\ &\textit{QLabel}\ \verb|=|\ \textit{Expr}\\[4pt]
\textit{Qual} ::=\ &\textit{Pattern}\ \verb|<-|\ \textit{Expr} &&(\textit{generator})\\
&\quad | \quad \verb|let|\ \textit{LocalDecls} &&(\textit{local declarations})\\
&\quad | \quad \textit{Expr} &&(\textit{guard})\\[4pt]
\textit{Stmt} ::=\ &\textit{Pattern}\ \verb|<-|\ \textit{Expr}\\
&\quad | \quad \verb|let|\ \textit{LocalDecls}\\
&\quad | \quad \textit{Expr}\\[4pt]
\textit{Literal} ::=\ &\textit{Int}\ |\ \textit{Float}\ |\ \textit{Char}\ |\ \textit{String}\\[4pt]
\textit{GDataConstr} ::=\ &\verb|()| &&(\textit{unit})\\
&\quad | \quad \verb|[]| &&(\textit{empty list})\\
&\quad | \quad \verb|(,{,})| &&(\textit{tuple})\\
&\quad | \quad \textit{QDataConstr}\\[4pt]
\textit{Variable} ::=\ &\textit{VariableID}\ |\ (\ \textit{InfixOpID}\ ) &&(\textit{variable})\\
\textit{Function} ::=\ &\textit{FunctionID}\ |\ (\ \textit{InfixOpID}\ ) &&(\textit{function})\\
\textit{QFunction} ::=\ &\textit{QFunctionID}\ |\ (\ \textit{QInfixOpID}\ ) &&(\textit{qualified function})\\
\textit{DataConstr} ::=\ &\textit{DataConstrID}\ |\ (\ \textit{InfixOpID}\ ) &&(\textit{constructor})\\
\textit{QDataConstr} ::=\ &\textit{QDataConstrID}\ |\ (\ \textit{QInfixOpID}\ ) &&(\textit{qualified constructor})\\
\textit{Label} ::=\ &\textit{LabelID}\ |\ (\ \textit{InfixOpID}\ ) &&(\textit{label})\\
\textit{QLabel} ::=\ &\textit{QLabelID}\ |\ (\ \textit{QInfixOpID}\ ) &&(\textit{qualified label})\\[4pt]
\textit{VarOp} ::=\ &\textit{InfixOpID}\ |\ \grave{}\ \textit{VariableID}\ \grave{} &&(\textit{variable operator})\\
\textit{FunOp} ::=\ &\textit{InfixOpID}\ |\ \grave{}\ \textit{FunctionID}\ \grave{} &&(\textit{function operator})\\
\textit{QFunOp} ::=\ &\textit{QInfixOpID}\ |\ \grave{}\ \textit{QFunctionID}\ \grave{} &&(\textit{qualified function operator})\\
\textit{ConOp} ::=\ &\textit{InfixOpID}\ |\ \grave{}\ \textit{DataConstrID}\ \grave{} &&(\textit{constructor operator})\\
\textit{QConOp} ::=\ &\textit{GConSym}\ |\ \grave{}\ \textit{QDataConstrID}\ \grave{} &&(\textit{qualified constructor operator})\\
\textit{LabelOp} ::=\ &\textit{InfixOpID}\ |\ \grave{}\ \textit{LabelID}\ \grave{} &&(\textit{label operator})\\
\textit{QLabelOp} ::=\ &\textit{QInfixOpID}\ |\ \grave{}\ \textit{QLabelID}\ \grave{} &&(\textit{qualified label operator})\\[4pt]
\textit{Op} ::=\ &\textit{FunOp}\ |\ \textit{ConOp}\ |\ \textit{LabelOp} &&(\textit{operator})\\
\textit{QOp} ::=\ &\textit{VarOp}\ |\ \textit{QFunOp}\ |\ \textit{QConOp}\ |\ \textit{QLabelOp} &&(\textit{qualified operator})\\
\textit{GConSym} ::=\ &\verb|:|\ |\ \textit{QInfixOpID} &&(\textit{general constructor symbol})
\end{aligned}
$$

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 [6]. 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 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 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. 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 [6]) 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 ...
```

6 `curry browse`: A Tool for Analyzing and Browsing Curry Programs

CurryBrowser is a tool to browse through the modules and functions of a Curry application, show them in various formats, and analyze their properties.[3] Moreover, it is constructed in a way so that new analyzers can be easily connected to CurryBrowser. A detailed description of the ideas behind this tool can be found in [20, 21].

CurryBrowser is part of the KiCS2 distribution and can be started in two ways:

- In the command shell via the command: *kics2home*/bin/curry browse mod

- In the KiCS2 environment after loading the module `mod` and typing the command ":`browse`".

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 [28] 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.

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

---

[3]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.
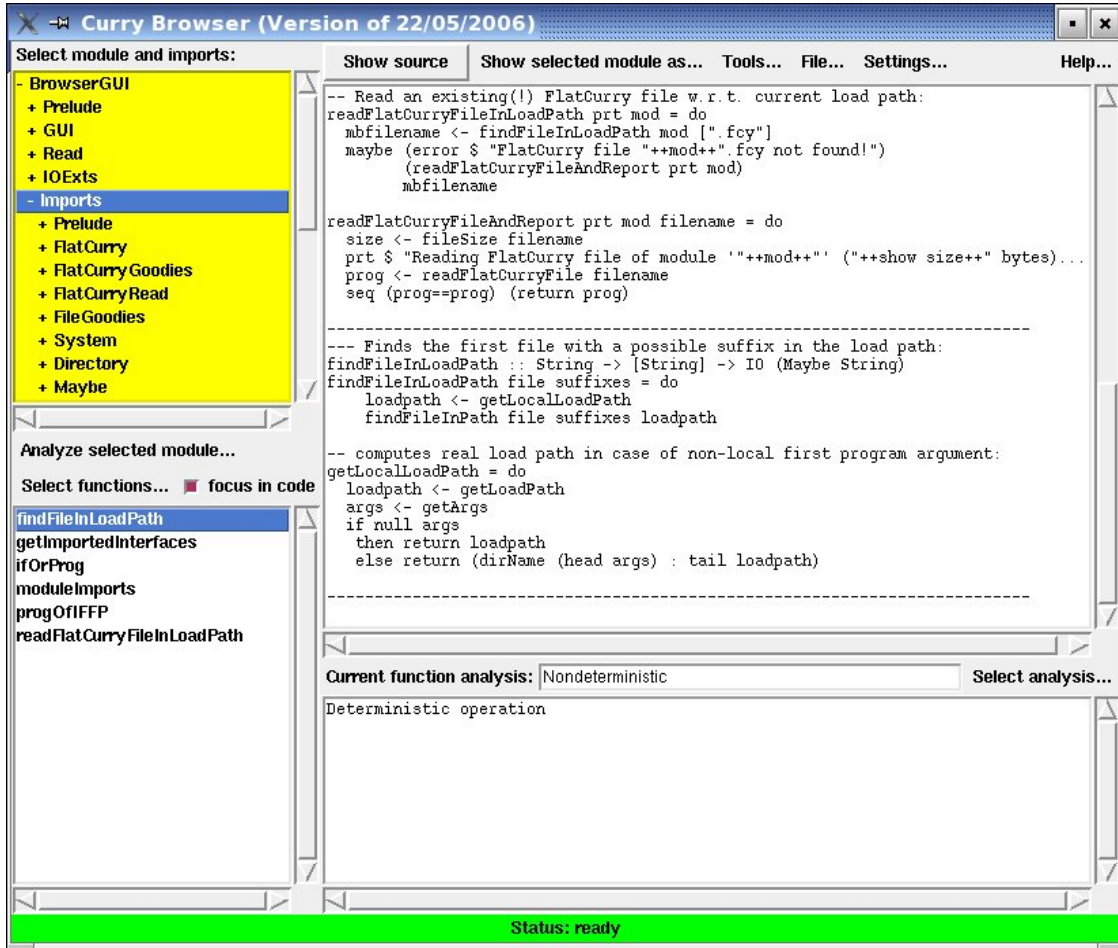
Figure 1: Snapshot of the main window of CurryBrowser

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.

7 `curry check`: 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 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`,[4] 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

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
```

---

[4]The library `Test.Prop` is a clone of the library `Test.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.

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.

7.2   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 genτ, 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
```

7.3 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 [4]. 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 [4], 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 | ys == perm xs && sorted ys = ys   where ys free

-- 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 yields the same values as the 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 =
  let r = sort x
  in (r == r) ==> always (sort'post x r)

sortSatisfiesSpecification :: [Int]  →  Prop
sortSatisfiesSpecification x = sort x <~> sort'spec x
```

7.4   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 [2]. It should not be used in source programs to avoid unintended uses. Hence, CurryCheck reports such unintended uses.

- Set functions [3] 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 module `SetFunctions`, see Section A.2.36) 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   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.

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.[5] 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).

`@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's syntax (the currently supported set of elements is described in detail in the appendix). 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

---

[5]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.

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]
```

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`, `M2`,...,`Mod`n and their imported modules) in the directory `docdir`.

9 `curry style`: A Style Checker for Curry Programs

CASC is a tool to check the formatting style of Curry programs. The preferred style for writing Curry programs, which is partially checked by this tool, is described in a separate web page[6] Currently, CASC only checks a few formatting rules, like line lengths or indentation of `if-then-else`, but the kind of checks performed by CASC will be extended in the future.

9.1 Basic Usage

To check the style of some Curry program stored in the file `prog.curry`, one can invoke the style checker by the command

```
curry style prog
```

After processing the program, a list of all positions with stylistic errors is printed.

9.2 Configuration

CASC can be configured so that not all stylistic rules are checked. For this purpose, one should copy the global configuration file of CASC, which is stored in *kics2home*`/currytools/casc/cascrc` (where *kics2home* is the installation directory of KiCS2), into the home directory under the name "`.cascrc`". Then one can configure this file according to your own preferences, which are described in this file.

---

[6]http://www.informatik.uni-kiel.de/~pakcs/CurryStyleGuide.html

10 `curry test`: A Tool for Testing Curry Programs

General remark: *The CurryTest tool described in this section has been replaced by the more advanced tool CurryCheck (see Section 7). CurryTest is still available in KiCS2 but is no more supported. Hence, it is recommended to use CurryCheck for writing test cases.*

CurryTest is a simple tool in the KiCS2 distribution to write and run repeatable tests. CurryTest simplifies the task of writing test cases for a module and executing them. The tool is easy to use. Assume one has implemented a module `MyMod` and wants to write some test cases to test its functionality, making regression tests in future versions, etc. For this purpose, there is a system library `Assertion` (Section A.2.2) which contains the necessary definitions for writing tests. In particular, it exports an abstract polymorphic type "`Assertion a`" together with the following operations:

```
assertTrue      :: String  →  Bool  →  Assertion ()
assertEqual     :: String  →  a  →  a  →  Assertion a
assertValues    :: String  →  a  →  [a]  →  Assertion a
assertSolutions :: String  →  (a →Bool)  →  [a]  →  Assertion a
assertIO        :: String  →  IO a  →  a  →  Assertion a
assertEqualIO   :: String  →  IO a  →  IO a  →  Assertion a
```

The expression "`assertTrue s b`" is an assertion (named s) that the expression b has the value `True`. Similarly, the expression "`assertEqual s e_1 e_2`" asserts that the expressions e_1 and e_2 must be equal (i.e., $e_1 == e_2$ must hold), the expression "`assertValues s e vs`" asserts that vs is the multiset of all values of e, and the expression "`assertSolutions s c vs`" asserts that the constraint abstraction c has the multiset of solutions vs. Furthermore, the expression "`assertIO s a v`" asserts that the I/O action a yields the value v whenever it is executed, and the expression "`assertEqualIO s a_1 a_2`" asserts that the I/O actions a_1 and a_2 yield equal values. The name s provided as a first argument in each assertion is used in the protocol produced by the test tool.

One can define a test program by importing the module to be tested together with the module `Assertion` and defining top-level functions of type `Assertion` in this module (which must also be exported). As an example, consider the following program that can be used to test some list processing functions:

```
import List
import Assertion

test1 = assertEqual      "++"     ([1,2]++[3,4]) [1,2,3,4]

test2 = assertTrue       "all"    (all (<5) [1,2,3,4])

test3 = assertSolutions "prefix" (\x  →  x++_ =:= [1,2])
                                  [[],[1],[1,2]]
```

For instance, `test1` asserts that the result of evaluating the expression (`[1,2]++[3,4]`) is equal to `[1,2,3,4]`.

We can execute a test suite by the command

```
curry test TestList
```
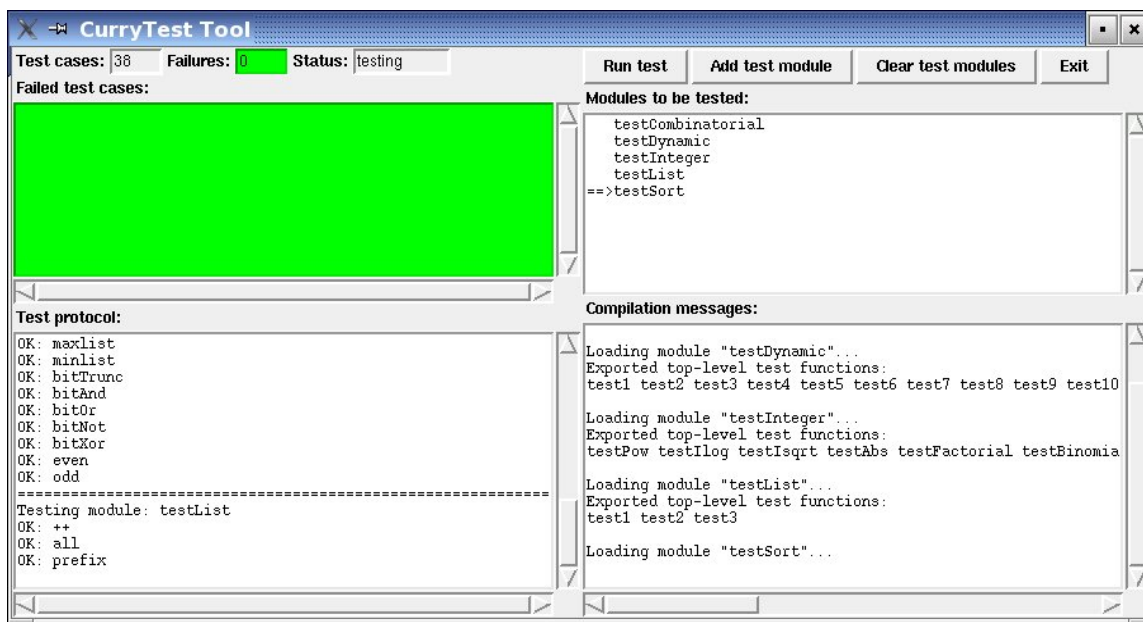
Figure 2: Snapshot of CurryTest's graphical interface

In our example, "`TestList.curry`" is the program containing the definition of all assertions. This has the effect that all exported top-level functions of type `Assertion` are tested (i.e., the corresponding assertions are checked) and the results ("`OK`" or failure) are reported together with the name of each assertion. For our example above, we obtain the following successful protocol:

```
=============================================================
Testing module "TestList"...
OK: ++
OK: all
OK: prefix
All tests successfully passed.
=============================================================
```

There is also a graphical interface that summarizes the results more nicely. In order to start this interface, one has to add the parameter "`--window`" (or "`-w`"), e.g., executing a test suite by

```
curry test --window TestList
```

or

```
curry test -w TestList
```

A snapshot of the interface is shown in Figure 2.

11 `curry verify`: A Tool to Support the Verification of Curry Programs

Curry2Verify is a tool that supports the verification of Curry programs with the help of other theorem provers or proof assistants. Basically, Curry2Verify extends CurryCheck (see Section 7), which tests given properties of a program, by the possibility to verify these properties. For this purpose, Curry2Verify 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.

11.1 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 3, 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 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 4.

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

```
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 3: Curry program `Double.curry`

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.

11.2 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 4: Agda program TO-PROVE-evendoublecoin.agda

12 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.

Sequential rules: If this feature is used, all rules in a Curry module are interpreted as sequential, i.e., a rule is applied only if all previous rules defining the same operation are not applicable. The idea of sequential rules are described in [5].

Default rules: If this feature is used, one can add a default rule to operations defined in a Curry module. This provides a similar power than sequential rules but with a better operational behavior. The idea of default rules is described in [7].

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 [4].

The preprocessor is an executable named "`currypp`", which is stored in the directory *kics2home*/bin. 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_CYMAKE -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 `currypp` before actually parsing the source text.

The option "`defaultrules`" has to be replaced by "`seqrules`" if the sequential rule matching should be replaced, or by "`contracts`" to enable dynamic contract checking. To support integrated code, one has to set the option "`foreigncode`" (which can also be combined with either "`defaultrules`" or "`seqrules`"). 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_CYMAKE -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 (except for "`seqrules`") are applied. Thus, the preprocessor directive

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

is equivalent to

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

12.1 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., `regexp` or `html` (see below). For
instance, if one uses regular expressions (see below for more details), the following expressions are
valid in source programs:

```
s ``regex (a|(bc*))+''
s ````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_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}
```

The next sections describe the currently supported foreign languages.

12.1.1 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 "`regexp`". 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_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import RegExp

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

12.1.2 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 KiCS2 library `Format` 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_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}
```

```
import 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
```

12.1.3 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_CYMAKE -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_CYMAKE -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}''
```

12.1.4 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_CYMAKE -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
 ''
```

12.2   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 12.2.1). 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.

12.2.1   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 5 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:

```
 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)
```
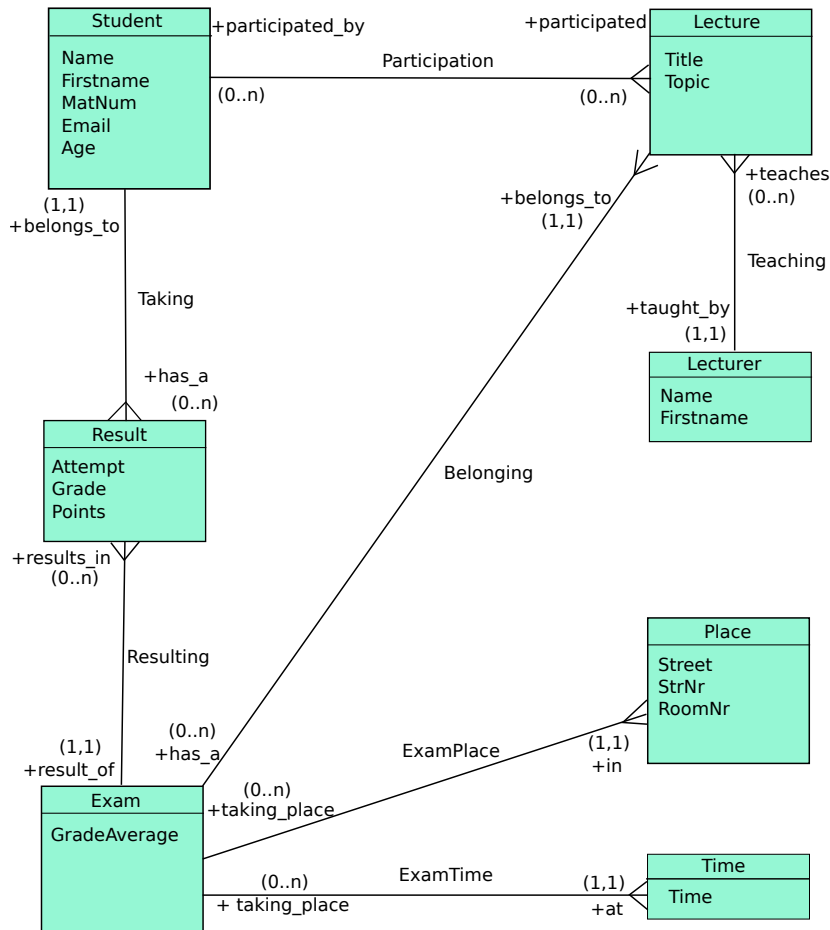
Figure 5: A simple entity-relationship diagram for university lectures [24]

```
| 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.5 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 6 on (page 62) shows the complete ER data term specification corresponding to the ERD of Fig. 5.

If such a data term specification is stored in file `UniERD.term`, then one can use the tool "erd2cdbi", which is stored in the directory *kics2home*/bin, to process the ER model so that it can be used in SQL statements. This tool is invoked with the name of the term file and the (preferably absolute) file name of the SQLite database. If the later does not exist, it will be initialized by the tool. In our example, we execute the following command (provided that the directory *kics2home*/bin is in the path):

```
> erd2cdbi Uni_ERD.term 'pwd'/Uni.db
```

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). This task is performed by the tool `erd2curry` (see Sect. 15).

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

57

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.

12.2.2 SQL Statements as Integrated Code

After specifying and processing the ER model of the database, one can write SQL statement in their standard syntax as integrated code (marked by the prefix "`sql`") in Curry programs. 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 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 = liftIO fromSQLResult ''sql Select * From Student;''
```

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 C. More details about the implementation of this SQL translator can be found in [24, 29].

12.3   Sequential Rules

If the Curry preprocessor is called with the option "`seqrules`", then all rules in the Curry module are interpreted in a sequential manner, i.e., a rule is applied only if all previous rules defining the same operation are not applicable, either because the left-hand side's pattern does not match or the condition is not satisfiable. The idea and detailed semantics of sequential rules are described in [5]. Sequential rules are useful and preferable over rules with multiple guards if the patterns are non-trivial (e.g., functional patterns) or the condition involve complex constraints.

As a simple example, the following module defines a lookup operation in association lists by a functional pattern. Due to the sequential rule strategy, the second rule is applied only if there is no appropriate key in the association list:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=seqrules #-}

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

12.4   Default Rules

An alternative to sequential rules are default rules, i.e., these two options cannot be simultaneously used. Default rules are activated by the preprocessor option "`defaultrules`". In this case, one can add to each 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 [7].

Default rules are preferable over the sequential rule selection strategy since they have a better operational behavior. This is due to the fact that the test for the application of default rules is done with the same (sometimes optimal) strategy than the selection of standard rules. Moreover,

default rules provide a similar power than sequential rules, i.e., they can be applied if the standard rules have complex (functional) patterns or complex conditions.

As a simple example, we show the implementation of the previous example for sequential rules with a default rule:

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

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_CYMAKE -F --pgmF=currypp --optF=defaultrules #-}

import Combinatorial(permute)
import Integer(abs)

-- A placement is safe if two queens are not in a same diagonal:
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])
```

12.5   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 [4]. 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 [4]. 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 has been shortly introduced in Section 7.3. 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 as shown in Section 7.3 (where the code for `sorted` and `perm` is not shown here):

```
{-# OPTIONS_CYMAKE -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!
```

```
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 6: The ER data term specification of Fig. 5

13 runcurry: Running Curry Programs

runcurry is a command usually stored in *kics2home*/bin (where *kics2home* is the installation directory of KiCS2; see Section 1.2). This command supports 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.

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. 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.

14 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.

14.1 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:

```
> curry analyze -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:

```
> curry analyze -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.

14.1.1 Batch Mode

In the batch mode, CASS is started as a separate application via the shell command `curry analyze`, where the analysis name and the name of the module to be analyzed must be provided:[7]

```
> curry analyze 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.

14.1.2 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 `AnalysisServer`)

```
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.

   The modules of the CASS implementation are stored in the directory *kics2home*/currytools/CASS and the modules implementing the various program analyses are stored in *kics2home*/currytools/analysis. Hence, one should add these directories to the Curry load path when using CASS in API mode.

   The CASS module `GenericProgInfo` 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 `Demandedness` by the following operation:

```
demandAnalysis :: Analysis DemandedArgs
```

---

[7]More output is generated when the parameter `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.

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

```
import AnalysisServer   (analyzeGeneric)
import GenericProgInfo  (lookupProgInfo)
import 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]
```

14.1.3   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 *kics2home*/currytools/CASS/Protocol.txt and sketched below.

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

```
> curry analyze --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

```
> curry analyze --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.
```

14.2   Implementing Program Analyses

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

```
Analysis a
```

where `a` denotes the type of analysis results. For instance, the `Overlapping` analysis is implemented as a function

```
overlapAnalysis :: Analysis Bool
```

where the Boolean analysis result indicates whether a Curry operation is defined by overlapping rules.

In order to add a new analysis to CASS, one has to implement a corresponding analysis operation, registering it in the module `Registry` (in the constant `registeredAnalysis`) and compile the modified CASS implementation.

An analysis is implemented as a mapping from Curry programs 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==(pre "?") || 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 enable the inclusion of different analyses in CASS, CASS offers several constructor operations for the abstract type "`Analysis a`" (defined in the CASS module `Analysis`). 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
...
overlapAnalysis :: Analysis Bool
overlapAnalysis = simpleFuncAnalysis "Overlapping" isOverlappingFunction
```

Another analysis constructor supports the definition of a function analysis with dependencies (which is implemented via a fixpoint computation):

```
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.

For instance, a determinism 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—it requires a fixpoint computation. CASS provides such fixpoint computations and requires 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.

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

70

```
detFunc :: FuncDecl  → [(QName,Deterministic)]  → Deterministic
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.

The complete determinism analysis can be specified as

```
detAnalysis :: Analysis Deterministic
detAnalysis = dependencyFuncAnalysis "Deterministic" Det detFunc
```

This definition is sufficient to execute the analysis with CASS since the analysis system takes care of computing fixpoints, calling the analysis functions with appropriate values, analyzing imported modules, etc. Nevertheless, the analysis must be defined so that the fixpoint computation always terminates. This can be achieved by using an abstract domain with finitely many values and ensuring that the analysis function is monotone w.r.t. some ordering on the values.

15 ERD2Curry: A Tool to Generate Programs from ER Specifications

ERD2Curry is a tool to generate Curry code to access and manipulate data persistently stored from entity relationship diagrams. The idea of this tool is described in detail in [11]. Thus, we describe only the basic steps to use this tool in the following.

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

```
curry 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`). If this definition is stored in the Curry program file "`MyERD.curry`", it can be compiled into a Curry program by the command

```
curry erd2curry MyERD.curry
```

The directory *kics2home*`/currytools/erd2curry/` 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].

There is also the possibility to visualize an ERD term 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

```
curry erd2curry -v MyERD.curry
```

16 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.

The idea of this tool, which is part of the distribution of KiCS2, is described in detail in [23]. Thus, we describe 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 given in the system library `Database.ERD`) and store it in some program file, e.g., "MyERD.curry". The directory *kics2home*/`currytools/spicey/` 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 change to the directory in which you want to create the project sources. Execute the command

```
curry spiceup .../MyERD.curry
```

with the path to the ERD term file as a parameter You can also provide a path name, i.e., the name of a directory, where the database files should be stored, e.g.,

```
curry spiceup --dbpath DBDIR .../MyERD.curry
```

If the parameter "`--dbpath DBDIR`" is not provided, then DBDIR is set to the current directory ("."). Since this specification will be used in the *generated* web programs, a relative database directory name will be relative to the place where the web programs are stored. In order to avoid such confusion, it might be better to specify an absolute path name for the database directory.

After the generation of this project (see the generated file `README.txt` for 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`.

17 `curry peval`: A Partial Evaluator for Curry

peval is a tool for the partial evaluation of Curry programs. It operates on the FlatCurry representation and can thus easily be incorporated into the normal compilation chain. The essence of partial evaluation is to anticipate at compile time (or partial evaluation time) some of the computations normally performed at run time. Typically, partial evaluation is worthwhile for functions or operations where some of the input arguments are already known at compile time, or operations built by the composition of multiple other ones. The theoretical foundations, design and implementation of the partial evaluator is described in detail in [32].

17.1 Basic Usage

The partial evaluator is supplied as a binary that can be invoked for a single or multiple modules that should be partially evaluated. In each module, the partially evaluator assumes the parts of the program that should be partially evaluated to be annotated by the function

```
PEVAL :: a
PEVAL x = x
```

predefined in the module `Prelude`, such that the user can choose the parts to be considered.

To give an example, we consider the following module which is assumed to be placed in the file `Examples/power4.curry`:

```
square  x = x * x
even    x = mod x 2 == 0
power n x = if n <= 0 then 1
                      else if (even n) then power (div n 2) (square x)
                                       else x * (power (n - 1) x)
power4  x = PEVAL (power 4 x)
```

By the call to `PEVAL`, the expression `power 4 x` is marked for partial evaluation, such that the function `power` will be improved w.r.t. the arguments 4 and x. Since the first argument is known in this case, the partial evalautor is able to remove the case distinctions in the implementation of `power`, and we invoke it via

```
$ curry peval Examples/power4.curry
Curry Partial Evaluator
Version 0.1 of 12/09/2016
CAU Kiel

Annotated Expressions
---------------------
power4.power 4 v1

Final Partial Evaluation
------------------------
power4._pe0 :: Prelude.Int  →  Prelude.Int
power4._pe0 v1 = let { v2 = v1 * v1 } in v2 * v2

Writing specialized program into file 'Examples/.curry/power4_pe.fcy'.
```

Note that the partial evaluator successfully removed the case distinction, such that the operation `power4` can be expected to run reasonably faster. The new auxiliary function `power4._pe0` is integrated into the existing module such that only the implementation of `power4` is changed, which becomes visible if we increase the level of verbosity:

```
$ curry peval -v2 Examples/power4.curry
Curry Partial Evaluator
Version 0.1 of 12/09/2016
CAU Kiel

Annotated Expressions
---------------------
power4.power 4 v1


... (skipped output)


Resulting program
-----------------
module power4 ( power4.square, power4.even, power4.power, power4.power4 ) where


import Prelude


power4.square :: Prelude.Int  → Prelude.Int
power4.square v1 = v1 * v1


power4.even :: Prelude.Int  → Prelude.Bool
power4.even v1 = (Prelude.mod v1 2) == 0


power4.power :: Prelude.Int  → Prelude.Int  → Prelude.Int
power4.power v1 v2 = case (v1 <= 0) of
    Prelude.True  → 1
    Prelude.False  → case (power4.even v1) of
        Prelude.True  → power4.power (Prelude.div v1 2) (power4.square v2)
        Prelude.False  → v2 * (power4.power (v1 - 1) v2)


power4.power4 :: Prelude.Int  → Prelude.Int
power4.power4 v1 = power4._pe0 v1


power4._pe0 :: Prelude.Int  → Prelude.Int
power4._pe0 v1 = let { v2 = v1 * v1 } in v2 * v2
```

17.2 Options

The partial evaluator can be parametrized using a number of options, which can also be shown using `--help`.

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

-V, `--version` These options trigger the output of the version information of the partial evaluator.

-d, --debug This flag is intended for development and testing issues only, and necessary to print the resulting program to the standard output stream even if the verbosity is set to zero.

--assert, --closed These flags enable some internal assertions which are reasonable during development of the partial evaluator.

--no-funpats Normally, functions defined using functional patterns are automatically considered for partial evaluation, since their annotation using PEVAL is a little bit cumbersome. However, this automatic consideration can be disabled using this flag.

-v n, --verbosity=n Set the verbosity level to n, see above for the explanation of the different levels.

--color=mode, --colour=mode Set the coloring mode to mode, see above for the explanation of the different modes.

-S semantics, --semantics=semantics Allows the use to choose a semantics used during partial evaluation. Note that only the natural semantics can be considered correct for non-confluent programs, which is why it is the default semantics [32]. However, the rlnt calculus can also be chosen which is based on term rewriting, thus implementing a run-time choice semantics [1]. The letrw semantics is currently not fully supported, but implements the gist of let-rewriting [30].

-A mode, --abstract=mode During partial evaluation, all expressions that may potentially occur in the evaluation of an annotated expression are considered and evaluated, in order to ensure that all these expressions are also defined in the resulting program. Unfortunately, this imposes the risk of non-termination, which is why similar expressions are generalized according to the abstraction criterion. While the none criterion avoids generalizations and thus may lead to non-termination of the partial evaluator, the criteria wqo and wfo both ensure termination. In general, the criterion wqo seems to be a good compromise of ensured termination and the quality of the computed result program.

-P mode, --proceed=mode While the abstraction mode is responsible to limit the number of different expressions to be considered, the proceed mode limits the number of function calls to be evaluated during the evaluation of a *single* expressions. While the mode one only allows a single function call to be evaluated, the mode each allows a single call of each single function, while all puts no restrictions on the number of function calls to be evaluated. Clearly, the last alternative also imposes a risk of non-termination.

--suffix=SUFFIX Set the suffix appended to the file name to compute the output file. If the suffix is set to the empty string, then the original FlatCurry file will be replaced.

18   Technical Problems

One can implement distributed systems with KiCS2 by the use of the library `NamedSocket` (Section A.2.27) 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 libraries `HTML` (Section A.5.3) or `WUI` (Section A.5.9). 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 8767: 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 `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 | fgrep 8767`" (or similar).

The CPNS is implemented as a demon listening on its port 8767 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 scripts *kics2home*/currytools/cpns/start and *kics2home*/currytools/cpns/stop. If the demon is already running, the command *kics2home*/currytools/cpns/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

<div align="center">

`kics2@curry-language.org`

</div>

References

[1] E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[2] S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.

[3] 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.

[4] 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.

[5] S. Antoy and M. Hanus. Curry without Success. In *Proc. of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*, volume 1335 of *CEUR Workshop Proceedings*, pages 140–154. CEUR-WS.org, 2014.

[6] 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.

[7] S. Antoy and M. Hanus. Default rules for Curry. In *Proc. of the 18th International Symposium on Practical Aspects of Declarative Languages (PADL 2016)*, pages 65–82. Springer LNCS 9585, 2016.

[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), to appear in EPTCS*, 2016.

[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 *Pre-Proc. of the 24th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2016)*. HTWK Leipzig, 2016.

[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 (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-language.org, 2016.

[28] T. Johnsson. Lambda lifting: Transforming programs to recursive functions. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer LNCS 201, 1985.

[29] J. Krone. Integration of SQL into Curry. Master's thesis, University of Kiel, 2015.

[30] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '07, pages 197–208, New York, NY, USA, 2007. ACM.

[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] Björn Peemöller. *Normalization and Partial Evaluation of Functional Logic Programs*. Department of Computer Science, Kiel University, 2016. Dissertation, Faculty of Engineering, Kiel University.

[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 an extensive collection of libraries for application programming. The libraries for meta-programming by representing Curry programs as datatypes in Curry are described in the following subsection in more detail. The complete set of libraries with all exported types and functions are described in the further subsections. For a more detailed online documentation of all libraries of KiCS2, see http://www-ps.informatik.uni-kiel.de/kics2/lib/index.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 system modules `FlatCurry.Types` (Section A.6.6) and `AbstractCurry.Types` (Section A.6.1) which define datatypes for the representation of Curry programs. `AbstractCurry.Types` is a more direct representation of a Curry program, whereas `FlatCurry.Types` 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[8] of these modules.

As an example, consider a program file "`test.curry`" containing the following two lines:

```
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
        (FuncType (TCons ("Prelude","[]") [(TVar 0)])
                  (TCons ("Prelude","[]") [(TVar 0)]))
        (Rule [0]
           (Case Flex (Var 1)
              [Branch (Pattern ("Prelude","[]") [])
                  (Comb ConsCall ("Prelude","[]") []),
               Branch (Pattern ("Prelude",":") [2,3])
                  (Comb FuncCall ("Prelude","++")
                        [Comb FuncCall ("test","rev") [Var 3],
```

---

[8]http://www-ps.informatik.uni-kiel.de/kics2/lib/FlatCurry.Types.html and http://www-ps.informatik.uni-kiel.de/kics2/lib/AbstractCurry.Types.html

```
                          Comb ConsCall ("Prelude",":")
                                [Var 2,Comb ConsCall ("Prelude","[]") []]
                          ])
                 ]))]
   []
 )
```

A.2 General Libraries

A.2.1 Library AllSolutions

This module contains a collection of functions for obtaining lists of solutions to constraints. These operations are useful to encapsulate non-deterministic operations between I/O actions in order to connect the worlds of logic and functional programming and to avoid non-determinism failures on the I/O level.

In contrast the "old" concept of encapsulated search (which could be applied to any subexpression in a computation), the operations to encapsulate search in this module are I/O actions in order to avoid some anomalies in the old concept.

Exported functions:

`getAllValues :: a → IO [a]`

> Gets all values of an expression (currently, via an incomplete depth-first strategy). Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables.

`getOneValue :: a → IO (Maybe a)`

> Gets one value of an expression (currently, via an incomplete left-to-right strategy). Returns Nothing if the search space is finitely failed.

`getAllSolutions :: (a → Bool) → IO [a]`

> Gets all solutions to a constraint (currently, via an incomplete depth-first left-to-right strategy). Conceptually, all solutions are computed on a copy of the constraint, i.e., the evaluation of the constraint does not share any results. Moreover, this evaluation suspends if the constraints contain unbound variables. Similar to Prolog's findall.

`getOneSolution :: (a → Bool) → IO (Maybe a)`

> Gets one solution to a constraint (currently, via an incomplete left-to-right strategy). Returns Nothing if the search space is finitely failed.

`getAllFailures :: a → (a → Bool) → IO [a]`

> Returns a list of values that do not satisfy a given constraint.

A.2.2 Library Assertion

This module defines the datatype and operations for the Curry module tester "currytest".

Exported types:

`data Assertion`

> Datatype for defining test cases.

Exported constructors:

`data ProtocolMsg`

> The messages sent to the test GUI. Used by the currytest tool.

Exported constructors:

- `TestModule :: String → ProtocolMsg`

- `TestCase :: String → Bool → ProtocolMsg`

- `TestFinished :: ProtocolMsg`

- `TestCompileError :: ProtocolMsg`

Exported functions:

`assertTrue :: String → Bool → Assertion ()`

> `(assertTrue s b)` asserts (with name `s`) that `b` must be true.

`assertEqual :: String → a → a → Assertion a`

> `(assertEqual s e1 e2)` asserts (with name s) that `e1` and `e2` must be equal (w.r.t. `==`).

`assertValues :: String → a → [a] → Assertion a`

> `(assertValues s e vs)` asserts (with name s) that `vs` is the multiset of all values of `e`. All values of `e` are compared with the elements in `vs` w.r.t. `==`.

`assertSolutions :: String → (a → Bool) → [a] → Assertion a`

> `(assertSolutions s c vs)` asserts (with name s) that constraint abstraction `c` has the multiset of solutions `vs`. The solutions of `c` are compared with the elements in `vs` w.r.t. `==`.

`assertIO :: String → IO a → a → Assertion a`

> `(assertIO s a r)` asserts (with name s) that I/O action `a` yields the result value `r`.

`assertEqualIO :: String → IO a → IO a → Assertion a`

(`assertEqualIO s a1 a2`) asserts (with name **s**) that I/O actions **a1** and **a2** yield equal (w.r.t. `==`) results.

`seqStrActions :: IO (String,Bool) → IO (String,Bool) → IO (String,Bool)`

Combines two actions and combines their results. Used by the currytest tool.

`checkAssertion :: String → ((String,Bool) → IO (String,Bool)) → Assertion a → IO (String,Bool)`

Executes and checks an assertion, and process the result by an I/O action. Used by the currytest tool.

`writeAssertResult :: (String,Bool) → IO Int`

Prints the results of assertion checking. If failures occurred, the return code is positive. Used by the currytest tool.

`showTestMod :: Int → String → IO ()`

Sends message to GUI for showing test of a module. Used by the currytest tool.

`showTestCase :: Int → (String,Bool) → IO (String,Bool)`

Sends message to GUI for showing result of executing a test case. Used by the currytest tool.

`showTestEnd :: Int → IO ()`

Sends message to GUI for showing end of module test. Used by the currytest tool.

`showTestCompileError :: Int → IO ()`

Sends message to GUI for showing compilation errors in a module test. Used by the currytest tool.

A.2.3 Library 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.

`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.

`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.2.4 Library Combinatorial

A collection of common non-deterministic and/or combinatorial operations. Many operations are intended to operate on sets. The representation of these sets is not hidden; rather sets are represented as lists. Ideally these lists contains no duplicate elements and the order of their elements cannot be observed. In practice, these conditions are not enforced.

Exported functions:

`permute :: [a]` $\rightarrow$ `[a]`

> Compute any permutation of a list.

`subset :: [a]` $\rightarrow$ `[a]`

> Compute any sublist of a list. The sublist contains some of the elements of the list in the same order.

`allSubsets :: [a]` $\rightarrow$ `[[a]]`

> Compute all the sublists of a list.

`splitSet :: [a]` $\rightarrow$ `([a],[a])`

> Split a list into any two sublists.

`sizedSubset :: Int` $\rightarrow$ `[a]` $\rightarrow$ `[a]`

> Compute any sublist of fixed length of a list. Similar to `subset`, but the length of the result is fixed.

`partition :: [a]` $\rightarrow$ `[[a]]`

> Compute any partition of a list. The output is a list of non-empty lists such that their concatenation is a permutation of the input list. No guarantee is made on the order of the arguments in the output.

A.2.5 Library CPNS

Implementation of a Curry Port Name Server based on raw sockets. It is used to implement the library Ports for distributed programming with ports.

Exported functions:

`cpnsStart :: IO ()`

> Starts the "Curry Port Name Server" (CPNS) running on the local machine. The CPNS is responsible to resolve symbolic names for ports into physical socket numbers so that a port can be reached under its symbolic name from any machine in the world.

`cpnsShow :: IO ()`

Shows all registered ports at the local CPNS demon (in its logfile).

`cpnsStop :: IO ()`

Terminates the local CPNS demon

`registerPort :: String → Int → Int → IO ()`

Registers a symbolic port at the local host.

`getPortInfo :: String → String → IO (Int,Int)`

Gets the information about a symbolic port at some host.

`unregisterPort :: String → IO ()`

Unregisters a symbolic port at the local host.

`cpnsAlive :: Int → String → IO Bool`

Tests whether the CPNS demon at a host is alive.

`main :: IO ()`

Main function for CPNS demon. Check arguments and execute command.

A.2.6 Library CSV

Library for reading/writing files in CSV format. Files in CSV (comma separated values) format can be imported and exported by most spreadsheed and database applications.

Exported functions:

`writeCSVFile :: String → [[String]] → IO ()`

Writes a list of records (where each record is a list of strings) into a file in CSV format.

`showCSV :: [[String]] → String`

Shows a list of records (where each record is a list of strings) as a string in CSV format.

`readCSVFile :: String → IO [[String]]`

Reads a file in CSV format and returns the list of records (where each record is a list of strings).

`readCSVFileWithDelims :: String → String → IO [[String]]`

Reads a file in CSV format and returns the list of records (where each record is a list of strings).

`readCSV :: String → [[String]]`

Reads a string in CSV format and returns the list of records (where each record is a list of strings).

`readCSVWithDelims :: String → String → [[String]]`

Reads a string in CSV format and returns the list of records (where each record is a list of strings).

A.2.7 Library Debug

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 :: a → b → b`

> Prints the first argument using `show` and returns the second argument afterwards.

`traceShowId :: 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.2.8 Library Directory

Library for accessing the directory structure of the underlying operating system.

Exported functions:

`doesFileExist :: String → IO Bool`

> Returns true if the argument is the name of an existing file.

`doesDirectoryExist :: String → IO Bool`

> Returns true if the argument is the name of an existing directory.

`fileSize :: String → IO Int`

> Returns the size of the file.

`getModificationTime :: String → IO ClockTime`

> Returns the modification time of the file.

`getCurrentDirectory :: IO String`

> Returns the current working directory.

`setCurrentDirectory :: String → IO ()`

> Sets the current working directory.

`getDirectoryContents :: String → IO [String]`

> Returns the list of all entries in a directory.

`createDirectory :: String → IO ()`

> Creates a new directory with the given name.

`createDirectoryIfMissing :: Bool → String → IO ()`

> Creates a new directory with the given name if it does not already exist. If the first parameter is `True` it will also create all missing parent directories.

`removeDirectory :: String → IO ()`

> Deletes a directory from the file system.

`renameDirectory :: String → String → IO ()`

> Renames a directory.

`getHomeDirectory :: IO String`

> Returns the home directory of the current user.

`getTemporaryDirectory :: IO String`

> Returns the temporary directory of the operating system.

`getAbsolutePath :: String → IO String`

> Convert a path name into an absolute one. For instance, a leading ~ is replaced by the current home directory.

`removeFile :: String → IO ()`

> Deletes a file from the file system.

`renameFile :: String → String → IO ()`

> Renames a file.

`copyFile :: String → String → IO ()`

> Copy the contents from one file to another file

A.2.9 Library Distribution

This module contains functions to obtain information concerning the current distribution of the Curry implementation, e.g., compiler version, load paths, front end.

Exported types:

`data FrontendTarget`

> Data type for representing the different target files that can be produced by the front end of the Curry compiler.

Exported constructors:

- `FCY :: FrontendTarget`

  `FCY`

  - FlatCurry file ending with .fcy

- `FINT :: FrontendTarget`

  `FINT`

  - FlatCurry interface file ending with .fint

- `ACY :: FrontendTarget`

  `ACY`

  - AbstractCurry file ending with .acy

- `UACY :: FrontendTarget`

  `UACY`

  - Untyped (without type checking) AbstractCurry file ending with .uacy

- `HTML :: FrontendTarget`

  `HTML`

  - colored HTML representation of source program

- `CY :: FrontendTarget`

  `CY`

  - source representation employed by the frontend

- `TOKS :: FrontendTarget`

  `TOKS`

  - token stream of source program

```
data FrontendParams
```

Abstract data type for representing parameters supported by the front end of the Curry compiler.

Exported constructors:

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.

```
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.

```
installDir :: String
```

Path of the main installation directory of the Curry compiler.

```
rcFileName :: IO String
```

The name of the file specifying configuration parameters of the current distribution. This file must have the usual format of property files (see description in module PropertyFile).

```
rcFileContents :: IO [(String,String)]
```

Returns the current configuration parameters of the distribution. This action yields the list of pairs (var,val).

```
getRcVar :: String → IO (Maybe String)
```

Look up a specific configuration variable as specified by user in his rc file. Uppercase/lowercase is ignored for the variable names.

```
getRcVars :: [String] → IO [Maybe String]
```

Look up configuration variables as specified by user in his rc file. Uppercase/lowercase is ignored for the variable names.

splitModuleFileName :: String → String → (String,String)

Split the `FilePath` of a module into the directory prefix and the `FilePath` corresponding to the module name. For instance, the call **splitModuleFileName "Data.Set" "lib/Data/Set.curry"** evaluates to **("lib", "Data/Set.curry")**. This can be useful to compute output directories while retaining the hierarchical module structure.

splitModuleIdentifiers :: String → [String]

Split up the components of a module identifier. For instance, **splitModuleIdentifiers "Data.Set"** evaluates to **["Data", "Set"]**.

joinModuleIdentifiers :: [String] → String

Join the components of a module identifier. For instance, **joinModuleIdentifiers ["Data", "Set"]** evaluates to **"Data.Set"**.

stripCurrySuffix :: String → String

Strips the suffix ".curry" or ".lcurry" from a file name.

modNameToPath :: String → String

Transforms a hierarchical module name into a path name, i.e., replace the dots in the name by directory separator chars.

currySubdir :: String

Name of the sub directory where auxiliary files (.fint, .fcy, etc) are stored.

inCurrySubdir :: String → String

Transforms a path to a module name into a file name by adding the **currySubDir** to the path and transforming a hierarchical module name into a path. For instance, **inCurrySubdir "mylib/Data.Char"** evaluates to **"mylib/.curry/Data/Char"**.

inCurrySubdirModule :: String → String → String

Transforms a file name by adding the currySubDir to the file name. This version respects hierarchical module names.

addCurrySubdir :: String → String

Transforms a directory name into the name of the corresponding sub directory containing auxiliary files.

sysLibPath :: [String]

finding files in correspondence to compiler load path Returns the current path (list of directory names) of the system libraries.

`getLoadPathForModule :: String` $\rightarrow$ `IO [String]`

> Returns the current path (list of directory names) that is used for loading modules w.r.t. a given module path. The directory prefix of the module path (or "." if there is no such prefix) is the first element of the load path and the remaining elements are determined by the environment variable CURRYRPATH and the entry "libraries" of the system's rc file.

`lookupModuleSourceInLoadPath :: String` $\rightarrow$ `IO (Maybe (String,String))`

> Returns a directory name and the actual source file name for a module by looking up the module source in the current load path. If the module is hierarchical, the directory is the top directory of the hierarchy. Returns Nothing if there is no corresponding source file.

`lookupModuleSource :: [String]` $\rightarrow$ `String` $\rightarrow$ `IO (Maybe (String,String))`

> Returns a directory name and the actual source file name for a module by looking up the module source in the load path provided as the first argument. If the module is hierarchical, the directory is the top directory of the hierarchy. Returns Nothing if there is no corresponding source file.

`defaultParams :: FrontendParams`

> The default parameters of the front end.

`rcParams :: IO FrontendParams`

> The default parameters of the front end as configured by the compiler specific resource configuration file.

`setQuiet :: Bool` $\rightarrow$ `FrontendParams` $\rightarrow$ `FrontendParams`

> Set quiet mode of the front end.

`setExtended :: Bool` $\rightarrow$ `FrontendParams` $\rightarrow$ `FrontendParams`

> Set extended mode of the front end.

`setOverlapWarn :: Bool` $\rightarrow$ `FrontendParams` $\rightarrow$ `FrontendParams`

> Set overlap warn mode of the front end.

`setFullPath :: [String]` $\rightarrow$ `FrontendParams` $\rightarrow$ `FrontendParams`

> Set the full path of the front end. If this parameter is set, the front end searches all modules in this path (instead of using the default path).

`setHtmlDir :: String` $\rightarrow$ `FrontendParams` $\rightarrow$ `FrontendParams`

> Set the htmldir parameter of the front end. Relevant for HTML generation.

`setLogfile :: String` $\rightarrow$ `FrontendParams` $\rightarrow$ `FrontendParams`

Set the logfile parameter of the front end. If this parameter is set, all messages produced by the front end are stored in this file.

`setSpecials :: String → FrontendParams → FrontendParams`

Set additional specials parameters of the front end. These parameters are specific for the current front end and should be used with care, since their form might change in the future.

`addTarget :: FrontendTarget → FrontendParams → FrontendParams`

Add an additional front end target.

`quiet :: FrontendParams → Bool`

Returns the value of the "quiet" parameter.

`extended :: FrontendParams → Bool`

Returns the value of the "extended" parameter.

`overlapWarn :: FrontendParams → Bool`

Returns the value of the "overlapWarn" parameter.

`fullPath :: FrontendParams → Maybe [String]`

Returns the full path parameter of the front end.

`htmldir :: FrontendParams → Maybe String`

Returns the htmldir parameter of the front end.

`logfile :: FrontendParams → Maybe String`

Returns the logfile parameter of the front end.

`specials :: FrontendParams → String`

Returns the special parameters of the front end.

`callFrontend :: FrontendTarget → String → IO ()`

In order to make sure that compiler generated files (like .fcy, .fint, .acy) are up to date, one can call the front end of the Curry compiler with this action. If the front end returns with an error, an exception is raised.

`callFrontendWithParams :: FrontendTarget → FrontendParams → String → IO ()`

In order to make sure that compiler generated files (like .fcy, .fint, .acy) are up to date, one can call the front end of the Curry compiler with this action where various parameters can be set. If the front end returns with an error, an exception is raised.

A.2.10 Library Either

Library with some useful operations for the `Either` data type.

Exported functions:

`lefts :: [Either a b] → [a]`

> 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 a b → a`

> Extract the value from a `Left` constructor.

`fromRight :: Either a b → b`

> Extract the value from a `Right` constructor.

`partitionEithers :: [Either a b] → ([a],[b])`

> 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.2.11 Library ErrorState

A combination of Error and state monad like `ErrorT State` in Haskell.

Exported types:

`type ES a b c = b → Either a (c,b)`

> Error state monad.

Exported functions:

evalES :: (a → Either b (c,a)) → a → Either b c

    Evaluate an ES monad

returnES :: a → b → Either c (a,b)

    Lift a value into the ES monad

failES :: a → b → Either a (c,b)

    Failing computation in the ES monad

(>+=) :: (a → Either b (c,a)) → (c → a → Either b (d,a)) → a → Either b (d,a)

    Bind of the ES monad

(>+) :: (a → Either b (c,a)) → (a → Either b (d,a)) → a → Either b (d,a)

    Sequence operator of the ES monad

(<$>) :: (a → b) → (c → Either d (a,c)) → c → Either d (b,c)

    Apply a pure function onto a monadic value.

(<*>) :: (a → Either b (c → d,a)) → (a → Either b (c,a)) → a → Either b (d,a)

    Apply a function yielded by a monadic action to a monadic value.

gets :: a → Either b (a,a)

    Retrieve the current state

puts :: a → a → Either b ((),a)

    Replace the current state

modify :: (a → a) → a → Either b ((),a)

    Modify the current state

mapES :: (a → b → Either c (d,b)) → [a] → b → Either c ([d],b)

    Map a monadic function on all elements of a list by sequencing the effects.

concatMapES :: (a → b → Either c ([d],b)) → [a] → b → Either c ([d],b)

    Same as concatMap, but for a monadic function.

mapAccumES :: (a → b → c → Either d ((a,e),c)) → a → [b] → c → Either d ((a,[e]),c)

    Same as mapES but with an additional accumulator threaded through.

A.2.12 Library FileGoodies

A collection of useful operations when dealing with files.

Exported functions:

`separatorChar :: Char`

>   The character for separating hierarchies in file names. On UNIX systems the value is `/`.

`pathSeparatorChar :: Char`

>   The character for separating names in path expressions. On UNIX systems the value is `:`.

`suffixSeparatorChar :: Char`

>   The character for separating suffixes in file names. On UNIX systems the value is `.`.

`isAbsolute :: String → Bool`

>   Is the argument an absolute name?

`dirName :: String → String`

>   Extracts the directoy prefix of a given (Unix) file name. Returns `"."` if there is no prefix.

`baseName :: String → String`

>   Extracts the base name without directoy prefix of a given (Unix) file name.

`splitDirectoryBaseName :: String → (String,String)`

>   Splits a (Unix) file name into the directory prefix and the base name. The directory prefix is `"."` if there is no real prefix in the name.

`stripSuffix :: String → String`

>   Strips a suffix (the last suffix starting with a dot) from a file name.

`fileSuffix :: String → String`

>   Yields the suffix (the last suffix starting with a dot) from given file name.

`splitBaseName :: String → (String,String)`

>   Splits a file name into prefix and suffix (the last suffix starting with a dot and the rest).

`splitPath :: String → [String]`

>   Splits a path string into list of directory names.

`lookupFileInPath :: String → [String] → [String] → IO (Maybe String)`

>   Looks up the first file with a possible suffix in a list of directories. Returns Nothing if such a file does not exist.

`getFileInPath :: String → [String] → [String] → IO String`

>   Gets the first file with a possible suffix in a list of directories. An error message is delivered if there is no such file.

A.2.13 Library FilePath

This library is a direct port of the Haskell library System.FilePath of Neil Mitchell.

Exported types:

```
type FilePath = String
```

Exported functions:

```
pathSeparator :: Char
```

```
pathSeparators :: String
```

```
isPathSeparator :: Char → Bool
```

```
searchPathSeparator :: Char
```

```
isSearchPathSeparator :: Char → Bool
```

```
extSeparator :: Char
```

```
isExtSeparator :: Char → Bool
```

```
splitSearchPath :: String → [String]
```

```
getSearchPath :: IO [String]
```

```
splitExtension :: String → (String,String)
```

```
takeExtension :: String → String
```

```
replaceExtension :: String → String → String


(<.>) :: String → String → String


dropExtension :: String → String


addExtension :: String → String → String


hasExtension :: String → Bool


splitExtensions :: String → (String,String)


dropExtensions :: String → String


takeExtensions :: String → String


splitDrive :: String → (String,String)


joinDrive :: String → String → String


takeDrive :: String → String


dropDrive :: String → String


hasDrive :: String → Bool


isDrive :: String → Bool
```

splitFileName :: String → (String,String)

replaceFileName :: String → String → String

dropFileName :: String → String

takeFileName :: String → String

takeBaseName :: String → String

replaceBaseName :: String → String → String

hasTrailingPathSeparator :: String → Bool

addTrailingPathSeparator :: String → String

dropTrailingPathSeparator :: String → String

takeDirectory :: String → String

replaceDirectory :: String → String → String

combine :: String → String → String

(</>) :: String → String → String

splitPath :: String → [String]

```
splitDirectories :: String → [String]


joinPath :: [String] → String


equalFilePath :: String → String → Bool


makeRelative :: String → String → String


normalise :: String → String


isValid :: String → Bool


makeValid :: String → String


isRelative :: String → Bool


isAbsolute :: String → Bool
```

A.2.14 Library Findall

Library with some operations for encapsulating search. Note that some of these operations are not fully declarative, i.e., the results depend on the order of evaluation and program rules. There are newer and better approaches the encpasulate search, in particular, set functions (see module `SetFunctions`), which should be used.
This library is included only for compatibility with PAKCS.

Exported functions:

```
getAllValues :: a → IO [a]
```

> Gets all values of an expression (currently, via an incomplete depth-first strategy). Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables. Similar to Prolog's findall.

```
getSomeValue :: a → IO a
```

Gets a value of an expression (currently, via an incomplete depth-first strategy). The expression must have a value, otherwise the computation fails. Conceptually, the value is computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables.

```
allValues :: a → [a]
```

Returns all values of an expression (currently, via an incomplete depth-first strategy). Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

Note that this operation is not purely declarative since the ordering of the computed values depends on the ordering of the program rules.

```
someValue :: a → a
```

Returns some value for an expression (currently, via an incomplete depth-first strategy). If the expression has no value, the computation fails. Conceptually, the value is computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value.

```
allSolutions :: (a → Bool) → [a]
```

Returns all values satisfying a predicate, i.e., all arguments such that the predicate applied to the argument can be evaluated to `True` (currently, via an incomplete depth-first strategy).

Note that this operation is not purely declarative since the ordering of the computed values depends on the ordering of the program rules.

```
someSolution :: (a → Bool) → a
```

Returns some values satisfying a predicate, i.e., some argument such that the predicate applied to the argument can be evaluated to `True` (currently, via an incomplete depth-first strategy). If there is no value satisfying the predicate, the computation fails.

Note that this operation is not purely declarative since the ordering of the computed values depends on the ordering of the program rules. Thus, this operation should be used only if the predicate has a single solution.

A.2.15   Library Float

A collection of operations on floating point numbers.

Exported functions:

`pi :: Float`

> The number pi.

`(+.) :: Float → Float → Float`

> Addition on floats.

`(-.) :: Float → Float → Float`

> Subtraction on floats.

`(*.) :: Float → Float → Float`

> Multiplication on floats.

`(/.) :: Float → Float → Float`

> Division on floats.

`(^.) :: Float → Int → Float`

> The value of `a ^. b` is `a` raised to the power of `b`. Executes in `O(log b)` steps.

`i2f :: Int → Float`

> Conversion function from integers to floats.

`truncate :: Float → Int`

> Conversion function from floats to integers. The result is the closest integer between the argument and 0.

`round :: Float → Int`

> Conversion function from floats to integers. The result is the nearest integer to the argument. If the argument is equidistant between two integers, it is rounded to the closest even integer value.

`recip :: Float → Float`

> Reciprocal

`sqrt :: Float → Float`

> Square root.

`log :: Float → Float`

> Natural logarithm.

`logBase :: Float → Float → Float`

Logarithm to arbitrary Base.

`exp :: Float → Float`

Natural exponent.

`sin :: Float → Float`

Sine.

`cos :: Float → Float`

Cosine.

`tan :: Float → Float`

Tangent.

`asin :: Float → Float`

Arc sine.

`acos :: Float → Float`

`atan :: Float → Float`

Arc tangent.

`sinh :: Float → Float`

Hyperbolic sine.

`cosh :: Float → Float`

`tanh :: Float → Float`

Hyperbolic tangent.

`asinh :: Float → Float`

Hyperbolic Arc sine.

`acosh :: Float → Float`

`atanh :: Float → Float`

Hyperbolic Arc tangent.

A.2.16   Library 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 = \x y -> f x * f y`. Typical usage: `sortBy (compare 'on' fst)`.

`first :: (a → b) → (a,c) → (b,c)`

> Apply a function to the first component of a tuple.

`second :: (a → b) → (c,a) → (c,b)`

> Apply a function to the second component of a tuple.

`(***) :: (a → b) → (c → d) → (a,c) → (b,d)`

> Apply two functions to the two components of a tuple.

`(&&&) :: (a → b) → (a → c) → a → (b,c)`

> Apply two functions to a value and returns a tuple of the results.

`both :: (a → b) → (a,a) → (b,b)`

> Apply a function to both components of a tuple.

A.2.17 Library FunctionInversion

This module provides some utility functions for inverting functions.

Exported functions:

`invf1 :: (a → b) → b → a`

> Inverts a unary function.

`invf2 :: (a → b → c) → c → (a,b)`

> Inverts a binary function.

`invf3 :: (a → b → c → d) → d → (a,b,c)`

> Inverts a ternary function.

`invf4 :: (a → b → c → d → e) → e → (a,b,c,d)`

> Inverts a function of arity 4.

`invf5 :: (a → b → c → d → e → f) → f → (a,b,c,d,e)`

> Inverts a function of arity 5.

A.2.18 Library 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.2.19 Library Global

Library for handling global entities. A global entity has a name declared in the program. Its value can be accessed and modified by IO actions. Furthermore, global entities can be declared as persistent so that their values are stored across different program executions.
Currently, it is still experimental so that its interface might be slightly changed in the future.
A global entity g with an initial value v of type t must be declared by:

g :: Global t
g = global v spec

Here, the type t must not contain type variables and spec specifies the storage mechanism for the global entity (see type GlobalSpec).

Exported types:

data Global

> The abstract type of a global entity.

Exported constructors:

data GlobalSpec

> The storage mechanism for the global entity.

Exported constructors:

- Temporary :: GlobalSpec

  Temporary

    - the global value exists only during a single execution of a program

- Persistent :: String → GlobalSpec

  Persistent f

    - the global value is stored persisently in file f (which is created and initialized if it does not exists)

Exported functions:

global :: a → GlobalSpec → Global a

> global is only used for the declaration of a global value and should not be used elsewhere. In the future, it might become a keyword.

readGlobal :: Global a → IO a

> Reads the current value of a global.

safeReadGlobal :: Global a → a → IO a

> Safely reads the current value of a global. If readGlobal fails (e.g., due to a corrupted persistent storage), the global is re-initialized with the default value given as the second argument.

writeGlobal :: Global a → a → IO ()

> Updates the value of a global. The value is evaluated to a ground constructor term before it is updated.

A.2.20   Library GUI

This library contains definitions and functions to implement graphical user interfaces for Curry programs. It is based on Tcl/Tk and its basic ideas are described in detail in this paper

Exported types:

data GuiPort

> The port to a GUI is just the stream connection to a GUI where Tcl/Tk communication is done.

Exported constructors:

data Widget

> The type of possible widgets in a GUI.

Exported constructors:

- PlainButton :: [ConfItem] → Widget

  PlainButton

  - a button in a GUI whose event handler is activated if the user presses the button

- Canvas :: [ConfItem] → Widget

  Canvas

  - a canvas to draw pictures containing CanvasItems

- CheckButton :: [ConfItem] → Widget

  CheckButton

  - a check button: it has value "0" if it is unchecked and value "1" if it is checked

- Entry :: [ConfItem] → Widget

  Entry

  - an entry widget for entering single lines

- Label :: [ConfItem] → Widget

  Label

  - a label for showing a text

- ListBox :: [ConfItem] → Widget

  ListBox

  - a widget containing a list of items for selection

- Message :: [ConfItem] → Widget

  Message

  - a message for showing simple string values

- MenuButton :: [ConfItem] → Widget

  MenuButton

  - a button with a pull-down menu

- Scale :: Int → Int → [ConfItem] → Widget

  Scale

  - a scale widget to input values by a slider

- ScrollH :: WidgetRef → [ConfItem] → Widget

  ScrollH

– a horizontal scroll bar

- ScrollV :: WidgetRef → [ConfItem] → Widget

  ScrollV

    – a vertical scroll bar

- TextEdit :: [ConfItem] → Widget

  TextEdit

    – a text editor widget to show and manipulate larger text paragraphs

- Row :: [ConfCollection] → [Widget] → Widget

  Row

    – a horizontal alignment of widgets

- Col :: [ConfCollection] → [Widget] → Widget

  Col

    – a vertical alignment of widgets

- Matrix :: [ConfCollection] → [[Widget]] → Widget

  Matrix

    – a 2-dimensional (matrix) alignment of widgets

data ConfItem

The data type for possible configurations of a widget.

Exported constructors:

- Active :: Bool → ConfItem

  Active

    – define the active state for buttons, entries, etc.

- Anchor :: String → ConfItem

  Anchor

    – alignment of information inside a widget where the argument must be: n, ne, e, se, s, sw, w, nw, or center

- Background :: String → ConfItem

  Background

    – the background color

- `Foreground :: String → ConfItem`

  `Foreground`

    – the foreground color

- `Handler :: Event → (GuiPort → IO [ReconfigureItem]) → ConfItem`

  `Handler`

    – an event handler associated to a widget. The event handler returns a list of widget ref/-configuration pairs that are applied after the handler in order to configure GUI widgets

- `Height :: Int → ConfItem`

  `Height`

    – the height of a widget (chars for text, pixels for graphics)

- `CheckInit :: String → ConfItem`

  `CheckInit`

    – initial value for checkbuttons

- `CanvasItems :: [CanvasItem] → ConfItem`

  `CanvasItems`

    – list of items contained in a canvas

- `List :: [String] → ConfItem`

  `List`

    – list of values shown in a listbox

- `Menu :: [MenuItem] → ConfItem`

  `Menu`

    – the items of a menu button

- `WRef :: WidgetRef → ConfItem`

  `WRef`

    – a reference to this widget

- `Text :: String → ConfItem`

  `Text`

    – an initial text contents

- `Width :: Int → ConfItem`

  `Width`

– the width of a widget (chars for text, pixels for graphics)

- `Fill :: ConfItem`

  `Fill`

    – fill widget in both directions

- `FillX :: ConfItem`

  `FillX`

    – fill widget in horizontal direction

- `FillY :: ConfItem`

  `FillY`

    – fill widget in vertical direction

- `TclOption :: String → ConfItem`

  `TclOption`

    – further options in Tcl syntax (unsafe!)

`data ReconfigureItem`

Data type for describing configurations that are applied to a widget or GUI by some event handler.

Exported constructors:

- `WidgetConf :: WidgetRef → ConfItem → ReconfigureItem`

  `WidgetConf wref conf`

    – reconfigure the widget referred by wref with configuration item conf

- `StreamHandler :: Handle → (Handle → GuiPort → IO [ReconfigureItem]) → ReconfigureItem`

  `StreamHandler hdl handler`

    – add a new handler to the GUI that processes inputs on an input stream referred by hdl

- `RemoveStreamHandler :: Handle → ReconfigureItem`

  `RemoveStreamHandler hdl`

    – remove a handler for an input stream referred by hdl from the GUI (usually used to remove handlers for closed streams)

`data Event`

The data type of possible events on which handlers can react. This list is still incomplete and might be extended or restructured in future releases of this library.

Exported constructors:

- `DefaultEvent :: Event`

  `DefaultEvent`

  – the default event of the widget

- `MouseButton1 :: Event`

  `MouseButton1`

  – left mouse button pressed

- `MouseButton2 :: Event`

  `MouseButton2`

  – middle mouse button pressed

- `MouseButton3 :: Event`

  `MouseButton3`

  – right mouse button pressed

- `KeyPress :: Event`

  `KeyPress`

  – any key is pressed

- `Return :: Event`

  `Return`

  – return key is pressed

`data ConfCollection`

The data type for possible configurations of widget collections (e.g., columns, rows).

Exported constructors:

- `CenterAlign :: ConfCollection`

  `CenterAlign`

  – centered alignment

- `LeftAlign :: ConfCollection`

  `LeftAlign`

– left alignment

- **RightAlign :: ConfCollection**

  RightAlign

  – right alignment

- **TopAlign :: ConfCollection**

  TopAlign

  – top alignment

- **BottomAlign :: ConfCollection**

  BottomAlign

  – bottom alignment

data MenuItem

The data type for specifying items in a menu.

Exported constructors:

- **MButton :: (GuiPort → IO [ReconfigureItem]) → String → MenuItem**

  MButton

  – a button with an associated command and a label string

- **MSeparator :: MenuItem**

  MSeparator

  – a separator between menu entries

- **MMenuButton :: String → [MenuItem] → MenuItem**

  MMenuButton

  – a submenu with a label string

data CanvasItem

The data type of items in a canvas. The last argument are further options in Tcl/Tk (for testing).

Exported constructors:

- **CLine :: [(Int,Int)] → String → CanvasItem**

- **CPolygon :: [(Int,Int)] → String → CanvasItem**

- **CRectangle :: (Int,Int) → (Int,Int) → String → CanvasItem**

114

- COval :: (Int,Int) $\rightarrow$ (Int,Int) $\rightarrow$ String $\rightarrow$ CanvasItem

- CText :: (Int,Int) $\rightarrow$ String $\rightarrow$ String $\rightarrow$ CanvasItem

data WidgetRef

The (hidden) data type of references to a widget in a GUI window. Note that the constructor WRefLabel will not be exported so that values can only be created inside this module.

Exported constructors:

data Style

The data type of possible text styles.

Exported constructors:

- Bold :: Style

  Bold

    – text in bold font

- Italic :: Style

  Italic

    – text in italic font

- Underline :: Style

  Underline

    – underline text

- Fg :: Color $\rightarrow$ Style

  Fg

    – foreground color, i.e., color of the text font

- Bg :: Color $\rightarrow$ Style

  Bg

    – background color of the text

data Color

The data type of possible colors.

Exported constructors:

- Black :: Color

- Blue :: Color

- Brown :: Color

- Cyan :: Color

- Gold :: Color

- Gray :: Color

- Green :: Color

- Magenta :: Color

- Navy :: Color

- Orange :: Color

- Pink :: Color

- Purple :: Color

- Red :: Color

- Tomato :: Color

- Turquoise :: Color

- Violet :: Color

- White :: Color

- Yellow :: Color

Exported functions:

row :: [Widget] → Widget

    Horizontal alignment of widgets.

col :: [Widget] → Widget

    Vertical alignment of widgets.

matrix :: [[Widget]] → Widget

    Matrix alignment of widgets.

debugTcl :: Widget → IO ()

    Prints the generated Tcl commands of a main widget (useful for debugging).

runPassiveGUI :: String → Widget → IO GuiPort

IO action to show a Widget in a new GUI window in passive mode, i.e., ignore all GUI events.

`runGUI :: String → Widget → IO ()`

IO action to run a Widget in a new window.

`runGUIwithParams :: String → String → Widget → IO ()`

IO action to run a Widget in a new window.

`runInitGUI :: String → Widget → (GuiPort → IO [ReconfigureItem]) → IO ()`

IO action to run a Widget in a new window. The GUI events are processed after executing an initial action on the GUI.

`runInitGUIwithParams :: String → String → Widget → (GuiPort → IO [ReconfigureItem]) → IO ()`

IO action to run a Widget in a new window. The GUI events are processed after executing an initial action on the GUI.

`runControlledGUI :: String → (Widget,String → GuiPort → IO ()) → Handle → IO ()`

Runs a Widget in a new GUI window and process GUI events. In addition, an event handler is provided that process messages received from an external stream identified by a handle (third argument). This operation is useful to run a GUI that should react on user events as well as messages written to the given handle.

`runConfigControlledGUI :: String → (Widget,String → GuiPort → IO [ReconfigureItem]) → Handle → IO ()`

Runs a Widget in a new GUI window and process GUI events. In addition, an event handler is provided that process messages received from an external stream identified by a handle (third argument). This operation is useful to run a GUI that should react on user events as well as messages written to the given handle.

`runInitControlledGUI :: String → (Widget,String → GuiPort → IO ()) → (GuiPort → IO [ReconfigureItem]) → Handle → IO ()`

Runs a Widget in a new GUI window and process GUI events after executing an initial action on the GUI window. In addition, an event handler is provided that process messages received from an external message stream. This operation is useful to run a GUI that should react on user events as well as messages written to the given handle.

`runHandlesControlledGUI :: String → (Widget,[Handle → GuiPort → IO [ReconfigureItem]]) → [Handle] → IO ()`

Runs a Widget in a new GUI window and process GUI events. In addition, a list of event handlers is provided that process inputs received from a corresponding list of handles to input streams. Thus, if the i-th handle has some data available, the i-th event handler is executed with the i-th handle as a parameter. This operation is useful to run a GUI that should react on inputs provided by other processes, e.g., via sockets.

runInitHandlesControlledGUI :: String → (Widget,[Handle → GuiPort → IO
[ReconfigureItem]]) → (GuiPort → IO [ReconfigureItem]) → [Handle] → IO ()

Runs a Widget in a new GUI window and process GUI events after executing an initial action on the GUI window. In addition, a list of event handlers is provided that process inputs received from a corresponding list of handles to input streams. Thus, if the i-th handle has some data available, the i-th event handler is executed with the i-th handle as a parameter. This operation is useful to run a GUI that should react on inputs provided by other processes, e.g., via sockets.

setConfig :: WidgetRef → ConfItem → GuiPort → IO ()

Changes the current configuration of a widget (deprecated operation, only included for backward compatibility). Warning: does not work for Command options!

exitGUI :: GuiPort → IO ()

An event handler for terminating the GUI.

getValue :: WidgetRef → GuiPort → IO String

Gets the (String) value of a variable in a GUI.

setValue :: WidgetRef → String → GuiPort → IO ()

Sets the (String) value of a variable in a GUI.

updateValue :: (String → String) → WidgetRef → GuiPort → IO ()

Updates the (String) value of a variable w.r.t. to an update function.

appendValue :: WidgetRef → String → GuiPort → IO ()

Appends a String value to the contents of a TextEdit widget and adjust the view to the end of the TextEdit widget.

appendStyledValue :: WidgetRef → String → [Style] → GuiPort → IO ()

Appends a String value with style tags to the contents of a TextEdit widget and adjust the view to the end of the TextEdit widget. Different styles can be combined, e.g., to get bold blue text on a red background. If Bold, Italic and Underline are combined, currently all but one of these are ignored. This is an experimental function and might be changed in the future.

addRegionStyle :: WidgetRef → (Int,Int) → (Int,Int) → Style → GuiPort → IO ()

Adds a style value in a region of a TextEdit widget. The region is specified a start and end position similarly to `getCursorPosition`. Different styles can be combined, e.g., to get bold blue text on a red background. If `Bold`, `Italic` and `Underline` are combined, currently all but one of these are ignored. This is an experimental function and might be changed in the future.

removeRegionStyle :: WidgetRef → (Int,Int) → (Int,Int) → Style → GuiPort → IO ()

Removes a style value in a region of a TextEdit widget. The region is specified a start and end position similarly to `getCursorPosition`. This is an experimental function and might be changed in the future.

getCursorPosition :: WidgetRef → GuiPort → IO (Int,Int)

Get the position (line,column) of the insertion cursor in a TextEdit widget. Lines are numbered from 1 and columns are numbered from 0.

seeText :: WidgetRef → (Int,Int) → GuiPort → IO ()

Adjust the view of a TextEdit widget so that the specified line/column character is visible. Lines are numbered from 1 and columns are numbered from 0.

focusInput :: WidgetRef → GuiPort → IO ()

Sets the input focus of this GUI to the widget referred by the first argument. This is useful for automatically selecting input entries in an application.

addCanvas :: WidgetRef → [CanvasItem] → GuiPort → IO ()

Adds a list of canvas items to a canvas referred by the first argument.

popupMessage :: String → IO ()

A simple popup message.

Cmd :: (GuiPort → IO ()) → ConfItem

A simple event handler that can be associated to a widget. The event handler takes a GUI port as parameter in order to read or write values from/into the GUI.

Command :: (GuiPort → IO [ReconfigureItem]) → ConfItem

An event handler that can be associated to a widget. The event handler takes a GUI port as parameter (in order to read or write values from/into the GUI) and returns a list of widget reference/configuration pairs which is applied after the handler in order to configure some GUI widgets.

Button :: (GuiPort → IO ()) → [ConfItem] → Widget

A button with an associated event handler which is activated if the button is pressed.

`ConfigButton :: (GuiPort → IO [ReconfigureItem]) → [ConfItem] → Widget`

A button with an associated event handler which is activated if the button is pressed. The event handler is a configuration handler (see Command) that allows the configuration of some widgets.

`TextEditScroll :: [ConfItem] → Widget`

A text edit widget with vertical and horizontal scrollbars. The argument contains the configuration options for the text edit widget.

`ListBoxScroll :: [ConfItem] → Widget`

A list box widget with vertical and horizontal scrollbars. The argument contains the configuration options for the list box widget.

`CanvasScroll :: [ConfItem] → Widget`

A canvas widget with vertical and horizontal scrollbars. The argument contains the configuration options for the text edit widget.

`EntryScroll :: [ConfItem] → Widget`

An entry widget with a horizontal scrollbar. The argument contains the configuration options for the entry widget.

`getOpenFile :: IO String`

Pops up a GUI for selecting an existing file. The file with its full path name will be returned (or `""` if the user cancels the selection).

`getOpenFileWithTypes :: [(String,String)] → IO String`

Pops up a GUI for selecting an existing file. The parameter is a list of pairs of file types that could be selected. A file type pair consists of a name and an extension for that file type. The file with its full path name will be returned (or `""` if the user cancels the selection).

`getSaveFile :: IO String`

Pops up a GUI for choosing a file to save some data. If the user chooses an existing file, she/he will asked to confirm to overwrite it. The file with its full path name will be returned (or `""` if the user cancels the selection).

`getSaveFileWithTypes :: [(String,String)] → IO String`

Pops up a GUI for choosing a file to save some data. The parameter is a list of pairs of file types that could be selected. A file type pair consists of a name and an extension for that file type. If the user chooses an existing file, she/he will asked to confirm to overwrite it. The file with its full path name will be returned (or `""` if the user cancels the selection).

`chooseColor :: IO String`

Pops up a GUI dialog box to select a color. The name of the color will be returned (or `""` if the user cancels the selection).

A.2.21 Library Integer

A collection of common operations on integer numbers. Most operations make no assumption on the precision of integers. Operation `bitNot` is necessarily an exception.

Exported functions:

`(^) :: Int → Int → Int`

> The value of `a ^ b` is `a` raised to the power of `b`. Fails if `b < 0`. Executes in `O(log b)` steps.

`pow :: Int → Int → Int`

> The value of `pow a b` is `a` raised to the power of `b`. Fails if `b < 0`. Executes in `O(log b)` steps.

`ilog :: Int → Int`

> The value of `ilog n` is the floor of the logarithm in the base 10 of `n`. Fails if `n <= 0`. For positive integers, the returned value is 1 less the number of digits in the decimal representation of `n`.

`isqrt :: Int → Int`

> The value of `isqrt n` is the floor of the square root of `n`. Fails if `n < 0`. Executes in `O(log n)` steps, but there must be a better way.

`factorial :: Int → Int`

> The value of `factorial n` is the factorial of `n`. Fails if `n < 0`.

`binomial :: Int → Int → Int`

> The value of `binomial n m` is n*(n-1)...(n-m+1)/m*(m-1)*...1 Fails if 'm <= 0' or 'n < m'.

`abs :: Int → Int`

> The value of `abs n` is the absolute value of `n`.

`max3 :: a → a → a → a`

> Returns the maximum of the three arguments.

`min3 :: a → a → a → a`

> Returns the minimum of the three arguments.

`maxlist :: [a] → a`

> Returns the maximum of a list of integer values. Fails if the list is empty.

`minlist :: [a] → a`

Returns the minimum of a list of integer values. Fails if the list is empty.

bitTrunc :: Int → Int → Int

The value of bitTrunc n m is the value of the n least significant bits of m.

bitAnd :: Int → Int → Int

Returns the bitwise AND of the two arguments.

bitOr :: Int → Int → Int

Returns the bitwise inclusive OR of the two arguments.

bitNot :: Int → Int

Returns the bitwise NOT of the argument. Since integers have unlimited precision, only the 32 least significant bits are computed.

bitXor :: Int → Int → Int

Returns the bitwise exclusive OR of the two arguments.

even :: Int → Bool

Returns whether an integer is even

odd :: Int → Bool

Returns whether an integer is odd

A.2.22   Library IO

Library for IO operations like reading and writing files that are not already contained in the prelude.

Exported types:

data Handle

The abstract type of a handle for a stream.

Exported constructors:

data IOMode

The modes for opening a file.

Exported constructors:

- ReadMode :: IOMode

- WriteMode :: IOMode

- AppendMode :: IOMode

122

```
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` $\rightarrow$ `IOMode` $\rightarrow$ `IO Handle`

Opens a file in specified mode and returns a handle to it.

`hClose :: Handle` $\rightarrow$ `IO ()`

Closes a file handle and flushes the buffer in case of output file.

`hFlush :: Handle` $\rightarrow$ `IO ()`

Flushes the buffer associated to handle in case of output file.

`hIsEOF :: Handle` $\rightarrow$ `IO Bool`

Is handle at end of file?

```
isEOF :: IO Bool
```

Is standard input at end of file?

`hSeek :: Handle` $\rightarrow$ `SeekMode` $\rightarrow$ `Int` $\rightarrow$ `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` $\rightarrow$ `Int` $\rightarrow$ `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 t milliseconds, it returns -1, otherwise it returns the index of the corresponding handle with the available data.

`hWaitForInputOrMsg :: Handle → [a] → IO (Either Handle [a])`

Waits until input is available on a given handles or a message in the message stream. Usually, the message stream comes from an external port. Thus, this operation implements a committed choice over receiving input from an IO handle or an external port.

Note that the implementation of this operation works only with Sicstus-Prolog 3.8.5 or higher (due to a bug in previous versions of Sicstus-Prolog).

`hWaitForInputsOrMsg :: [Handle] → [a] → IO (Either Int [a])`

Waits until input is available on some of the given handles or a message in the message stream. Usually, the message stream comes from an external port. Thus, this operation implements a committed choice over receiving input from IO handles or an external port.

Note that the implementation of this operation works only with Sicstus-Prolog 3.8.5 or higher (due to a bug in previous versions of Sicstus-Prolog).

`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 :: 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.2.23   Library IOExts

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:

`execCmd :: String → IO (Handle,Handle,Handle)`

>   Executes a command with a new default shell process. The standard I/O streams
>   of the new process (stdin,stdout,stderr) are returned as handles so that they can be
>   explicitly manipulated. They should be closed with `IO.hClose` since they are not closed
>   automatically when the process terminates.

`evalCmd :: String → [String] → String → IO (Int,String,String)`

Executes a command with the given arguments as a new default shell process and provides the input via the process' stdin input stream. The exit code of the process and the contents written to the standard I/O streams stdout and stderr are returned.

`connectToCommand :: String → IO Handle`

Executes a command with a new default shell process. The input and output streams of the new process is returned as one handle which is both readable and writable. Thus, writing to the handle produces input to the process and output from the process can be retrieved by reading from this handle. The handle should be closed with `IO.hClose` since they are not closed automatically when the process terminates.

`readCompleteFile :: String → IO String`

An action that reads the complete contents of a file and returns it. This action can be used instead of the (lazy) `readFile` action if the contents of the file might be changed.

`updateFile :: (String → String) → String → IO ()`

An action that updates the contents of a file.

`exclusiveIO :: String → IO a → IO a`

Forces the exclusive execution of an action via a lock file. For instance, (exclusiveIO "myaction.lock" act) ensures that the action "act" is not executed by two processes on the same system at the same time.

`setAssoc :: String → String → IO ()`

Defines a global association between two strings. Both arguments must be evaluable to ground terms before applying this operation.

`getAssoc :: String → IO (Maybe String)`

Gets the value associated to a string. Nothing is returned if there does not exist an associated value.

`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.2.24   Library JavaScript

A library to represent JavaScript programs.

Exported types:

`data JSExp`

>   Type of JavaScript expressions.

Exported constructors:

- `JSString :: String → JSExp`

  `JSString`

  - string constant

- `JSInt :: Int → JSExp`

  `JSInt`

  - integer constant

- `JSBool :: Bool → JSExp`

  `JSBool`

  - Boolean constant

- `JSIVar :: Int → JSExp`

  `JSIVar`

  - indexed variable

- `JSIArrayIdx :: Int → Int → JSExp`

  `JSIArrayIdx`

  - array access to index array variable

- `JSOp :: String → JSExp → JSExp → JSExp`

  `JSOp`

  - infix operator expression

- `JSFCall :: String → [JSExp] → JSExp`

  `JSFCall`

  - function call

- `JSApply :: JSExp → JSExp → JSExp`

  `JSApply`

– function call where the function is an expression

- JSLambda :: [Int] → [JSStat] → JSExp

  JSLambda

  – (anonymous) function with indexed variables as arguments

data JSStat

   Type of JavaScript statements.

Exported constructors:

- JSAssign :: JSExp → JSExp → JSStat

  JSAssign

  – assignment

- JSIf :: JSExp → [JSStat] → [JSStat] → JSStat

  JSIf

  – conditional

- JSSwitch :: JSExp → [JSBranch] → JSStat

  JSSwitch

  – switch statement

- JSPCall :: String → [JSExp] → JSStat

  JSPCall

  – procedure call

- JSReturn :: JSExp → JSStat

  JSReturn

  – return statement

- JSVarDecl :: Int → JSStat

  JSVarDecl

  – local variable declaration

data JSBranch

Exported constructors:

- JSCase :: String → [JSStat] → JSBranch

  JSCase

    – case branch

- JSDefault :: [JSStat] → JSBranch

  JSDefault

    – default branch

data JSFDecl

Exported constructors:

- JSFDecl :: String → [Int] → [JSStat] → JSFDecl

Exported functions:

showJSExp :: JSExp → String

  Shows a JavaScript expression as a string in JavaScript syntax.

showJSStat :: Int → JSStat → String

  Shows a JavaScript statement as a string in JavaScript syntax with indenting.

showJSFDecl :: JSFDecl → String

  Shows a JavaScript function declaration as a string in JavaScript syntax.

jsConsTerm :: String → [JSExp] → JSExp

  Representation of constructor terms in JavaScript.

A.2.25   Library List

Library with some useful operations on lists.

Exported functions:

elemIndex :: 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 :: 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 :: [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 :: 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.

(\\) :: [a] → [a] → [a]

Computes the difference of two lists.

union :: [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 :: [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 :: [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 :: [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 (==a) "aabbaca" == ["","","bb","c",""] split (==a) "" == [""]

`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 :: [a] → [a] → Bool`

> Checks whether a list is a prefix of another.

`isSuffixOf :: [a] → [a] → Bool`

> Checks whether a list is a suffix of another.

`isInfixOf :: [a] → [a] → Bool`

> Checks whether a list is contained in another.

`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 :: [Int] → Int`

> Returns the sum of a list of integers.

`product :: [Int] → Int`

> Returns the product of a list of integers.

`maximum :: [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 :: [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 :: (a → b → a) → a → [b] → [a]`

> `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 :: (a → b → (a,c)) → a → [b] → (a,[c])`

> 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 :: (a → b → (a,c)) → a → [b] → (a,[c])`

> 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 :: (a → Maybe (b,a)) → a → [b]`

> Builds a list from a seed value.

A.2.26   Library 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.

`(>>-) :: Maybe a → (a → Maybe b) → Maybe b`

>   Monadic bind for Maybe. Maybe can be interpreted as a monad where Nothing is interpreted as the error case by this monadic binding.

`sequenceMaybe :: [Maybe a] → Maybe [a]`

>   Monadic `sequence` for Maybe.

`mapMMaybe :: (a → Maybe b) → [a] → Maybe [b]`

>   Monadic `map` for Maybe.

`mplus :: Maybe a → Maybe a → Maybe a`

>   Combine two `Maybe`s, returning the first `Just` value, if any.

A.2.27 Library NamedSocket

Library to support network programming with sockets that are addressed by symbolic names. In contrast to raw sockets (see library `Socket`), this library uses the Curry Port Name Server to provide sockets that are addressed by symbolic names rather than numbers.

In standard applications, the server side uses the operations `listenOn` and `socketAccept` to provide some service on a named socket, and the client side uses the operation `connectToSocket` to request a service.

Exported types:

`data Socket`

> Abstract type for named sockets.

Exported constructors:

Exported functions:

`listenOn :: String → IO Socket`

> Creates a server side socket with a symbolic name.

`socketAccept :: Socket → IO (String,Handle)`

> Returns a connection of a client to a socket. The connection is returned as a pair consisting of a string identifying the client (the format of this string is implementation-dependent) and a handle to a stream communication with the client. The handle is both readable and writable.

`waitForSocketAccept :: Socket → Int → IO (Maybe (String,Handle))`

> Waits until a connection of a client to a socket is available. If no connection is available within the time limit, it returns Nothing, otherwise the connection is returned as a pair consisting of a string identifying the client (the format of this string is implementation-dependent) and a handle to a stream communication with the client.

`sClose :: Socket → IO ()`

> Closes a server socket.

`socketName :: Socket → String`

> Returns a the symbolic name of a named socket.

`connectToSocketRepeat :: Int → IO a → Int → String → IO (Maybe Handle)`

> Waits for connection to a Unix socket with a symbolic name. In contrast to `connectToSocket`, this action waits until the socket has been registered with its symbolic name.

```
connectToSocketWait :: String → IO Handle
```

> Waits for connection to a Unix socket with a symbolic name and return the handle of the connection. This action waits (possibly forever) until the socket with the symbolic name is registered.

```
connectToSocket :: String → IO Handle
```

> Creates a new connection to an existing(!) Unix socket with a symbolic name. If the symbolic name is not registered, an error is reported.

A.2.28 Library Parser

Library with functional logic parser combinators.
Adapted from: Rafael Caballero and Francisco J. Lopez-Fraguas: A Functional Logic Perspective of Parsing. In Proc. FLOPS'99, Springer LNCS 1722, pp. 85-99, 1999

Exported types:

```
type Parser a = [a] → [a]
```

```
type ParserRep a b = a → [b] → [b]
```

Exported functions:

```
(<|>) :: ([a] → [a]) → ([a] → [a]) → [a] → [a]
```

> Combines two parsers without representation in an alternative manner.

```
(<||>) :: (a → [b] → [b]) → (a → [b] → [b]) → a → [b] → [b]
```

> Combines two parsers with representation in an alternative manner.

```
(<*>) :: ([a] → [a]) → ([a] → [a]) → [a] → [a]
```

> Combines two parsers (with or without representation) in a sequential manner.

```
(>>>) :: ([a] → [a]) → b → b → [a] → [a]
```

> Attaches a representation to a parser without representation.

```
empty :: [a] → [a]
```

> The empty parser which recognizes the empty word.

```
terminal :: a → [a] → [a]
```

> A parser recognizing a particular terminal symbol.

```
satisfy :: (a → Bool) → a → [a] → [a]
```

A parser (with representation) recognizing a terminal satisfying a given predicate.

```
star :: (a → [b] → [b]) → [a] → [b] → [b]
```

A star combinator for parsers. The returned parser repeats zero or more times a parser p with representation and returns the representation of all parsers in a list.

```
some :: (a → [b] → [b]) → [a] → [b] → [b]
```

A some combinator for parsers. The returned parser repeats the argument parser (with representation) at least once.

A.2.29   Library Pretty

This library provides pretty printing combinators. The interface is that of Daan Leijen's library linear-time, bounded implementation by Olaf Chitil. Note that the implementation of `fill` and `fillBreak` is not linear-time bounded Support of ANSI escape codes for formatting and colorisation of documents in text terminals (see https://en.wikipedia.org/wiki/ANSI $escape$ code)

Exported types:

```
data Doc
```

The abstract data type Doc represents pretty documents.

Exported constructors:

Exported functions:

```
pPrint :: Doc → String
```

Standard printing with a column length of 80.

```
empty :: Doc
```

The empty document

```
isEmpty :: Doc → Bool
```

Is the document empty?

```
text :: String → Doc
```

The document (`text s`) contains the literal string `s`. The string shouldn't contain any newline (`\n`) characters. If the string contains newline characters, the function `string` should be used.

```
linesep :: String → Doc
```

The document (`linesep s`) advances to the next line and indents to the current nesting level. Document (`linesep s`) behaves like (`text s`) if the line break is undone by `group`.

`hardline :: Doc`

The document `hardline` advances to the next line and indents to the current nesting level. `hardline` cannot be undone by `group`.

`line :: Doc`

The document `line` advances to the next line and indents to the current nesting level. Document `line` behaves like (`text " "`) if the line break is undone by `group`.

`linebreak :: Doc`

The document `linebreak` advances to the next line and indents to the current nesting level. Document `linebreak` behaves like (`text ""`) if the line break is undone by `group`.

`softline :: Doc`

The document `softline` behaves like `space` if the resulting output fits the page, otherwise it behaves like `line`. `softline = group line`

`softbreak :: Doc`

The document `softbreak` behaves like (`text ""`) if the resulting output fits the page, otherwise it behaves like `line`. `softbreak = group linebreak`

`group :: Doc → Doc`

The combinator `group` is used to specify alternative layouts. The document (`group x`) undoes all line breaks in document `x`. The resulting line is added to the current line if that fits the page. Otherwise, the document `x` is rendered without any changes.

`nest :: Int → Doc → Doc`

The document (`nest i d`) renders document `d` with the current indentation level increased by `i` (See also `hang`, `align` and `indent`).

```
nest 2 (text "hello" $$ text "world") $$ text "!"
```

outputs as:

```
hello
  world
!
```

`hang :: Int → Doc → Doc`

The combinator `hang` implements hanging indentation. The document (`hang i d`) renders document `d` with a nesting level set to the current column plus `i`. The following example uses hanging indentation for some text:

```
test = hang 4
           (fillSep
              (map text
                    (words "the hang combinator indents these words !")))
```

Which lays out on a page with a width of 20 characters as:

```
the hang combinator
    indents these
    words !
```

The hang combinator is implemented as:

```
hang i x  = align (nest i x)
```

`align :: Doc → Doc`

The document (`align d`) renders document `d` with the nesting level set to the current column. It is used for example to implement `hang`‘.

As an example, we will put a document right above another one, regardless of the current nesting level:

```
x $$ y  = align (x $$ y)
test    = text "hi" <+> (text "nice" $$ text "world")
```

which will be layed out as:

```
hi nice
   world
```

`indent :: Int → Doc → Doc`

The document (`indent i d`) indents document `d` with `i` spaces.

```
test  = indent 4 (fillSep (map text
          (words "the indent combinator indents these words !")))
```

Which lays out with a page width of 20 as:

```
      the indent
      combinator
      indents these
      words !
```

`combine :: Doc → Doc → Doc → Doc`

The document (`combine c d1 d2`) combines document `d1` and `d2` with document `c` in between using (`<>`) with identity `empty`. Thus, the following equations hold.

```
combine c d1    empty == d1
combine c empty d2    == d2
combine c d1    d2    == d1 <> c <> d2 if neither d1 nor d2 are empty
```

`(<>) :: Doc → Doc → Doc`

The document (`x <> y`) concatenates document `x` and document `y`. It is an associative operation having `empty` as a left and right unit.

`(<+>) :: Doc → Doc → Doc`

The document (`x <+> y`) concatenates document `x` and `y` with a `space` in between with identity `empty`.

`($$) :: Doc → Doc → Doc`

The document (`x $$ y`) concatenates document `x` and `y` with a `line` in between with identity `empty`.

`(<$+$>) :: Doc → Doc → Doc`

The document (`x <$+$> y`) concatenates document `x` and `y` with a blank line in between with identity `empty`.

`(</>) :: Doc → Doc → Doc`

The document (`x </> y`) concatenates document `x` and `y` with a `softline` in between with identity `empty`. This effectively puts `x` and `y` either next to each other (with a `space` in between) or underneath each other.

`(<$$>) :: Doc → Doc → Doc`

The document (`x <$$> y`) concatenates document `x` and `y` with a `linebreak` in between with identity `empty`.

`(<//>) :: Doc → Doc → Doc`

The document (`x <//> y`) concatenates document `x` and `y` with a `softbreak` in between with identity `empty`. This effectively puts `x` and `y` either right next to each other or underneath each other.

`(<$!$>) :: Doc → Doc → Doc`

> The document `(x <$!$> y)` concatenates document `x` and `y` with a `hardline` in between with identity `empty`. This effectively puts `x` and `y` underneath each other.

`compose :: (Doc → Doc → Doc) → [Doc] → Doc`

> The document `(compose f xs)` concatenates all documents `xs` with function `f`. Function `f` should be like `(<+>)`, `($$)` and so on.

`hsep :: [Doc] → Doc`

> The document `(hsep xs)` concatenates all documents `xs` horizontally with `(<+>)`.

`vsep :: [Doc] → Doc`

> The document `(vsep xs)` concatenates all documents `xs` vertically with `($$)`. If a group undoes the line breaks inserted by `vsep`, all documents are separated with a `space`.

```
someText = map text (words ("text to lay out"))
test     = text "some" <+> vsep someText
```

> This is layed out as:

```
some text
to
lay
out
```

> The `align` combinator can be used to align the documents under their first element:

```
test     = text "some" <+> align (vsep someText)
```

> This is printed as:

```
some text
    to
    lay
    out
```

`vsepBlank :: [Doc] → Doc`

> The document `vsep xs` concatenates all documents `xs` vertically with `(<$+$>)`. If a group undoes the line breaks inserted by `vsepBlank`, all documents are separated with a `space`.

`fillSep :: [Doc] → Doc`

The document (`fillSep xs`) concatenates documents `xs` horizontally with ($</>$) as long as its fits the page, than inserts a `line` and continues doing that for all documents in `xs`. `fillSep xs = foldr (`$</>$`) empty xs`

`sep :: [Doc]` $\rightarrow$ `Doc`

The document (`sep xs`) concatenates all documents `xs` either horizontally with ($<+>$), if it fits the page, or vertically with ($\$\$$). `sep xs = group (vsep xs)`

`hcat :: [Doc]` $\rightarrow$ `Doc`

The document (`hcat xs`) concatenates all documents `xs` horizontally with ($<>$).

`vcat :: [Doc]` $\rightarrow$ `Doc`

The document (`vcat xs`) concatenates all documents `xs` vertically with ($<\$\$>$). If a `group` undoes the line breaks inserted by `vcat`, all documents are directly concatenated.

`fillCat :: [Doc]` $\rightarrow$ `Doc`

The document (`fillCat xs`) concatenates documents `xs` horizontally with ($<//>$) as long as its fits the page, than inserts a `linebreak` and continues doing that for all documents in `xs`. `fillCat xs = foldr (`$<//>$`) empty xs`

`cat :: [Doc]` $\rightarrow$ `Doc`

The document (`cat xs`) concatenates all documents `xs` either horizontally with ($<>$), if it fits the page, or vertically with ($<\$\$>$). `cat xs = group (vcat xs)`

`punctuate :: Doc` $\rightarrow$ `[Doc]` $\rightarrow$ `[Doc]`

(`punctuate p xs`) concatenates all documents `xs` with document `p` except for the last document.

```
someText = map text ["words","in","a","tuple"]
test     = parens (align (cat (punctuate comma someText)))
```

This is layed out on a page width of 20 as:

```
(words,in,a,tuple)
```

But when the page width is 15, it is layed out as:

```
(words,
 in,
 a,
 tuple)
```

(If you want put the commas in front of their elements instead of at the end, you should
use `tupled` or, in general, `encloseSep`.)

encloseSep :: Doc → Doc → Doc → [Doc] → Doc

The document (`encloseSep l r s xs`) concatenates the documents `xs` seperated by `s`
and encloses the resulting document by `l` and `r`. The documents are rendered horizon-
tally if that fits the page. Otherwise they are aligned vertically. All seperators are put
in front of the elements.

For example, the combinator `list` can be defined with `encloseSep`:

```
list xs  = encloseSep lbracket rbracket comma xs
test     = text "list" <+> (list (map int [10,200,3000]))
```

Which is layed out with a page width of 20 as:

```
list [10,200,3000]
```

But when the page width is 15, it is layed out as:

```
list [10
     ,200
     ,3000]
```

encloseSepSpaced :: Doc → Doc → Doc → [Doc] → Doc

The document (`encloseSepSpaced l r s xs`) concatenates the documents `xs` seper-
ated by `s` and encloses the resulting document by `l` and `r`. In addition, after each
occurrence of `s`, after `l`, and before `r`, a `space` is inserted. The documents are rendered
horizontally if that fits the page. Otherwise they are aligned vertically. All seperators
are put in front of the elements.

hEncloseSep :: Doc → Doc → Doc → [Doc] → Doc

The document (`hEncloseSep l r s xs`) concatenates the documents `xs` seperated by
`s` and encloses the resulting document by `l` and `r`.

The documents are rendered horizontally.

fillEncloseSep :: Doc → Doc → Doc → [Doc] → Doc

The document (`fillEncloseSep l r s xs`) concatenates the documents `xs` seperated
by `s` and encloses the resulting document by `l` and `r`.

The documents are rendered horizontally if that fits the page. Otherwise they are aligned
vertically. All seperators are put in front of the elements.

fillEncloseSepSpaced :: Doc → Doc → Doc → [Doc] → Doc

The document (`fillEncloseSepSpaced l r s xs`) concatenates the documents `xs` seperated by `s` and encloses the resulting document by `l` and `r`. In addition, after each occurrence of `s`, after `l`, and before `r`, a `space` is inserted.

The documents are rendered horizontally if that fits the page. Otherwise, they are aligned vertically. All seperators are put in front of the elements.

`list :: [Doc] → Doc`

The document (`list xs`) comma seperates the documents `xs` and encloses them in square brackets. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All comma seperators are put in front of the elements.

`listSpaced :: [Doc] → Doc`

Spaced version of `list`

`set :: [Doc] → Doc`

The document (`set xs`) comma seperates the documents `xs` and encloses them in braces. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All comma seperators are put in front of the elements.

`setSpaced :: [Doc] → Doc`

Spaced version of `set`

`tupled :: [Doc] → Doc`

The document (`tupled xs`) comma seperates the documents `xs` and encloses them in parenthesis. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All comma seperators are put in front of the elements.

`tupledSpaced :: [Doc] → Doc`

Spaced version of `tupled`

`semiBraces :: [Doc] → Doc`

The document (`semiBraces xs`) seperates the documents `xs` with semi colons and encloses them in braces. The documents are rendered horizontally if that fits the page. Otherwise they are aligned vertically. All semi colons are put in front of the elements.

`semiBracesSpaced :: [Doc] → Doc`

Spaced version of `semiBraces`

`enclose :: Doc → Doc → Doc → Doc`

The document (`enclose l r x`) encloses document `x` between documents `l` and `r` using (`<>`). `enclose l r x = l <> x <> r`

`squotes :: Doc → Doc`

Document (`squotes x`) encloses document `x` with single quotes "'".

`dquotes :: Doc → Doc`

Document (`dquotes x`) encloses document `x` with double quotes.

`bquotes :: Doc → Doc`

Document (`bquotes x`) encloses document `x` with back quotes "`".

`parens :: Doc → Doc`

Document (`parens x`) encloses document `x` in parenthesis, "(" and ")".

`parensIf :: Bool → Doc → Doc`

Document (`parensIf x`) encloses document `x` in parenthesis,"(" and ")", iff the condition is true.

`angles :: Doc → Doc`

Document (`angles x`) encloses document `x` in angles, "<" and ">".

`braces :: Doc → Doc`

Document (`braces x`) encloses document `x` in braces, "{" and "}".

`brackets :: Doc → Doc`

Document (`brackets x`) encloses document `x` in square brackets, "[" and "]".

`char :: Char → Doc`

The document (`char c`) contains the literal character `c`. The character should not be a newline (`\n`), the function `line` should be used for line breaks.

`string :: String → Doc`

The document (`string s`) concatenates all characters in `s` using `line` for newline characters and `char` for all other characters. It is used instead of `text` whenever the text contains newline characters.

`int :: Int → Doc`

The document (`int i`) shows the literal integer `i` using `text`.

`float :: Float → Doc`

The document (`float f`) shows the literal float `f` using `text`.

`lparen :: Doc`

The document `lparen` contains a left parenthesis, "(".

`rparen :: Doc`

The document **rparen** contains a right parenthesis, ")".

langle :: Doc

The document **langle** contains a left angle, "<".

rangle :: Doc

The document **rangle** contains a right angle, ">".

lbrace :: Doc

The document **lbrace** contains a left brace, "{".

rbrace :: Doc

The document **rbrace** contains a right brace, "}".

lbracket :: Doc

The document **lbracket** contains a left square bracket, "[".

rbracket :: Doc

The document **rbracket** contains a right square bracket, "]".

squote :: Doc

The document **squote** contains a single quote, "'".

dquote :: Doc

The document **dquote** contains a double quote.

semi :: Doc

The document **semi** contains a semi colon, ";".

colon :: Doc

The document **colon** contains a colon, ":".

comma :: Doc

The document **comma** contains a comma, ",".

space :: Doc

The document **space** contains a single space, " ".

```
x <+> y   = x <> space <> y
```

dot :: Doc

The document **dot** contains a single dot, ".".

`backslash :: Doc`

> The document `backslash` contains a back slash, `"\"`.

`equals :: Doc`

> The document `equals` contains an equal sign, `"="`.

`larrow :: Doc`

> The document `larrow` contains a left arrow sign, `"<-"`.

`rarrow :: Doc`

> The document `rarrow` contains a right arrow sign, `"->"`.

`doubleArrow :: Doc`

> The document `doubleArrow` contains an double arrow sign, `"=>"`.

`doubleColon :: Doc`

> The document `doubleColon` contains a double colon sign, `"::"`.

`bar :: Doc`

> The document `bar` contains a vertical bar sign, `"|"`.

`at :: Doc`

> The document `at` contains an at sign, `"@"`.

`tilde :: Doc`

> The document `tilde` contains a tilde sign, `"~"`.

`fill :: Int → Doc → Doc`

> The document (`fill i d`) renders document `d`. It than appends `spaces` until the width is equal to `i`. If the width of `d` is already larger, nothing is appended. This combinator is quite useful in practice to output a list of bindings. The following example demonstrates this.

```
types  = [("empty","Doc")
         ,("nest","Int -> Doc -> Doc")
         ,("linebreak","Doc")]


ptype (name,tp)
      = fill 6 (text name) <+> text "::" <+> text tp


test   = text "let" <+> align (vcat (map ptype types))
```

Which is layed out as:

```
let empty  :: Doc
    nest   :: Int -> Doc -> Doc
    linebreak :: Doc
```

Note that `fill` is not guaranteed to be linear-time bounded since it has to compute the width of a document before pretty printing it

`fillBreak :: Int → Doc → Doc`

The document (`fillBreak i d`) first renders document `d`. It than appends `spaces` until the width is equal to `i`. If the width of `d` is already larger than `i`, the nesting level is increased by `i` and a `line` is appended. When we redefine `ptype` in the previous example to use `fillBreak`, we get a useful variation of the previous output:

```
ptype (name,tp)
    = fillBreak 6 (text name) <+> text "::" <+> text tp
```

The output will now be:

```
let empty  :: Doc
    nest   :: Int -> Doc -> Doc
    linebreak
           :: Doc
```

Note that `fillBreak` is not guaranteed to be linear-time bounded since it has to compute the width of a document before pretty printing it

`bold :: Doc → Doc`

The document (`bold d`) displays document `d` with bold text

`faint :: Doc → Doc`

The document (`faint d`) displays document `d` with faint text

`blinkSlow :: Doc → Doc`

The document (`blinkSlow d`) displays document `d` with slowly blinking text (rarely supported)

`blinkRapid :: Doc → Doc`

The document (`blinkRapid d`) displays document `d` with rapidly blinking text (rarely supported)

`italic :: Doc → Doc`

The document (`italic d`) displays document `d` with italicized text (rarely supported)

`underline :: Doc → Doc`

The document (`underline d`) displays document `d` with underlined text

`crossout :: Doc → Doc`

The document (`crossout d`) displays document `d` with crossed out text

`inverse :: Doc → Doc`

The document (`inverse d`) displays document `d` with inversed coloring, i.e. use text color of `d` as background color and background color of `d` as text color

`black :: Doc → Doc`

The document (`black d`) displays document `d` with black text color

`red :: Doc → Doc`

The document (`red d`) displays document `d` with red text color

`green :: Doc → Doc`

The document (`green d`) displays document `d` with green text color

`yellow :: Doc → Doc`

The document (`yellow d`) displays document `d` with yellow text color

`blue :: Doc → Doc`

The document (`blue d`) displays document `d` with blue text color

`magenta :: Doc → Doc`

The document (`magenta d`) displays document `d` with magenta text color

`cyan :: Doc → Doc`

The document (`cyan d`) displays document `d` with cyan text color

`white :: Doc → Doc`

The document (`white d`) displays document `d` with white text color

`bgBlack :: Doc → Doc`

The document (`bgBlack d`) displays document `d` with black background color

`bgRed :: Doc → Doc`

The document (`bgRed d`) displays document `d` with red background color

`bgGreen :: Doc → Doc`

The document (`bgGreen d`) displays document `d` with green background color

`bgYellow :: Doc → Doc`

　　The document (`bgYellow d`) displays document `d` with yellow background color

`bgBlue :: Doc → Doc`

　　The document (`bgBlue d`) displays document `d` with blue background color

`bgMagenta :: Doc → Doc`

　　The document (`bgMagenta d`) displays document `d` with magenta background color

`bgCyan :: Doc → Doc`

　　The document (`bgCyan d`) displays document `d` with cyan background color

`bgWhite :: Doc → Doc`

　　The document (`bgWhite d`) displays document `d` with white background color

`pretty :: Int → Doc → String`

　　(`pretty w d`) pretty prints document `d` with a page width of `w` characters

A.2.30　Library Profile

Preliminary library to support profiling.

Exported types:

`data ProcessInfo`

　　The data type for representing information about the state of a Curry process.

Exported constructors:

- `RunTime :: ProcessInfo`

  RunTime

  – the run time in milliseconds

- `ElapsedTime :: ProcessInfo`

  ElapsedTime

  – the elapsed time in milliseconds

- `Memory :: ProcessInfo`

  Memory

  – the total memory in bytes

- `Code :: ProcessInfo`

  `Code`

  – the size of the code area in bytes

- `Stack :: ProcessInfo`

  `Stack`

  – the size of the local stack for recursive functions in bytes

- `Heap :: ProcessInfo`

  `Heap`

  – the size of the heap to store term structures in bytes

- `Choices :: ProcessInfo`

  `Choices`

  – the size of the choicepoint stack

- `GarbageCollections :: ProcessInfo`

  `GarbageCollections`

  – the number of garbage collections performed

Exported functions:

`getProcessInfos :: IO [(ProcessInfo,Int)]`

Returns various informations about the current state of the Curry process. Note that the returned values are implementation dependent so that one should interpret them with care!

Note for kics2 users: Since GHC version 7.x, one has to set the run-time option `-T` when this operation is used. This can be done by the kics2 command

`:set rts -T`

`garbageCollectorOff :: IO ()`

Turns off the garbage collector of the run-time system (if possible). This could be useful to get more precise data of memory usage.

`garbageCollectorOn :: IO ()`

Turns on the garbage collector of the run-time system (if possible).

`garbageCollect :: IO ()`

Invoke the garbage collector (if possible). This could be useful before run-time critical operations.

showMemInfo :: [(ProcessInfo,Int)] $\rightarrow$ String

Get a human readable version of the memory situation from the process infos.

printMemInfo :: IO ()

Print a human readable version of the current memory situation of the Curry process.

profileTime :: IO a $\rightarrow$ IO a

Print the time needed to execute a given IO action.

profileTimeNF :: a $\rightarrow$ IO ()

Evaluates the argument to normal form and print the time needed for this evaluation.

profileSpace :: IO a $\rightarrow$ IO a

Print the time and space needed to execute a given IO action. During the executation, the garbage collector is turned off to get the total space usage.

profileSpaceNF :: a $\rightarrow$ IO ()

Evaluates the argument to normal form and print the time and space needed for this evaluation. During the evaluation, the garbage collector is turned off to get the total space usage.

A.2.31   Library Prolog

A library defining a representation for Prolog programs together with a simple pretty printer. It does not cover all aspects of Prolog but might be useful for applications generating Prolog programs.

Exported types:

data PlClause

A Prolog clause is either a program clause consisting of a head and a body, or a directive or a query without a head.

Exported constructors:

- PlClause :: String $\rightarrow$ [PlTerm] $\rightarrow$ [PlGoal] $\rightarrow$ PlClause

- PlDirective :: [PlGoal] $\rightarrow$ PlClause

- PlQuery :: [PlGoal] $\rightarrow$ PlClause

data PlGoal

A Prolog goal is a literal, a negated goal, or a conditional.

Exported constructors:

- PlLit :: String → [PlTerm] → PlGoal

- PlNeg :: [PlGoal] → PlGoal

- PlCond :: [PlGoal] → [PlGoal] → [PlGoal] → PlGoal

data PlTerm

A Prolog term is a variable, atom, number, or structure.

Exported constructors:

- PlVar :: String → PlTerm

- PlAtom :: String → PlTerm

- PlInt :: Int → PlTerm

- PlFloat :: Float → PlTerm

- PlStruct :: String → [PlTerm] → PlTerm

Exported functions:

plList :: [PlTerm] → PlTerm

A Prolog list of Prolog terms.

showPlProg :: [PlClause] → String

Shows a Prolog program in standard Prolog syntax.

showPlClause :: PlClause → String

showPlGoals :: [PlGoal] → String

showPlGoal :: PlGoal → String

showPlTerm :: PlTerm → String

A.2.32 Library PropertyFile

A library to read and update files containing properties in the usual equational syntax, i.e., a property is defined by a line of the form prop=value where prop starts with a letter. All other lines (e.g., blank lines or lines starting with # are considered as comment lines and are ignored.

Exported functions:

readPropertyFile :: String → IO [(String,String)]

    Reads a property file and returns the list of properties. Returns empty list if the property file does not exist.

updatePropertyFile :: String → String → String → IO ()

    Update a property in a property file or add it, if it is not already there.

A.2.33 Library Read

Library with some functions for reading special tokens.
This library is included for backward compatibility. You should use the library ReadNumeric which provides a better interface for these functions.

Exported functions:

readNat :: String → Int

    Read a natural number in a string. The string might contain leadings blanks and the the number is read up to the first non-digit.

readInt :: String → Int

    Read a (possibly negative) integer in a string. The string might contain leadings blanks and the the integer is read up to the first non-digit.

readHex :: String → Int

    Read a hexadecimal number in a string. The string might contain leadings blanks and the the integer is read up to the first non-heaxdecimal digit.

A.2.34 Library ReadNumeric

Library with some functions for reading and converting numeric tokens.

Exported functions:

readInt :: String → Maybe (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. If the string does not start with an integer token, Nothing is returned, otherwise the result is Just (v, s), where v is the value of the integer and s is the remaing string without the integer token.

readNat :: String → Maybe (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. If the string does not start with a natural number token, Nothing is returned, otherwise the result is Just (v, s) where v is the value of the number and s is the remaing string without the number token.

`readHex :: String → Maybe (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. If the string does not start with a hexadecimal number token, `Nothing` is returned, otherwise the result is `Just (v, s)` where v is the value of the number and s is the remaing string without the number token.

`readOct :: String → Maybe (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. If the string does not start with an octal number token, `Nothing` is returned, otherwise the result is `Just (v, s)` where v is the value of the number and s is the remaing string without the number token.

`readBin :: String → Maybe (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. If the string does not start with a binary number token, `Nothing` is returned, otherwise the result is `Just (v, s)` where v is the value of the number and s is the remaing string without the number token.

A.2.35 Library ReadShowTerm

Library for converting ground terms to strings and vice versa.

Exported functions:

`showTerm :: a → String`

Transforms a ground(!) term into a string representation in standard prefix notation. Thus, showTerm suspends until its argument is ground. This function is similar to the prelude function `show` but can read the string back with `readUnqualifiedTerm` (provided that the constructor names are unique without the module qualifier).

`showQTerm :: a → String`

Transforms a ground(!) term into a string representation in standard prefix notation. Thus, showTerm suspends until its argument is ground. Note that this function differs from the prelude function `show` since it prefixes constructors with their module name in order to read them back with `readQTerm`.

`readsUnqualifiedTerm :: [String] → String → [(a,String)]`

Transform a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The first argument is a non-empty list of module qualifiers that are tried to prefix the constructor in the string in order to get the qualified

155

constructors (that must be defined in the current program!). 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.

readUnqualifiedTerm :: [String] → String → a

Transforms a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The first argument is a non-empty list of module qualifiers that are tried to prefix the constructor in the string in order to get the qualified constructors (that must be defined in the current program!).

Example: readUnqualifiedTerm ["Prelude"] "Just 3" evaluates to (Just 3)

readsTerm :: String → [(a,String)]

For backward compatibility. Should not be used since their use can be problematic in case of constructors with identical names in different modules.

readTerm :: String → a

For backward compatibility. Should not be used since their use can be problematic in case of constructors with identical names in different modules.

readsQTerm :: String → [(a,String)]

Transforms a string containing a term in standard prefix notation with qualified constructor names into the corresponding data term. 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.

readQTerm :: String → a

Transforms a string containing a term in standard prefix notation with qualified constructor names into the corresponding data term.

readQTermFile :: String → IO a

Reads a file containing a string representation of a term in standard prefix notation and returns the corresponding data term.

readQTermListFile :: String → IO [a]

Reads a file containing lines with string representations of terms of the same type and returns the corresponding list of data terms.

writeQTermFile :: String → a → IO ()

Writes a ground term into a file in standard prefix notation.

writeQTermListFile :: String → [a] → IO ()

Writes a list of ground terms into a file. Each term is written into a separate line which might be useful to modify the file with a standard text editor.

A.2.36 Library SetFunctions

This module contains an implementation of set functions. The general idea of set functions is described in:

> S. Antoy, M. Hanus: Set Functions for Functional Logic Programming Proc. 11th International Conference on Principles and Practice of Declarative Programming (PPDP'09), pp. 73-82, ACM Press, 2009

Intuition: If `f` is an n-ary function, then `(setn f)` is a set-valued function that collects all non-determinism caused by f (but not the non-determinism caused by evaluating arguments!) in a set. Thus, `(setn f a1 ...  an)` returns the set of all values of `(f b1 ...  bn)` where `b1`,...,`bn` are values of the arguments `a1`,...,`an` (i.e., the arguments are evaluated "outside" this capsule so that the non-determinism caused by evaluating these arguments is not captured in this capsule but yields several results for `(setn...)`. Similarly, logical variables occuring in `a1`,...,`an` are not bound inside this capsule.

The set of values returned by a set function is represented by an abstract type `Values` on which several operations are defined in this module. Actually, it is a multiset of values, i.e., duplicates are not removed.

The handling of failures and nested occurrences of set functions is not specified in the previous paper. Thus, a detailed description of the semantics of set functions as implemented in this library can be found in the paper

> J. Christiansen, M. Hanus, F. Reck, D. Seidel: A Semantics for Weakly Encapsulated Search in Functional Logic Programs Proc. 15th International Conference on Principles and Practice of Declarative Programming (PPDP'13), pp. 49-60, ACM Press, 2013

Exported types:

`data Values`

> Abstract type representing multisets of values.

Exported constructors:

Exported functions:

`set0 :: a → Values a`

> Combinator to transform a 0-ary function into a corresponding set function.

`set0With :: (SearchTree a → ValueSequence a) → a → Values a`

> Combinator to transform a 0-ary function into a corresponding set function that uses a given strategy to compute its values.

`set1 :: (a → b) → a → Values b`

> Combinator to transform a unary function into a corresponding set function.

`set1With :: (SearchTree a → ValueSequence a) → (b → a) → b → Values a`

> Combinator to transform a unary function into a corresponding set function that uses
> a given strategy to compute its values.

`set2 :: (a → b → c) → a → b → Values c`

> Combinator to transform a binary function into a corresponding set function.

`set2With :: (SearchTree a → ValueSequence a) → (b → c → a) → b → c → Values a`

> Combinator to transform a binary function into a corresponding set function that uses
> a given strategy to compute its values.

`set3 :: (a → b → c → d) → a → b → c → Values d`

> Combinator to transform a function of arity 3 into a corresponding set function.

`set3With :: (SearchTree a → ValueSequence a) → (b → c → d → a) → b → c → d → Values a`

> Combinator to transform a function of arity 3 into a corresponding set function that
> uses a given strategy to compute its values.

`set4 :: (a → b → c → d → e) → a → b → c → d → Values e`

> Combinator to transform a function of arity 4 into a corresponding set function.

`set4With :: (SearchTree a → ValueSequence a) → (b → c → d → e → a) → b → c → d → e → Values a`

> Combinator to transform a function of arity 4 into a corresponding set function that
> uses a given strategy to compute its values.

`set5 :: (a → b → c → d → e → f) → a → b → c → d → e → Values f`

> Combinator to transform a function of arity 5 into a corresponding set function.

`set5With :: (SearchTree a → ValueSequence a) → (b → c → d → e → f → a) → b → c → d → e → f → Values a`

> Combinator to transform a function of arity 5 into a corresponding set function that
> uses a given strategy to compute its values.

`set6 :: (a → b → c → d → e → f → g) → a → b → c → d → e → f → Values g`

> Combinator to transform a function of arity 6 into a corresponding set function.

`set6With :: (SearchTree a → ValueSequence a) → (b → c → d → e → f → g → a) → b → c → d → e → f → g → Values a`

Combinator to transform a function of arity 6 into a corresponding set function that uses a given strategy to compute its values.

```
set7 :: (a → b → c → d → e → f → g → h) → a → b → c → d → e → f → g
→ Values h
```

Combinator to transform a function of arity 7 into a corresponding set function.

```
set7With :: (SearchTree a → ValueSequence a) → (b → c → d → e → f → g → h
→ a) → b → c → d → e → f → g → h → Values a
```

Combinator to transform a function of arity 7 into a corresponding set function that uses a given strategy to compute its values.

```
isEmpty :: Values a → Bool
```

Is a multiset of values empty?

```
notEmpty :: Values a → Bool
```

Is a multiset of values not empty?

```
valueOf :: a → Values a → Bool
```

Is some value an element of a multiset of values?

```
choose :: Values a → (a,Values a)
```

Chooses (non-deterministically) some value in a multiset of values and returns the chosen value and the remaining multiset of values. Thus, if we consider the operation `chooseValue` by

```
chooseValue x = fst (choose x)
```

then (`set1 chooseValue`) is the identity on value sets, i.e., (`set1 chooseValue s`) contains the same elements as the value set `s`.

```
chooseValue :: Values a → a
```

Chooses (non-deterministically) some value in a multiset of values and returns the chosen value. Thus, (`set1 chooseValue`) is the identity on value sets, i.e., (`set1 chooseValue s`) contains the same elements as the value set `s`.

```
select :: Values a → (a,Values a)
```

Selects (indeterministically) some value in a multiset of values and returns the selected value and the remaining multiset of values. Thus, `select` has always at most one value. It fails if the value set is empty.

NOTE: The usage of this operation is only safe (i.e., does not destroy completeness) if all values in the argument set are identical.

`selectValue :: Values a → a`

> Selects (indeterministically) some value in a multiset of values and returns the selected value. Thus, `selectValue` has always at most one value. It fails if the value set is empty.
>
> **NOTE:** The usage of this operation is only safe (i.e., does not destroy completeness) if all values in the argument set are identical.

`mapValues :: (a → b) → Values a → Values b`

> Accumulates all elements of a multiset of values by applying a binary operation. This is similarly to fold on lists, but the binary operation must be **commutative** so that the result is independent of the order of applying this operation to all elements in the multiset.

`foldValues :: (a → a → a) → a → Values a → a`

> Accumulates all elements of a multiset of values by applying a binary operation. This is similarly to fold on lists, but the binary operation must be **commutative** so that the result is independent of the order of applying this operation to all elements in the multiset.

`minValue :: (a → a → Bool) → Values a → a`

> Returns the minimal element of a non-empty multiset of values with respect to a given total ordering on the elements.

`maxValue :: (a → a → Bool) → Values a → a`

> Returns the maximal element of a non-empty multiset of value with respect to a given total ordering on the elements.

`values2list :: Values a → IO [a]`

> Puts all elements of a multiset of values in a list. Since the order of the elements in the list might depend on the time of the computation, this operation is an I/O action.

`printValues :: Values a → IO ()`

> Prints all elements of a multiset of values.

`sortValues :: Values a → [a]`

> Transforms a multiset of values into a list sorted by the standard term ordering. As a consequence, the multiset of values is completely evaluated.

`sortValuesBy :: (a → a → Bool) → Values a → [a]`

> Transforms a multiset of values into a list sorted by a given ordering on the values. As a consequence, the multiset of values is completely evaluated. In order to ensure that the result of this operation is independent of the evaluation order, the given ordering must be a total order.

A.2.37 Library Socket

Library to support network programming with sockets. In standard applications, the server side uses the operations `listenOn` and `socketAccept` to provide some service on a socket, and the client side uses the operation `connectToSocket` to request a service.

Exported types:

`data Socket`

> The abstract type of sockets.

Exported constructors:

Exported functions:

`listenOn :: Int → IO Socket`

> Creates a server side socket bound to a given port number.

`listenOnFresh :: IO (Int,Socket)`

> Creates a server side socket bound to a free port. The port number and the socket is returned.

`socketAccept :: Socket → IO (String,Handle)`

> Returns a connection of a client to a socket. The connection is returned as a pair consisting of a string identifying the client (the format of this string is implementation-dependent) and a handle to a stream communication with the client. The handle is both readable and writable.

`waitForSocketAccept :: Socket → Int → IO (Maybe (String,Handle))`

> Waits until a connection of a client to a socket is available. If no connection is available within the time limit, it returns Nothing, otherwise the connection is returned as a pair consisting of a string identifying the client (the format of this string is implementation-dependent) and a handle to a stream communication with the client.

`sClose :: Socket → IO ()`

> Closes a server socket.

`connectToSocket :: String → Int → IO Handle`

> Creates a new connection to a Unix socket.

A.2.38 Library State

This library provides an implementation of the state monad.

Exported types:

```
type State a b = a → (b,a)
```

Exported functions:

```
bindS :: (a → (b,a)) → (b → a → (c,a)) → a → (c,a)
```

```
bindS_ :: (a → (b,a)) → (a → (c,a)) → a → (c,a)
```

```
returnS :: a → b → (a,b)
```

```
getS :: a → (a,a)
```

```
putS :: a → a → ((),a)
```

```
modifyS :: (a → a) → a → ((),a)
```

```
sequenceS :: [a → (b,a)] → a → ([b],a)
```

```
sequenceS_ :: [a → (b,a)] → a → ((),a)
```

```
mapS :: (a → b → (c,b)) → [a] → b → ([c],b)
```

```
mapS_ :: (a → b → (c,b)) → [a] → b → ((),b)
```

```
runState :: (a → (b,a)) → a → (b,a)
```

```
evalState :: (a → (b,a)) → a → b
```

```
execState :: (a → (b,a)) → a → a
```

```
liftS :: (a → b) → (c → (a,c)) → c → (b,c)
```

```
liftS2 :: (a → b → c) → (d → (a,d)) → (d → (b,d)) → d → (c,d)
```

A.2.39 Library System

Library to access parts of the system environment.

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.

getArgs :: IO [String]

>Returns the list of the program's command line arguments. The program name is not included.

getEnviron :: String → IO String

>Returns the value of an environment variable. The empty string is returned for undefined environment variables.

setEnviron :: String → String → IO ()

>Set an environment variable to a value. The new value will be passed to subsequent shell commands (see system) and visible to subsequent calls to getEnviron (but it is not visible in the environment of the process that started the program execution).

unsetEnviron :: String → IO ()

>Removes an environment variable that has been set by setEnviron.

getHostname :: IO String

>Returns the hostname of the machine running this process.

getPID :: IO Int

>Returns the process identifier of the current Curry process.

getProgName :: IO String

>Returns the name of the current program, i.e., the name of the main module currently executed.

system :: String → IO Int

>Executes a shell command and return with the exit code of the command. An exit status of zero means successful execution.

```
exitWith :: Int → IO a
```

> Terminates the execution of the current Curry program and returns the exit code given by the argument. An exit code of zero means successful execution.

```
sleep :: Int → IO ()
```

> The evaluation of the action (sleep n) puts the Curry process asleep for n seconds.

```
isPosix :: Bool
```

> Is the underlying operating system a POSIX system (unix, MacOS)?

```
isWindows :: Bool
```

> Is the underlying operating system a Windows system?

A.2.40 Library Time

Library for handling date and time information.

Exported types:

```
data ClockTime
```

> ClockTime represents a clock time in some internal representation.

> *Exported constructors:*

```
data CalendarTime
```

> A calendar time is presented in the following form: (CalendarTime year month day hour minute second timezone) where timezone is an integer representing the timezone as a difference to UTC time in seconds.

> *Exported constructors:*

> • CalendarTime :: Int → Int → Int → Int → Int → Int → Int → CalendarTime

Exported functions:

```
ctYear :: CalendarTime → Int
```

> The year of a calendar time.

```
ctMonth :: CalendarTime → Int
```

> The month of a calendar time.

```
ctDay :: CalendarTime → Int
```

> The day of a calendar time.

`ctHour :: CalendarTime → Int`

> The hour of a calendar time.

`ctMin :: CalendarTime → Int`

> The minute of a calendar time.

`ctSec :: CalendarTime → Int`

> The second of a calendar time.

`ctTZ :: CalendarTime → Int`

> The time zone of a calendar time. The value of the time zone is the difference to UTC time in seconds.

`getClockTime :: IO ClockTime`

> Returns the current clock time.

`getLocalTime :: IO CalendarTime`

> Returns the local calendar time.

`clockTimeToInt :: ClockTime → Int`

> Transforms a clock time into a unique integer. It is ensured that clock times that differs in at least one second are mapped into different integers.

`toCalendarTime :: ClockTime → IO CalendarTime`

> Transforms a clock time into a calendar time according to the local time (if possible). Since the result depends on the local environment, it is an I/O operation.

`toUTCTime :: ClockTime → CalendarTime`

> Transforms a clock time into a standard UTC calendar time. Thus, this operation is independent on the local time.

`toClockTime :: CalendarTime → ClockTime`

> Transforms a calendar time (interpreted as UTC time) into a clock time.

`calendarTimeToString :: CalendarTime → String`

> Transforms a calendar time into a readable form.

`toDayString :: CalendarTime → String`

> Transforms a calendar time into a string containing the day, e.g., "September 23, 2006".

`toTimeString :: CalendarTime → String`

> Transforms a calendar time into a string containing the time.

`addSeconds :: Int → ClockTime → ClockTime`

    Adds seconds to a given time.

`addMinutes :: Int → ClockTime → ClockTime`

    Adds minutes to a given time.

`addHours :: Int → ClockTime → ClockTime`

    Adds hours to a given time.

`addDays :: Int → ClockTime → ClockTime`

    Adds days to a given time.

`addMonths :: Int → ClockTime → ClockTime`

    Adds months to a given time.

`addYears :: Int → ClockTime → ClockTime`

    Adds years to a given time.

`daysOfMonth :: Int → Int → Int`

    Gets the days of a month in a year.

`validDate :: Int → Int → Int → Bool`

    Is a date consisting of year/month/day valid?

`compareDate :: CalendarTime → CalendarTime → Ordering`

    Compares two dates (don't use it, just for backward compatibility!).

`compareCalendarTime :: CalendarTime → CalendarTime → Ordering`

    Compares two calendar times.

`compareClockTime :: ClockTime → ClockTime → Ordering`

    Compares two clock times.

A.2.41 Library 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.

A.2.42 Library Test.EasyCheck

EasyCheck is a library for automated, property-based testing of Curry programs. The ideas behind EasyCheck are described in this paper The tool `currycheck` automatically executes tests defined with this library. EasyCheck supports the definition of unit tests (also for I/O operations) and property tests parameterized over some arguments.
Note that this module defines the interface of EasyCheck to define properties. The operations to actually execute the tests are contained in the accompanying library `Test.EasyCheckExec`.

Exported types:

`data PropIO`

> Abstract type to represent properties involving IO actions.

Exported constructors:

`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:

`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`

`data Prop`

> Abstract type to represent properties to be checked. Basically, it contains all tests to be executed to check the property.

Exported constructors:

Exported functions:

`returns :: 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 :: 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.

`ioTestOf :: PropIO → Bool → String → IO (Maybe String)`

> Extracts the tests of an I/O property (used by the test runner).

`testsOf :: Prop → [Test]`

> Extracts the tests of a property (used by the test runner).

`result :: Test → Result`

> Extracts the result of a test.

`args :: Test → [String]`

> Extracts the arguments of a test.

`stamp :: Test → [String]`

> Extracts the labels of a test.

`updArgs :: ([String] → [String]) → Test → Test`

> Updates the arguments of a test.

`test :: a → ([a] → Bool) → Prop`

> Constructs a property to be tested from an arbitrary expression (first argument) and a predicate that is applied to the list of non-deterministic values. The given predicate determines whether the constructed property is satisfied or falsified for the given expression.

`(-=-) :: a → a → Prop`

The property x `-=-` y is satisfied if x and y have deterministic values that are equal.

`(<~>) :: a → a → Prop`

The property x `<~>` y is satisfied if the sets of the values of x and y are equal.

`(~>) :: 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.

`(<~) :: 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.

`(<~~>) :: 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 :: (a → Bool) → a`

`solutionOf p` returns (non-deterministically) a solution of predicate p. This operation is useful to test solutions of predicates.

`is :: a → (a → Bool) → Prop`

The property `is x p` is satisfied if x has a deterministic value which satisfies p.

`isAlways :: a → (a → Bool) → Prop`

The property `isAlways x p` is satisfied if all values of x satisfy p.

`isEventually :: 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 :: a → Prop`

The property `failing` x is satisfied if x has no value.

`successful :: a → Prop`

    The property `successful` x is satisfied if x has at least one value.

`deterministic :: a → Prop`

    The property `deterministic` x is satisfied if x has exactly one value.

`(#) :: a → Int → Prop`

    The property `x # n` is satisfied if x has n values.

`(#<) :: a → Int → Prop`

    The property `x #< n` is satisfied if x has less than n values.

`(#>) :: a → Int → Prop`

    The property `x #> n` is satisfied if x has more than n values.

`for :: a → (a → Prop) → Prop`

    The property `for x p` is satisfied if all values y of x satisfy property `p y`.

`forAll :: [a] → (a → Prop) → Prop`

    The property `forAll xs p` is satisfied if all values x of the list `xs` satisfy property `p x`.

`forAllValues :: (a → Prop) → [b] → (b → a) → Prop`

    Only for internal use by the test runner.

`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 :: a → Prop → Prop`

    Assign a label showing the given argument to a property. All labeled tests are counted and shown at the end.

```
collectAs :: 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.

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

>   Extracts values of a search tree according to a given strategy (here: randomized diago-
>   nalization of levels with flattening).

```
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.3 Data Structures and Algorithms

A.3.1 Library Array

Implementation of Arrays with Braun Trees. Conceptually, Braun trees are always infinite. Conse-
quently, there is no test on emptiness.

Exported types:

```
data Array
```


Exported constructors:

Exported functions:

```
emptyErrorArray :: Array a
```

>   Creates an empty array which generates errors for non-initialized indexes.

```
emptyDefaultArray :: (Int → a) → Array a
```

>   Creates an empty array, call given function for non-initialized indexes.

```
(//) :: Array a → [(Int,a)] → Array a
```

>   Inserts a list of entries into an array.

```
update :: Array a → Int → a → Array a
```

>   Inserts a new entry into an array.

```
applyAt :: Array a → Int → (a → a) → Array a
```

>   Applies a function to an element.

```
(!) :: Array a → Int → a
```

Yields the value at a given position.

`listToDefaultArray :: (Int → a) → [a] → Array a`

Creates a default array from a list of entries.

`listToErrorArray :: [a] → Array a`

Creates an error array from a list of entries.

`combine :: (a → b → c) → Array a → Array b → Array c`

combine two arbitrary arrays

`combineSimilar :: (a → a → a) → Array a → Array a → Array a`

the combination of two arrays with identical default function and a combinator which is neutral in the default can be implemented much more efficient

A.3.2 Library Dequeue

An implementation of double-ended queues supporting access at both ends in constant amortized time.

Exported types:

`data Queue`

The datatype of a queue.

Exported constructors:

Exported functions:

`empty :: Queue a`

The empty queue.

`cons :: a → Queue a → Queue a`

Inserts an element at the front of the queue.

`snoc :: a → Queue a → Queue a`

Inserts an element at the end of the queue.

`isEmpty :: Queue a → Bool`

Is the queue empty?

`deqLength :: Queue a → Int`

Returns the number of elements in the queue.

`deqHead :: Queue a` $\rightarrow$ `a`

> The first element of the queue.

`deqTail :: Queue a` $\rightarrow$ `Queue a`

> Removes an element at the front of the queue.

`deqLast :: Queue a` $\rightarrow$ `a`

> The last element of the queue.

`deqInit :: Queue a` $\rightarrow$ `Queue a`

> Removes an element at the end of the queue.

`deqReverse :: Queue a` $\rightarrow$ `Queue a`

> Reverses a double ended queue.

`rotate :: Queue a` $\rightarrow$ `Queue a`

> Moves the first element to the end of the queue.

`matchHead :: Queue a` $\rightarrow$ `Maybe (a,Queue a)`

> Matches the front of a queue. `matchHead q` is equivalent to `if isEmpty q then Nothing else Just (deqHead q, deqTail q)` but more efficient.

`matchLast :: Queue a` $\rightarrow$ `Maybe (a,Queue a)`

> Matches the end of a queue. `matchLast q` is equivalent to `if isEmpty q then Nothing else Just (deqLast q,deqInit q)` but more efficient.

`listToDeq :: [a]` $\rightarrow$ `Queue a`

> Transforms a list to a double ended queue.

`deqToList :: Queue a` $\rightarrow$ `[a]`

> Transforms a double ended queue to a list.

A.3.3 Library FiniteMap

A finite map is an efficient purely functional data structure to store a mapping from keys to values. In order to store the mapping efficiently, an irreflexive(!) order predicate has to be given, i.e., the order predicate `le` should not satisfy (`le x x`) for some key `x`.
Example: To store a mapping from `Int -> String`, the finite map needs a Boolean predicate like (`<`). This version was ported from a corresponding Haskell library

Exported types:

`data FM`

Exported functions:

`emptyFM :: (a → a → Bool) → FM a b`

> The empty finite map.

`unitFM :: (a → a → Bool) → a → b → FM a b`

> Construct a finite map with only a single element.

`listToFM :: (a → a → Bool) → [(a,b)] → FM a b`

> Builts a finite map from given list of tuples (key,element). For multiple occurences of key, the last corresponding element of the list is taken.

`addToFM :: FM a b → a → b → FM a b`

> Throws away any previous binding and stores the new one given.

`addListToFM :: FM a b → [(a,b)] → FM a b`

> Throws away any previous bindings and stores the new ones given. The items are added starting with the first one in the list

`addToFM_C :: (a → a → a) → FM b a → b → a → FM b a`

> Instead of throwing away the old binding, addToFM_C combines the new element with the old one.

`addListToFM_C :: (a → a → a) → FM b a → [(b,a)] → FM b a`

> Combine with a list of tuples (key,element), cf. addToFM_C

`delFromFM :: FM a b → a → FM a b`

> Deletes key from finite map. Deletion doesn't complain if you try to delete something which isn't there

`delListFromFM :: FM a b → [a] → FM a b`

> Deletes a list of keys from finite map. Deletion doesn't complain if you try to delete something which isn't there

`updFM :: FM a b → a → (b → b) → FM a b`

> Applies a function to element bound to given key.

`splitFM :: FM a b → a → Maybe (FM a b,(a,b))`

Combines delFrom and lookup.

`plusFM :: FM a b → FM a b → FM a b`

Efficiently add key/element mappings of two maps into a single one. Bindings in right argument shadow those in the left

`plusFM_C :: (a → a → a) → FM b a → FM b a → FM b a`

Efficiently combine key/element mappings of two maps into a single one, cf. addToFM_C

`minusFM :: FM a b → FM a b → FM a b`

(minusFM a1 a2) deletes from a1 any bindings which are bound in a2

`intersectFM :: FM a b → FM a b → FM a b`

Filters only those keys that are bound in both of the given maps. The elements will be taken from the second map.

`intersectFM_C :: (a → b → c) → FM d a → FM d b → FM d c`

Filters only those keys that are bound in both of the given maps and combines the elements as in addToFM_C.

`foldFM :: (a → b → c → c) → c → FM a b → c`

Folds finite map by given function.

`mapFM :: (a → b → c) → FM a b → FM a c`

Applies a given function on every element in the map.

`filterFM :: (a → b → Bool) → FM a b → FM a b`

Yields a new finite map with only those key/element pairs matching the given predicate.

`sizeFM :: FM a b → Int`

How many elements does given map contain?

`eqFM :: FM a b → FM a b → Bool`

Do two given maps contain the same key/element pairs?

`isEmptyFM :: FM a b → Bool`

Is the given finite map empty?

`elemFM :: a → FM a b → Bool`

Does given map contain given key?

`lookupFM :: FM a b → a → Maybe b`

> Retrieves element bound to given key

`lookupWithDefaultFM :: FM a b → b → a → b`

> Retrieves element bound to given key. If the element is not contained in map, return default value.

`keyOrder :: FM a b → a → a → Bool`

> Retrieves the ordering on which the given finite map is built.

`minFM :: FM a b → Maybe (a,b)`

> Retrieves the smallest key/element pair in the finite map according to the basic key ordering.

`maxFM :: FM a b → Maybe (a,b)`

> Retrieves the greatest key/element pair in the finite map according to the basic key ordering.

`fmToList :: FM a b → [(a,b)]`

> Builds a list of key/element pairs. The list is ordered by the initially given irreflexive order predicate on keys.

`keysFM :: FM a b → [a]`

> Retrieves a list of keys contained in finite map. The list is ordered by the initially given irreflexive order predicate on keys.

`eltsFM :: FM a b → [b]`

> Retrieves a list of elements contained in finite map. The list is ordered by the initially given irreflexive order predicate on keys.

`fmToListPreOrder :: FM a b → [(a,b)]`

> Retrieves list of key/element pairs in preorder of the internal tree. Useful for lists that will be retransformed into a tree or to match any elements regardless of basic order.

`fmSortBy :: (a → a → Bool) → [a] → [a]`

> Sorts a given list by inserting and retrieving from finite map. Duplicates are deleted.

`showFM :: FM a b → String`

> Transforms a finite map into a string. For efficiency reasons, the tree structure is shown which is valid for reading only if one uses the same ordering predicate.

`readFM :: (a → a → Bool) → String → FM a b`

> Transforms a string representation of a finite map into a finite map. One has two provide the same ordering predicate as used in the original finite map.

A.3.4 Library GraphInductive

Library for inductive graphs (port of a Haskell library by Martin Erwig).
In this library, graphs are composed and decomposed in an inductive way.
The key idea is as follows:
A graph is either *empty* or it consists of *node context* and a *graph g'* which are put together by a constructor (`:&`).
This constructor (`:&`), however, is not a constructor in the sense of abstract data type, but more basically a defined constructing funtion.
A *context* is a node together withe the edges to and from this node into the nodes in the graph g'.
For examples of how to use this library, cf. the module `GraphAlgorithms`.

Exported types:

```
type Node = Int
```

> Nodes and edges themselves (in contrast to their labels) are coded as integers.
>
> For both of them, there are variants as labeled, unlabelwd and quasi unlabeled (labeled with ()).
>
> Unlabeled node

```
type LNode a = (Int,a)
```

> Labeled node

```
type UNode = (Int,())
```

> Quasi-unlabeled node

```
type Edge = (Int,Int)
```

> Unlabeled edge

```
type LEdge a = (Int,Int,a)
```

> Labeled edge

```
type UEdge = (Int,Int,())
```

> Quasi-unlabeled edge

```
type Context a b = ([(b,Int)],Int,a,[(b,Int)])
```

> The context of a node is the node itself (along with label) and its adjacent nodes. Thus, a context is a quadrupel, for node n it is of the form (edges to n,node n,n's label,edges from n)

```
type MContext a b = Maybe ([(b,Int)],Int,a,[(b,Int)])
```

> maybe context

```
type Context' a b = ([(b,Int)],a,[(b,Int)])
```

context with edges and node label only, without the node identifier itself

```
type UContext = ([Int],Int,[Int])
```

Unlabeled context.

```
type GDecomp a b = (([(b,Int)],Int,a,[(b,Int)]),Graph a b)
```

A graph decompostion is a context for a node n and the remaining graph without that node.

```
type Decomp a b = (Maybe ([(b,Int)],Int,a,[(b,Int)]),Graph a b)
```

a decomposition with a maybe context

```
type UDecomp a = (Maybe ([Int],Int,[Int]),a)
```

Unlabeled decomposition.

```
type Path = [Int]
```

Unlabeled path

```
type LPath a = [(Int,a)]
```

Labeled path

```
type UPath = [(Int,())]
```

Quasi-unlabeled path

```
type UGr = Graph () ()
```

a graph without any labels

```
data Graph
```

The type variables of Graph are *nodeLabel* and *edgeLabel*. The internal representation of Graph is hidden.

Exported constructors:

Exported functions:

```
(:&) :: ([(a,Int)],Int,b,[(a,Int)]) → Graph b a → Graph b a
```

(:&) takes a node-context and a Graph and yields a new graph.

The according key idea is detailed at the beginning.

nl is the type of the node labels and el the edge labels.

Note that it is an error to induce a context for a node already contained in the graph.

`matchAny :: Graph a b → (([(b,Int)],Int,a,[(b,Int)]),Graph a b)`

decompose a graph into the `Context` for an arbitrarily-chosen `Node` and the remaining `Graph`.

In order to use graphs as abstract data structures, we also need means to decompose a graph. This decompostion should work as much like pattern matching as possible. The normal matching is done by the function matchAny, which takes a graph and yields a graph decompostion.

According to the main idea, matchAny . (:&) should be an identity.

`empty :: Graph a b`

An empty `Graph`.

`mkGraph :: [(Int,a)] → [(Int,Int,b)] → Graph a b`

Create a `Graph` from the list of `LNode`s and `LEdge`s.

`buildGr :: [([(a,Int)],Int,b,[(a,Int)])] → Graph b a`

Build a `Graph` from a list of `Context`s.

`mkUGraph :: [Int] → [(Int,Int)] → Graph () ()`

Build a quasi-unlabeled `Graph` from the list of `Node`s and `Edge`s.

`insNode :: (Int,a) → Graph a b → Graph a b`

Insert a `LNode` into the `Graph`.

`insEdge :: (Int,Int,a) → Graph b a → Graph b a`

Insert a `LEdge` into the `Graph`.

`delNode :: Int → Graph a b → Graph a b`

Remove a `Node` from the `Graph`.

`delEdge :: (Int,Int) → Graph a b → Graph a b`

Remove an `Edge` from the `Graph`.

`insNodes :: [(Int,a)] → Graph a b → Graph a b`

Insert multiple `LNode`s into the `Graph`.

`insEdges :: [(Int,Int,a)] → Graph b a → Graph b a`

Insert multiple `LEdge`s into the `Graph`.

`delNodes :: [Int] → Graph a b → Graph a b`

Remove multiple `Node`s from the `Graph`.

`delEdges :: [(Int,Int)] → Graph a b → Graph a b`

      Remove multiple `Edge`s from the `Graph`.

`isEmpty :: Graph a b → Bool`

      test if the given `Graph` is empty.

`match :: Int → Graph a b → (Maybe ([(b,Int)],Int,a,[(b,Int)]),Graph a b)`

      match is the complement side of (:&), decomposing a `Graph` into the `MContext` found for the given node and the remaining `Graph`.

`noNodes :: Graph a b → Int`

      The number of `Node`s in a `Graph`.

`nodeRange :: Graph a b → (Int,Int)`

      The minimum and maximum `Node` in a `Graph`.

`context :: Graph a b → Int → ([(b,Int)],Int,a,[(b,Int)])`

      Find the context for the given `Node`. In contrast to "match", "context" causes an error if the `Node` is not present in the `Graph`.

`lab :: Graph a b → Int → Maybe a`

      Find the label for a `Node`.

`neighbors :: Graph a b → Int → [Int]`

      Find the neighbors for a `Node`.

`suc :: Graph a b → Int → [Int]`

      Find all `Node`s that have a link from the given `Node`.

`pre :: Graph a b → Int → [Int]`

      Find all `Node`s that link to to the given `Node`.

`lsuc :: Graph a b → Int → [(Int,b)]`

      Find all Nodes and their labels, which are linked from the given `Node`.

`lpre :: Graph a b → Int → [(Int,b)]`

      Find all `Node`s that link to the given `Node` and the label of each link.

`out :: Graph a b → Int → [(Int,Int,b)]`

      Find all outward-bound `LEdge`s for the given `Node`.

`inn :: Graph a b → Int → [(Int,Int,b)]`

Find all inward-bound LEdges for the given Node.

outdeg :: Graph a b → Int → Int

The outward-bound degree of the Node.

indeg :: Graph a b → Int → Int

The inward-bound degree of the Node.

deg :: Graph a b → Int → Int

The degree of the Node.

gelem :: Int → Graph a b → Bool

True if the Node is present in the Graph.

equal :: Graph a b → Graph a b → Bool

graph equality

node' :: ([(a,Int)],Int,b,[(a,Int)]) → Int

The Node in a Context.

lab' :: ([(a,Int)],Int,b,[(a,Int)]) → b

The label in a Context.

labNode' :: ([(a,Int)],Int,b,[(a,Int)]) → (Int,b)

The LNode from a Context.

neighbors' :: ([(a,Int)],Int,b,[(a,Int)]) → [Int]

All Nodes linked to or from in a Context.

suc' :: ([(a,Int)],Int,b,[(a,Int)]) → [Int]

All Nodes linked to in a Context.

pre' :: ([(a,Int)],Int,b,[(a,Int)]) → [Int]

All Nodes linked from in a Context.

lpre' :: ([(a,Int)],Int,b,[(a,Int)]) → [(Int,a)]

All Nodes linked from in a Context, and the label of the links.

lsuc' :: ([(a,Int)],Int,b,[(a,Int)]) → [(Int,a)]

All Nodes linked from in a Context, and the label of the links.

out' :: ([(a,Int)],Int,b,[(a,Int)]) → [(Int,Int,a)]

All outward-directed `LEdge`s in a `Context`.

`inn' :: ([(a,Int)],Int,b,[(a,Int)]) → [(Int,Int,a)]`

All inward-directed `LEdge`s in a `Context`.

`outdeg' :: ([(a,Int)],Int,b,[(a,Int)]) → Int`

The outward degree of a `Context`.

`indeg' :: ([(a,Int)],Int,b,[(a,Int)]) → Int`

The inward degree of a `Context`.

`deg' :: ([(a,Int)],Int,b,[(a,Int)]) → Int`

The degree of a `Context`.

`labNodes :: Graph a b → [(Int,a)]`

A list of all `LNode`s in the `Graph`.

`labEdges :: Graph a b → [(Int,Int,b)]`

A list of all `LEdge`s in the `Graph`.

`nodes :: Graph a b → [Int]`

List all `Node`s in the `Graph`.

`edges :: Graph a b → [(Int,Int)]`

List all `Edge`s in the `Graph`.

`newNodes :: Int → Graph a b → [Int]`

List N available `Node`s, ie `Node`s that are not used in the `Graph`.

`ufold :: (([(a,Int)],Int,b,[(a,Int)]) → c → c) → c → Graph b a → c`

Fold a function over the graph.

`gmap :: (([(a,Int)],Int,b,[(a,Int)]) → ([(c,Int)],Int,d,[(c,Int)])) → Graph b a → Graph d c`

Map a function over the graph.

`nmap :: (a → b) → Graph a c → Graph b c`

Map a function over the `Node` labels in a graph.

`emap :: (a → b) → Graph c a → Graph c b`

Map a function over the `Edge` labels in a graph.

`labUEdges :: [(a,b)] → [(a,b,())]`

add label () to list of edges (node,node)

`labUNodes :: [a] → [(a,())]`

add label () to list of nodes

`showGraph :: Graph a b → String`

Represent Graph as String

A.3.5 Library Random

Library for pseudo-random number generation in Curry.

This library provides operations for generating pseudo-random number sequences. For any given seed, the sequences generated by the operations in this module should be **identical** to the sequences generated by the `java.util.Random package`.

The algorithm is taken from http://en.wikipedia.org/wiki/Random_number_generation. There is an assumption that all operations are implicitly executed mod 2^32 (unsigned 32-bit integers) !!! GHC computes between -2^29 and 2^29-1, thus the sequence is NOT as random as one would like.

```
m_w = <choose-initializer>;    /* must not be zero */
m_z = <choose-initializer>;    /* must not be zero */

uint get_random()
{
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;  /* 32-bit result */
}
```

Exported functions:

`nextInt :: Int → [Int]`

> Returns a sequence of pseudorandom, integer values.

`nextIntRange :: Int → Int → [Int]`

> Returns a pseudorandom sequence of values between 0 (inclusive) and the specified value (exclusive).

`nextBoolean :: Int → [Bool]`

> Returns a pseudorandom sequence of boolean values.

`getRandomSeed :: IO Int`

> Returns a time-dependent integer number as a seed for really random numbers. Should only be used as a seed for pseudorandom number sequence and not as a random number since the precision is limited to milliseconds

`shuffle :: Int → [a] → [a]`

> Computes a random permutation of the given list.

A.3.6 Library RedBlackTree

Library with an implementation of red-black trees:

Serves as the base for both TableRBT and SetRBT All the operations on trees are generic, i.e., one has to provide two explicit order predicates ("`lessThan`" and "`eq`"below) on elements.

Exported types:

`data RedBlackTree`

A red-black tree consists of a tree structure and three order predicates. These predicates generalize the red black tree. They define 1) equality when inserting into the tree

eg for a set eqInsert is (==), for a multiset it is (-> False) for a lookUp-table it is ((==) . fst) 2) equality for looking up values eg for a set eqLookUp is (==), for a multiset it is (==) for a lookUp-table it is ((==) . fst) 3) the (less than) relation for the binary search tree

Exported constructors:

Exported functions:

`empty :: (a → a → Bool) → (a → a → Bool) → (a → a → Bool) → RedBlackTree a`

The three relations are inserted into the structure by function empty. Returns an empty tree, i.e., an empty red-black tree augmented with the order predicates.

`isEmpty :: RedBlackTree a → Bool`

Test on emptyness

`newTreeLike :: RedBlackTree a → RedBlackTree a`

Creates a new empty red black tree from with the same ordering as a give one.

`lookup :: a → RedBlackTree a → Maybe a`

Returns an element if it is contained in a red-black tree.

`update :: a → RedBlackTree a → RedBlackTree a`

Updates/inserts an element into a RedBlackTree.

`delete :: a → RedBlackTree a → RedBlackTree a`

Deletes entry from red black tree.

`tree2list :: RedBlackTree a → [a]`

Transforms a red-black tree into an ordered list of its elements.

`sortBy :: (a → a → Bool) → [a] → [a]`

Generic sort based on insertion into red-black trees. The first argument is the order for the elements.

`setInsertEquivalence :: (a → a → Bool) → RedBlackTree a → RedBlackTree a`

For compatibility with old version only

A.3.7   Library SCC

Computing strongly connected components
Copyright (c) 2000 - 2003, Wolfgang Lux See LICENSE for the full license.
The function `scc` computes the strongly connected components of a list of entities in two steps. First, the list is topologically sorted "downwards" using the *defines* relation. Then the resulting list is sorted "upwards" using the *uses* relation and partitioned into the connected components. Both relations are computed within this module using the bound and free names of each declaration.
In order to avoid useless recomputations, the code in the module first decorates the declarations with their bound and free names and a unique number. The latter is only used to provide a trivial ordering so that the declarations can be used as set elements.

Exported functions:

`scc :: (a → [b]) → (a → [b]) → [a] → [[a]]`

> Computes the strongly connected components of a list of entities. To be flexible, we distinguish the nodes and the entities defined in this node.

A.3.8   Library SearchTree

This library defines a representation of a search space as a tree and various search strategies on this tree. This module implements **strong encapsulation** as discussed in the JFLP'04 paper.

Exported types:

`type Strategy a = SearchTree a → ValueSequence a`

> A search strategy maps a search tree into some sequence of values. Using the abtract type of sequence of values (rather than list of values) enables the use of search strategies for encapsulated search with search trees (strong encapsulation) as well as with set functions (weak encapsulation).

`data SearchTree`

> A search tree is a value, a failure, or a choice between two search trees.

Exported constructors:

- `Value :: a → SearchTree a`

- `Fail :: Int → SearchTree a`

- `Or :: (SearchTree a) → (SearchTree a) → SearchTree a`

185

Exported functions:

getSearchTree :: a → IO (SearchTree a)

    Returns the search tree for some expression.

someSearchTree :: a → SearchTree a

    Internal operation to return the search tree for some expression. Note that this operation is not purely declarative since the ordering in the resulting search tree depends on the ordering of the program rules.

isDefined :: a → Bool

    Returns True iff the argument is defined, i.e., has a value.

showSearchTree :: SearchTree a → String

    Shows the search tree as an intended line structure

searchTreeSize :: SearchTree a → (Int,Int,Int)

    Returns the size (number of Value/Fail/Or nodes) of the search tree.

limitSearchTree :: Int → SearchTree a → SearchTree a

    Limit the depth of a search tree. Branches which a depth larger than the first argument are replace by Fail (-1).

dfsStrategy :: SearchTree a → ValueSequence a

    Depth-first search strategy.

bfsStrategy :: SearchTree a → ValueSequence a

    Breadth-first search strategy.

idsStrategy :: SearchTree a → ValueSequence a

    Iterative-deepening search strategy.

idsStrategyWith :: Int → (Int → Int) → SearchTree a → ValueSequence a

    Parameterized iterative-deepening search strategy. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

diagStrategy :: SearchTree a → ValueSequence a

    Diagonalization search strategy.

allValuesWith :: (SearchTree a → ValueSequence a) → SearchTree a → [a]

    Return all values in a search tree via some given search strategy.

`allValuesDFS :: SearchTree a → [a]`

Return all values in a search tree via depth-first search.

`allValuesBFS :: SearchTree a → [a]`

Return all values in a search tree via breadth-first search.

`allValuesIDS :: SearchTree a → [a]`

Return all values in a search tree via iterative-deepening search.

`allValuesIDSwith :: Int → (Int → Int) → SearchTree a → [a]`

Return all values in a search tree via iterative-deepening search. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

`allValuesDiag :: SearchTree a → [a]`

Return all values in a search tree via diagonalization search strategy.

`getAllValuesWith :: (SearchTree a → ValueSequence a) → a → IO [a]`

Gets all values of an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

`printAllValuesWith :: (SearchTree a → ValueSequence a) → a → IO ()`

Prints all values of an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all printed values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

`printValuesWith :: (SearchTree a → ValueSequence a) → a → IO ()`

Prints the values of an expression w.r.t. a search strategy on demand by the user. Thus, the user must type `<enter></enter>` before another value is computed and printed. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all printed values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results.

`someValue :: a → a`

Returns some value for an expression.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

```
someValueWith :: (SearchTree a → ValueSequence a) → a → a
```

> Returns some value for an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`.

> Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

A.3.9 Library SearchTreeTraversal

Implements additional traversals on search trees.

Exported functions:

```
depthDiag :: SearchTree a → [a]
```

> diagonalized depth first search.

```
rndDepthDiag :: Int → SearchTree a → [a]
```

> randomized variant of diagonalized depth first search.

```
levelDiag :: SearchTree a → [a]
```

> diagonalization of devels.

```
rndLevelDiag :: Int → SearchTree a → [a]
```

> randomized diagonalization of levels.

```
rndLevelDiagFlat :: Int → Int → SearchTree a → [a]
```

> randomized diagonalization of levels with flattening.

A.3.10 Library SetRBT

Library with an implementation of sets as red-black trees.
All the operations on sets are generic, i.e., one has to provide an explicit order predicate ($<$) (less-than) on elements.

Exported types:

```
type SetRBT a = RedBlackTree a
```

Exported functions:

emptySetRBT :: (a → a → Bool) → RedBlackTree a

>   Returns an empty set, i.e., an empty red-black tree augmented with an order predicate.

isEmptySetRBT :: RedBlackTree a → Bool

>   Test for an empty set.

elemRBT :: a → RedBlackTree a → Bool

>   Returns true if an element is contained in a (red-black tree) set.

insertRBT :: a → RedBlackTree a → RedBlackTree a

>   Inserts an element into a set if it is not already there.

insertMultiRBT :: a → RedBlackTree a → RedBlackTree a

>   Inserts an element into a multiset. Thus, the same element can have several occurrences in the multiset.

deleteRBT :: a → RedBlackTree a → RedBlackTree a

>   delete an element from a set. Deletes only a single element from a multi set

setRBT2list :: RedBlackTree a → [a]

>   Transforms a (red-black tree) set into an ordered list of its elements.

unionRBT :: RedBlackTree a → RedBlackTree a → RedBlackTree a

>   Computes the union of two (red-black tree) sets. This is done by inserting all elements of the first set into the second set.

intersectRBT :: RedBlackTree a → RedBlackTree a → RedBlackTree a

>   Computes the intersection of two (red-black tree) sets. This is done by inserting all elements of the first set contained in the second set into a new set, which order is taken from the first set.

sortRBT :: (a → a → Bool) → [a] → [a]

>   Generic sort based on insertion into red-black trees. The first argument is the order for the elements.

A.3.11   Library Sort

A collection of useful functions for sorting and comparing characters, strings, and lists.

189

Exported functions:

`sort :: [a] → [a]`

 The default sorting operation, mergeSort, with standard ordering <=.

`sortBy :: (a → a → Bool) → [a] → [a]`

 The default sorting operation: mergeSort

`sorted :: [a] → Bool`

 `sorted xs` is satisfied if the elements `xs` are in ascending order.

`sortedBy :: (a → a → Bool) → [a] → Bool`

 `sortedBy leq xs` is satisfied if all adjacent elements of the list `xs` satisfy the ordering predicate `leq`.

`permSort :: [a] → [a]`

 Permutation sort with standard ordering <=. Sorts a list by finding a sorted permutation of the input. This is not a usable way to sort a list but it can be used as a specification of other sorting algorithms.

`permSortBy :: (a → a → Bool) → [a] → [a]`

 Permutation sort with ordering as first parameter. Sorts a list by finding a sorted permutation of the input. This is not a usable way to sort a list but it can be used as a specification of other sorting algorithms.

`insertionSort :: [a] → [a]`

 Insertion sort with standard ordering <=. The list is sorted by repeated sorted insertion of the elements into the already sorted part of the list.

`insertionSortBy :: (a → a → Bool) → [a] → [a]`

 Insertion sort with ordering as first parameter. The list is sorted by repeated sorted insertion of the elements into the already sorted part of the list.

`quickSort :: [a] → [a]`

 Quicksort with standard ordering <=. The classical quicksort algorithm on lists.

`quickSortBy :: (a → a → Bool) → [a] → [a]`

 Quicksort with ordering as first parameter. The classical quicksort algorithm on lists.

`mergeSort :: [a] → [a]`

 Bottom-up mergesort with standard ordering <=.

`mergeSortBy :: (a → a → Bool) → [a] → [a]`

Bottom-up mergesort with ordering as first parameter.

`leqList :: (a → a → Bool) → [a] → [a] → Bool`

Less-or-equal on lists.

`cmpList :: (a → a → Ordering) → [a] → [a] → Ordering`

Comparison of lists.

`leqChar :: Char → Char → Bool`

Less-or-equal on characters (deprecated, use Prelude.`<=</code></=</code>`).

`cmpChar :: Char → Char → Ordering`

Comparison of characters (deprecated, use `Prelude.compare`).

`leqCharIgnoreCase :: Char → Char → Bool`

Less-or-equal on characters ignoring case considerations.

`leqString :: String → String → Bool`

Less-or-equal on strings (deprecated, use Prelude.`<=</code></=</code>`).

`cmpString :: String → String → Ordering`

Comparison of strings (deprecated, use `Prelude.compare`).

`leqStringIgnoreCase :: String → String → Bool`

Less-or-equal on strings ignoring case considerations.

`leqLexGerman :: String → String → Bool`

Lexicographical ordering on German strings. Thus, upper/lowercase are not distinguished and Umlauts are sorted as vocals.

A.3.12   Library TableRBT

Library with an implementation of tables as red-black trees:
A table is a finite mapping from keys to values. All the operations on tables are generic, i.e., one has to provide an explicit order predicate ("`cmp`" below) on elements. Each inner node in the red-black tree contains a key-value association.

Exported types:

```
type TableRBT a b = RedBlackTree (a,b)
```

Exported functions:

`emptyTableRBT :: (a → a → Bool) → RedBlackTree (a,b)`

> Returns an empty table, i.e., an empty red-black tree.

`isEmptyTable :: RedBlackTree (a,b) → Bool`

> tests whether a given table is empty

`lookupRBT :: a → RedBlackTree (a,b) → Maybe b`

> Looks up an entry in a table.

`updateRBT :: a → b → RedBlackTree (a,b) → RedBlackTree (a,b)`

> Inserts or updates an element in a table.

`tableRBT2list :: RedBlackTree (a,b) → [(a,b)]`

> Transforms the nodes of red-black tree into a list.

`deleteRBT :: a → RedBlackTree (a,b) → RedBlackTree (a,b)`

A.3.13   Library Traversal

Library to support lightweight generic traversals through tree-structured data. See here[9]   for a description of the library.

Exported types:

`type Traversable a b = a → ([b],[b] → a)`

> A datatype is `Traversable` if it defines a function that can decompose a value into a list of children of the same type and recombine new children to a new value of the original type.

Exported functions:

`noChildren :: a → ([b],[b] → a)`

> Traversal function for constructors without children.

`children :: (a → ([b],[b] → a)) → a → [b]`

> Yields the children of a value.

`replaceChildren :: (a → ([b],[b] → a)) → a → [b] → a`

> Replaces the children of a value.

---

[9]`http://www-ps.informatik.uni-kiel.de/~sebf/projects/traversal.html`

```
mapChildren :: (a → ([b],[b] → a)) → (b → b) → a → a
```

Applies the given function to each child of a value.

```
family :: (a → ([a],[a] → a)) → a → [a]
```

Computes a list of the given value, its children, those children, etc.

```
childFamilies :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → a → [b]
```

Computes a list of family members of the children of a value. The value and its children can have different types.

```
mapFamily :: (a → ([a],[a] → a)) → (a → a) → a → a
```

Applies the given function to each member of the family of a value. Proceeds bottom-up.

```
mapChildFamilies :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (b → b) →
a → a
```

Applies the given function to each member of the families of the children of a value. The value and its children can have different types. Proceeds bottom-up.

```
evalFamily :: (a → ([a],[a] → a)) → (a → Maybe a) → a → a
```

Applies the given function to each member of the family of a value as long as possible. On each member of the family of the result the given function will yield `Nothing`. Proceeds bottom-up.

```
evalChildFamilies :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (b → Maybe
b) → a → a
```

Applies the given function to each member of the families of the children of a value as long as possible. Similar to `evalFamily`.

```
fold :: (a → ([a],[a] → a)) → (a → [b] → b) → a → b
```

Implements a traversal similar to a fold with possible default cases.

```
foldChildren :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (a → [c] → d)
→ (b → [c] → c) → a → d
```

Fold the children and combine the results.

```
replaceChildrenIO :: (a → ([b],[b] → a)) → a → IO [b] → IO a
```

IO version of replaceChildren

```
mapChildrenIO :: (a → ([b],[b] → a)) → (b → IO b) → a → IO a
```

IO version of mapChildren

```
mapFamilyIO :: (a → ([a],[a] → a)) → (a → IO a) → a → IO a
```

IO version of mapFamily

```
mapChildFamiliesIO :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (b → IO
b) → a → IO a
```

IO version of mapChildFamilies

```
evalFamilyIO :: (a → ([a],[a] → a)) → (a → IO (Maybe a)) → a → IO a
```

IO version of evalFamily

```
evalChildFamiliesIO :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (b → IO
(Maybe b)) → a → IO a
```

IO version of evalChildFamilies

A.3.14   Library UnsafeSearchTree

This library defines a representation of a search space as a tree and various search strategies on this
tree. This module implements **strong encapsulation** as discussed in this paper
Warning: In contrast to the SearchTree Module, free variables that are not bound in the encap-
sulated expression remain free! This may lead to non-determinism if such an escaped variable is
bound later via pattern matching.

Exported types:

```
type Strategy a = SearchTree a → ValueSequence a
```


```
data SearchTree
```

A search tree is a value, a failure, or a choice between two search trees.

Exported constructors:

- Value :: a → SearchTree a

- Fail :: Int → SearchTree a

- Or :: (SearchTree a) → (SearchTree a) → SearchTree a

Exported functions:

```
isVar :: a → Bool
```

Tests whether the argument is a free variable This function is only meaningful when
applied to a part of a result of an encapsulated expression if the argument stems from
a `Value` node of a SearchTree

```
identicalVars :: a → a → Bool
```

Tests whether both arguments are identical free variables. This function is only meaningful when applied to parts of a result of an encapsulated expression if the argument stems from a `Value` node of a SearchTree

`varId :: a → Int`

Returns the unique identifier of a free variable, if the argument was not a free variable, otherwise an error is raised. This function is only meaningful when applied to a part of a result of an encapsulated expression if the argument stems from a `Value` node of a SearchTree

`getSearchTree :: a → IO (SearchTree a)`

Returns the search tree for some expression.

`someSearchTree :: a → SearchTree a`

Internal operation to return the search tree for some expression. Note that this operation is not purely declarative since the ordering in the resulting search tree depends on the ordering of the program rules.

`isDefined :: a → Bool`

Returns True iff the argument is is defined, i.e., has a value.

`showSearchTree :: SearchTree a → String`

Shows the search tree as an intended line structure

`searchTreeSize :: SearchTree a → (Int,Int,Int)`

Return the size (number of Value/Fail/Or nodes) of the search tree

`allValuesDFS :: SearchTree a → [a]`

Return all values in a search tree via depth-first search

`dfsStrategy :: SearchTree a → ValueSequence a`

`allValuesBFS :: SearchTree a → [a]`

Return all values in a search tree via breadth-first search

`bfsStrategy :: SearchTree a → ValueSequence a`

`allValuesIDS :: SearchTree a → [a]`

Return all values in a search tree via iterative-deepening search.

`idsStrategy :: SearchTree a → ValueSequence a`

```
allValuesIDSwith :: Int → (Int → Int) → SearchTree a → [a]
```

Return the list of all values in a search tree via iterative-deepening search. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

```
idsStrategyWith :: Int → (Int → Int) → SearchTree a → ValueSequence a
```

Return all values in a search tree via iterative-deepening search. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

```
getAllValuesWith :: (SearchTree a → ValueSequence a) → a → IO [a]
```

Gets all values of an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables.

```
someValue :: a → a
```

Returns some value for an expression.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

```
someValueWith :: (SearchTree a → ValueSequence a) → a → a
```

Returns some value for an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

A.3.15 Library ValueSequence

This library defines a data structure for sequence of values. It is used in search trees (module `SearchTree`) as well as in set functions (module `SetFunctions`). Using sequence of values (rather than standard lists of values) is necessary to get the behavior of set functions w.r.t. finite failures right, as described in the paper

J. Christiansen, M. Hanus, F. Reck, D. Seidel: A Semantics for Weakly Encapsulated Search in Functional Logic Programs Proc. 15th International Conference on Principles and Practice of Declarative Programming (PPDP'13), pp. 49-60, ACM Press, 2013

Exported types:

`data ValueSequence`

> A value sequence is an abstract sequence of values. It also contains failure elements in order to implement the semantics of set functions w.r.t. failures in the intended manner.

Exported constructors:

Exported functions:

`emptyVS :: ValueSequence a`

> An empty sequence of values.

`addVS :: a → ValueSequence a → ValueSequence a`

> Adds a value to a sequence of values.

`failVS :: Int → ValueSequence a`

> Adds a failure to a sequence of values. The argument is the encapsulation level of the failure.

`(|++|) :: ValueSequence a → ValueSequence a → ValueSequence a`

> Concatenates two sequences of values.

`vsToList :: ValueSequence a → [a]`

> Transforms a sequence of values into a list of values.

A.3.16   Library Rewriting.CriticalPairs

Library for representation and computation of critical pairs.

Exported types:

`type CPair a = (Term a,Term a)`

> A critical pair represented as a pair of terms and parameterized over the kind of function symbols, e.g., strings.

Exported functions:

`showCPair :: (a → String) → (Term a,Term a) → String`

> Transforms a critical pair into a string representation.

`showCPairs :: (a → String) → [(Term a,Term a)] → String`

> Transforms a list of critical pairs into a string representation.

197

```
cPairs :: [(Term a,Term a)] → [(Term a,Term a)]
```

> Returns the critical pairs of a term rewriting system.

```
isOrthogonal :: [(Term a,Term a)] → Bool
```

> Checks whether a term rewriting system is orthogonal.

```
isWeakOrthogonal :: [(Term a,Term a)] → Bool
```

> Checks whether a term rewriting system is weak-orthogonal.

A.3.17 Library Rewriting.DefinitionalTree

Library for representation and computation of definitional trees and representation of the reduction strategy phi.

Exported types:

```
data DefTree
```

> Representation of a definitional tree, parameterized over the kind of function symbols, e.g., strings.

> *Exported constructors:*

> - `Leaf :: (Term a,Term a) → DefTree a`
>
>   `Leaf r`
>
>   – The leaf with rule `r`.
>
> - `Branch :: (Term a) → [Int] → [DefTree a] → DefTree a`
>
>   `Branch pat p dts`
>
>   – The branch with pattern `pat`, inductive position `p` and definitional subtrees `dts`.
>
> - `Or :: (Term a) → [DefTree a] → DefTree a`
>
>   `Or pat dts`
>
>   – The or node with pattern `pat` and definitional subtrees `dts`.

Exported functions:

```
dtRoot :: DefTree a → Either Int a
```

> Returns the root symbol (variable or constructor) of a definitional tree.

```
dtPattern :: DefTree a → Term a
```

> Returns the pattern of a definitional tree.

```
hasDefTree :: [DefTree a] → Term a → Bool
```

Checks whether a term has a definitional tree with the same constructor pattern in the given list of definitional trees.

`selectDefTrees :: [DefTree a]` $\to$ `Term a` $\to$ `[DefTree a]`

Returns a list of definitional trees with the same constructor pattern for a term from the given list of definitional trees.

`fromDefTrees :: DefTree a` $\to$ `Int` $\to$ `[DefTree a]` $\to$ `DefTree a`

Returns the definitional tree with the given index from the given list of definitional trees or the provided default definitional tree if the given index is not in the given list of definitional trees.

`idtPositions :: [(Term a,Term a)]` $\to$ `[[Int]]`

Returns a list of all inductive positions in a term rewriting system.

`defTrees :: [(Term a,Term a)]` $\to$ `[DefTree a]`

Returns a list of definitional trees for a term rewriting system.

`defTreesL :: [[(Term a,Term a)]]` $\to$ `[DefTree a]`

Returns a list of definitional trees for a list of term rewriting systems.

`loDefTrees :: [DefTree a]` $\to$ `Term a` $\to$ `Maybe ([Int],[DefTree a])`

Returns the position and the definitional trees from the given list of definitional trees for the leftmost outermost defined constructor in a term or `Nothing` if no such pair exists.

`phiRStrategy :: Int` $\to$ `[(Term a,Term a)]` $\to$ `Term a` $\to$ `[[Int]]`

The reduction strategy phi. It uses the definitional tree with the given index for all constructor terms if possible or at least the first one.

`dotifyDefTree :: (a` $\to$ `String)` $\to$ `DefTree a` $\to$ `String`

Transforms a definitional tree into a graphical representation by using the *DOT graph description language*.

`writeDefTree :: (a` $\to$ `String)` $\to$ `DefTree a` $\to$ `String` $\to$ `IO ()`

Writes the graphical representation of a definitional tree with the *DOT graph description language* to a file with the given filename.

A.3.18 Library Rewriting.Files

Library to read and transform a curry program into an equivalent representation, where every function gets assigned the corresponding term rewriting system and every type has a corresponding type declaration.

Exported types:

```
type TRSData = FM (String,String) [(Term (String,String),Term (String,String))]
```

   Mappings from a function name to the corresponding term rewriting system represented as a finite map from qualified names to term rewriting systems.

```
type TypeData = [CTypeDecl]
```

   Information about types represented as a list of type declarations.

```
type RWData = (FM (String,String) [(Term (String,String),Term
(String,String))],[CTypeDecl])
```

   Representation of term rewriting system data and type data as a pair.

Exported functions:

```
showQName :: (String,String) → String
```

   Transforms a qualified name into a string representation.

```
readQName :: String → (String,String)
```

   Transforms a string into a qualified name.

```
condQName :: (String,String)
```

   Returns the qualified name for an **if-then-else**-constructor.

```
condTRS :: [(Term (String,String),Term (String,String))]
```

   Returns the term rewriting system for an **if-then-else**-function.

```
readCurryProgram :: String → IO (Either String (FM (String,String) [(Term
(String,String),Term (String,String))],[CTypeDecl]))
```

   Tries to read and transform a curry program into an equivalent representation, where every function gets assigned the corresponding term rewriting system and every type has a corresponding type declaration.

```
fromCurryProg :: CurryProg → (FM (String,String) [(Term (String,String),Term
(String,String))],[CTypeDecl])
```

   Transforms an abstract curry program into an equivalent representation, where every function gets assigned the corresponding term rewriting system and every type has a corresponding type declaration.

```
fromFuncDecl :: CFuncDecl → ((String,String),[(Term (String,String),Term
(String,String))])
```

   Transforms an abstract curry function declaration into a pair with function name and corresponding term rewriting system.

```
fromRule :: (String,String) → CRule → ((Term (String,String),Term
(String,String)),[(Term (String,String),Term (String,String))])
```

> Transforms an abstract curry rule for the function with the given name into a pair of a
> rule and a term rewriting system.

```
fromLiteral :: CLiteral → Term (String,String)
```

> Transforms an abstract curry literal into a term.

```
fromPattern :: (String,String) → CPattern → (Term (String,String),FM Int (Term
(String,String)))
```

> Transforms an abstract curry pattern for the function with the given name into a pair
> of a term and a substitution.

```
fromRhs :: (String,String) → CRhs → (Term (String,String),FM Int (Term
(String,String)),[(Term (String,String),Term (String,String))])
```

> Transforms an abstract curry right-hand side of a rule for the function with the given
> name into a tuple of a term, a substitution and a term rewriting system.

```
fromExpr :: (String,String) → CExpr → (Term (String,String),FM Int (Term
(String,String)),[(Term (String,String),Term (String,String))])
```

> Transforms an abstract curry expression for the function with the given name into a
> tuple of a term, a substitution and a term rewriting system.

A.3.19   Library Rewriting.Narrowing

Library for representation and computation of narrowings on first-order terms and representation
of narrowing strategies.

Exported types:

```
type NStrategy a = [(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]
```

> A narrowing strategy represented as a function that takes a term rewriting system and
> a term and returns a list of triples consisting of a position, a rule and a substitution,
> parameterized over the kind of function symbols, e.g., strings.

```
data Narrowing
```

> Representation of a narrowing on first-order terms, parameterized over the kind of func-
> tion symbols, e.g., strings.

> *Exported constructors:*

> - NTerm :: (Term a) → Narrowing a
>
>   NTerm t

– The narrowed term `t`.

- `NStep :: (Term a) → [Int] → (FM Int (Term a)) → (Narrowing a) → Narrowing a`

  `NStep t p sub n`

    – The narrowing of term `t` at position `p` with substitution `sub` to narrowing `n`.

data NarrowingTree

Representation of a narrowing tree for first-order terms, parameterized over the kind of function symbols, e.g., strings.

Exported constructors:

- `NTree :: (Term a) → [([Int],FM Int (Term a),NarrowingTree a)] → NarrowingTree a`

  `NTree t ns`

    – The narrowing of term `t` to a new term with a list of narrowing steps `ns`.

data NOptions

Representation of narrowing options for solving term equations, parameterized over the kind of function symbols, e.g., strings.

Exported constructors:

- `NOptions :: Bool → ([(Term a,Term a)] → Term a → [[Int]]) → NOptions a`

Exported functions:

`normalize :: NOptions a → Bool`

`rStrategy :: NOptions a → [(Term a,Term a)] → Term a → [[Int]]`

`defaultNOptions :: NOptions a`

The default narrowing options.

`showNarrowing :: (a → String) → Narrowing a → String`

Transforms a narrowing into a string representation.

`stdNStrategy :: [(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int (Term a))]`

The standard narrowing strategy.

```
imNStrategy :: [(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]
```

The innermost narrowing strategy.

```
omNStrategy :: [(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]
```

The outermost narrowing strategy.

```
loNStrategy :: [(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]
```

The leftmost outermost narrowing strategy.

```
lazyNStrategy :: [(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]
```

The lazy narrowing strategy.

```
wnNStrategy :: [(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]
```

The weakly needed narrowing strategy.

```
narrowBy :: ([(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int (Term
a))]) → [(Term a,Term a)] → Int → Term a → [(FM Int (Term a),Term a)]
```

Narrows a term with the given strategy and term rewriting system by the given number
of steps.

```
narrowByL :: ([(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int (Term
a))]) → [[(Term a,Term a)]] → Int → Term a → [(FM Int (Term a),Term a)]
```

Narrows a term with the given strategy and list of term rewriting systems by the given
number of steps.

```
narrowingBy :: ([(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]) → [(Term a,Term a)] → Int → Term a → [Narrowing a]
```

Returns a list of narrowings for a term with the given strategy, the given term rewriting
system and the given number of steps.

```
narrowingByL :: ([(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]) → [[(Term a,Term a)]] → Int → Term a → [Narrowing a]
```

Returns a list of narrowings for a term with the given strategy, the given list of term
rewriting systems and the given number of steps.

```
narrowingTreeBy :: ([(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM Int
(Term a))]) → [(Term a,Term a)] → Int → Term a → NarrowingTree a
```

Returns a narrowing tree for a term with the given strategy, the given term rewriting system and the given number of steps.

```
narrowingTreeByL :: ([(Term a,Term a)] → Term a → [([Int],(Term a,Term a),FM
Int (Term a))]) → [[(Term a,Term a)]] → Int → Term a → NarrowingTree a
```

Returns a narrowing tree for a term with the given strategy, the given list of term rewriting systems and the given number of steps.

```
solveEq :: NOptions a → ([(Term a,Term a)] → Term a → [([Int],(Term a,Term
a),FM Int (Term a))]) → [(Term a,Term a)] → Term a → [FM Int (Term a)]
```

Solves a term equation with the given strategy, the given term rewriting system and the given options. The term has to be of the form `TermCons c [l, r]` with c being a constructor like =. The term l and the term r are the left-hand side and the right-hand side of the term equation.

```
solveEqL :: NOptions a → ([(Term a,Term a)] → Term a → [([Int],(Term a,Term
a),FM Int (Term a))]) → [[(Term a,Term a)]] → Term a → [FM Int (Term a)]
```

Solves a term equation with the given strategy, the given list of term rewriting systems and the given options. The term has to be of the form `TermCons c [l, r]` with c being a constructor like =. The term l and the term r are the left-hand side and the right-hand side of the term equation.

```
dotifyNarrowingTree :: (a → String) → NarrowingTree a → String
```

Transforms a narrowing tree into a graphical representation by using the *DOT graph description language*.

```
writeNarrowingTree :: (a → String) → NarrowingTree a → String → IO ()
```

Writes the graphical representation of a narrowing tree with the *DOT graph description language* to a file with the given filename.

A.3.20 Library Rewriting.Position

Library for representation of positions in first-order terms.

Exported types:

```
type Pos = [Int]
```

A position in a term represented as a list of integers greater than zero.

Exported functions:

`showPos :: [Int] → String`

> Transforms a position into a string representation.

`eps :: [Int]`

> The root position of a term.

`above :: [Int] → [Int] → Bool`

> Checks whether the first position is above the second position.

`below :: [Int] → [Int] → Bool`

> Checks whether the first position is below the second position.

`leftOf :: [Int] → [Int] → Bool`

> Checks whether the first position is left from the second position.

`rightOf :: [Int] → [Int] → Bool`

> Checks whether the first position is right from the second position.

`disjoint :: [Int] → [Int] → Bool`

> Checks whether two positions are disjoint.

`positions :: Term a → [[Int]]`

> Returns a list of all positions in a term.

`(.>) :: [Int] → [Int] → [Int]`

> Concatenates two positions.

`(|>) :: Term a → [Int] → Term a`

> Returns the subterm of a term at the given position if the position exists within the term.

`replaceTerm :: Term a → [Int] → Term a → Term a`

> Replaces the subterm of a term at the given position with the given term if the position exists within the term.

A.3.21 Library Rewriting.Rules

Library for representation of rules and term rewriting systems.

Exported types:

`type Rule a = (Term a,Term a)`

A rule represented as a pair of terms and parameterized over the kind of function symbols, e.g., strings.

`type TRS a = [(Term a,Term a)]`

A term rewriting system represented as a list of rules and parameterized over the kind of function symbols, e.g., strings.

Exported functions:

`showRule :: (a → String) → (Term a,Term a) → String`

Transforms a rule into a string representation.

`showTRS :: (a → String) → [(Term a,Term a)] → String`

Transforms a term rewriting system into a string representation.

`rRoot :: (Term a,Term a) → Either Int a`

Returns the root symbol (variable or constructor) of a rule.

`rCons :: (Term a,Term a) → [a]`

Returns a list without duplicates of all constructors in a rule.

`rVars :: (Term a,Term a) → [Int]`

Returns a list without duplicates of all variables in a rule.

`maxVarInRule :: (Term a,Term a) → Maybe Int`

Returns the maximum variable in a rule or `Nothing` if no variable exists.

`minVarInRule :: (Term a,Term a) → Maybe Int`

Returns the minimum variable in a rule or `Nothing` if no variable exists.

`maxVarInTRS :: [(Term a,Term a)] → Maybe Int`

Returns the maximum variable in a term rewriting system or `Nothing` if no variable exists.

`minVarInTRS :: [(Term a,Term a)] → Maybe Int`

Returns the minimum variable in a term rewriting system or `Nothing` if no variable exists.

`renameRuleVars :: Int → (Term a,Term a) → (Term a,Term a)`

Renames the variables in a rule by the given number.

```
renameTRSVars :: Int → [(Term a,Term a)] → [(Term a,Term a)]
```

Renames the variables in every rule of a term rewriting system by the given number.

```
normalizeRule :: (Term a,Term a) → (Term a,Term a)
```

Normalizes a rule by renaming all variables with an increasing order, starting from the minimum possible variable.

```
normalizeTRS :: [(Term a,Term a)] → [(Term a,Term a)]
```

Normalizes all rules in a term rewriting system by renaming all variables with an increasing order, starting from the minimum possible variable.

```
isVariantOf :: (Term a,Term a) → (Term a,Term a) → Bool
```

Checks whether the first rule is a variant of the second rule.

```
isLeftLinear :: [(Term a,Term a)] → Bool
```

Checks whether a term rewriting system is left-linear.

```
isLeftNormal :: [(Term a,Term a)] → Bool
```

Checks whether a term rewriting system is left-normal.

```
isRedex :: [(Term a,Term a)] → Term a → Bool
```

Checks whether a term is reducible with some rule of the given term rewriting system.

```
isPattern :: [(Term a,Term a)] → Term a → Bool
```

Checks whether a term is a pattern, i.e., an root-reducible term where the argaccording to the given term rewriting system.

```
isConsBased :: [(Term a,Term a)] → Bool
```

Checks whether a term rewriting system is constructor-based.

```
isDemandedAt :: Int → (Term a,Term a) → Bool
```

Checks whether the given argument position of a rule is demanded. Returns `True` if the corresponding argument term is a constructor term.

A.3.22 Library Rewriting.Strategy

Library for representation and computation of reductions on first-order terms and representation of reduction strategies.

Exported types:

```
type RStrategy a = [(Term a,Term a)] → Term a → [[Int]]
```

A reduction strategy represented as a function that takes a term rewriting system and a term and returns the redex positions where the term should be reduced, parameterized over the kind of function symbols, e.g., strings.

```
data Reduction
```

Representation of a reduction on first-order terms, parameterized over the kind of function symbols, e.g., strings.

Exported constructors:

- `NormalForm :: (Term a) → Reduction a`

  `NormalForm t`

    – The normal form with term `t`.

- `RStep :: (Term a) → [[Int]] → (Reduction a) → Reduction a`

  `RStep t ps r`

    – The reduction of term `t` at positions `ps` to reduction `r`.

Exported functions:

```
showReduction :: (a → String) → Reduction a → String
```

Transforms a reduction into a string representation.

```
redexes :: [(Term a,Term a)] → Term a → [[Int]]
```

Returns the redex positions of a term according to the given term rewriting system.

```
seqRStrategy :: ([Int] → [Int] → Ordering) → [(Term a,Term a)] → Term a →
[[Int]]
```

Returns a sequential reduction strategy according to the given ordering function.

```
parRStrategy :: ([Int] → [Int] → Ordering) → [(Term a,Term a)] → Term a →
[[Int]]
```

Returns a parallel reduction strategy according to the given ordering function.

```
liRStrategy :: [(Term a,Term a)] → Term a → [[Int]]
```

The leftmost innermost reduction strategy.

```
loRStrategy :: [(Term a,Term a)] → Term a → [[Int]]
```

The leftmost outermost reduction strategy.

```
riRStrategy :: [(Term a,Term a)] → Term a → [[Int]]
```

The rightmost innermost reduction strategy.

```
roRStrategy :: [(Term a,Term a)] → Term a → [[Int]]
```

The rightmost outermost reduction strategy.

```
piRStrategy :: [(Term a,Term a)] → Term a → [[Int]]
```

The parallel innermost reduction strategy.

```
poRStrategy :: [(Term a,Term a)] → Term a → [[Int]]
```

The parallel outermost reduction strategy.

```
reduce :: ([(Term a,Term a)] → Term a → [[Int]]) → [(Term a,Term a)] → Term a
→ Term a
```

Reduces a term with the given strategy and term rewriting system.

```
reduceL :: ([(Term a,Term a)] → Term a → [[Int]]) → [[(Term a,Term a)]] → Term
a → Term a
```

Reduces a term with the given strategy and list of term rewriting systems.

```
reduceBy :: ([(Term a,Term a)] → Term a → [[Int]]) → [(Term a,Term a)] → Int
→ Term a → Term a
```

Reduces a term with the given strategy and term rewriting system by the given number
of steps.

```
reduceByL :: ([(Term a,Term a)] → Term a → [[Int]]) → [[(Term a,Term a)]] →
Int → Term a → Term a
```

Reduces a term with the given strategy and list of term rewriting systems by the given
number of steps.

```
reduceAt :: [(Term a,Term a)] → [Int] → Term a → Term a
```

Reduces a term with the given term rewriting system at the given redex position.

```
reduceAtL :: [(Term a,Term a)] → [[Int]] → Term a → Term a
```

Reduces a term with the given term rewriting system at the given redex positions.

```
reduction :: ([(Term a,Term a)] → Term a → [[Int]]) → [(Term a,Term a)] → Term
a → Reduction a
```

Returns the reduction of a term with the given strategy and term rewriting system.

```
reductionL :: ([(Term a,Term a)] → Term a → [[Int]]) → [[(Term a,Term a)]] →
Term a → Reduction a
```

Returns the reduction of a term with the given strategy and list of term rewriting systems.

```
reductionBy :: ([(Term a,Term a)] → Term a → [[Int]]) → [(Term a,Term a)] →
Int → Term a → Reduction a
```

Returns the reduction of a term with the given strategy, the given term rewriting system and the given number of steps.

```
reductionByL :: ([(Term a,Term a)] → Term a → [[Int]]) → [[(Term a,Term a)]] →
Int → Term a → Reduction a
```

Returns the reduction of a term with the given strategy, the given list of term rewriting systems and the given number of steps.

A.3.23 Library Rewriting.Substitution

Library for representation of substitutions on first-order terms.

Exported types:

```
type Subst a = FM Int (Term a)
```

A substitution represented as a finite map from variables to terms and parameterized over the kind of function symbols, e.g., strings.

Exported functions:

```
showSubst :: (a → String) → FM Int (Term a) → String
```

Transforms a substitution into a string representation.

```
emptySubst :: FM Int (Term a)
```

The empty substitution.

```
extendSubst :: FM Int (Term a) → Int → Term a → FM Int (Term a)
```

Extends a substitution with a new mapping from the given variable to the given term. An already existing mapping with the same variable will be thrown away.

```
listToSubst :: [(Int,Term a)] → FM Int (Term a)
```

Returns a substitution that contains all the mappings from the given list. For multiple mappings with the same variable, the last corresponding mapping of the given list is taken.

```
lookupSubst :: FM Int (Term a) → Int → Maybe (Term a)
```

Returns the term mapped to the given variable in a substitution or `Nothing` if no such mapping exists.

```
applySubst :: FM Int (Term a) → Term a → Term a
```

Applies a substitution to the given term.

```
applySubstEq :: FM Int (Term a) → (Term a,Term a) → (Term a,Term a)
```

Applies a substitution to both sides of the given term equation.

```
applySubstEqs :: FM Int (Term a) → [(Term a,Term a)] → [(Term a,Term a)]
```

Applies a substitution to every term equation in the given list.

```
restrictSubst :: FM Int (Term a) → [Int] → FM Int (Term a)
```

Returns a new substitution with only those mappings from the given substitution whose variable is in the given list of variables.

```
composeSubst :: FM Int (Term a) → FM Int (Term a) → FM Int (Term a)
```

Composes the first substitution `phi` with the second substitution `sigma`. The resulting substitution `sub` fulfills the property `sub(t) = phi(sigma(t))` for a term `t`. Mappings in the first substitution shadow those in the second.

A.3.24 Library Rewriting.Term

Library for representation of first-order terms.
This library is the basis of other libraries for the manipulation of first-order terms, e.g., unification of terms. Therefore, this library also defines other structures, like term equations.

Exported types:

```
type VarIdx = Int
```

A variable represented as an integer greater than or equal to zero.

```
type TermEq a = (Term a,Term a)
```

A term equation represented as a pair of terms and parameterized over the kind of function symbols, e.g., strings.

```
type TermEqs a = [(Term a,Term a)]
```

Multiple term equations represented as a list of term equations and parameterized over the kind of function symbols, e.g., strings.

```
data Term
```

Representation of a first-order term, parameterized over the kind of function symbols, e.g., strings.

Exported constructors:

- `TermVar :: Int → Term a`

  `TermVar v`

    – The variable term with variable `v`.

- `TermCons :: a → [Term a] → Term a`

  `TermCons c ts`

    – The constructor term with constructor `c` and argument terms `ts`.

Exported functions:

`showVarIdx :: Int → String`

    Transforms a variable into a string representation.

`showTerm :: (a → String) → Term a → String`

    Transforms a term into a string representation.

`showTermEq :: (a → String) → (Term a,Term a) → String`

    Transforms a term equation into a string representation.

`showTermEqs :: (a → String) → [(Term a,Term a)] → String`

    Transforms a list of term equations into a string representation.

`tConst :: a → Term a`

    Returns a term with the given constructor and no argument terms.

`tOp :: Term a → a → Term a → Term a`

    Returns an infix operator term with the given constructor and the given left and right
    argument term.

`tRoot :: Term a → Either Int a`

    Returns the root symbol (variable or constructor) of a term.

`tCons :: Term a → [a]`

    Returns a list without duplicates of all constructors in a term.

`tConsAll :: Term a → [a]`

    Returns a list of all constructors in a term. The resulting list may contain duplicates.

`tVars :: Term a → [Int]`

    Returns a list without duplicates of all variables in a term.

`tVarsAll :: Term a → [Int]`

Returns a list of all variables in a term. The resulting list may contain duplicates.

`isConsTerm :: Term a → Bool`

    Checks whether a term is a constructor term.

`isVarTerm :: Term a → Bool`

    Checks whether a term is a variable term.

`isGround :: Term a → Bool`

    Checks whether a term is a ground term (contains no variables).

`isLinear :: Term a → Bool`

    Checks whether a term is linear (contains no variable more than once).

`isNormal :: Term a → Bool`

    Checks whether a term is normal (behind a variable is not a constructor).

`maxVarInTerm :: Term a → Maybe Int`

    Returns the maximum variable in a term or `Nothing` if no variable exists.

`minVarInTerm :: Term a → Maybe Int`

    Returns the minimum variable in a term or `Nothing` if no variable exists.

`normalizeTerm :: Term a → Term a`

    Normalizes a term by renaming all variables with an increasing order, starting from the minimum possible variable.

`renameTermVars :: Int → Term a → Term a`

    Renames the variables in a term by the given number.

`mapTerm :: (a → b) → Term a → Term b`

    Transforms a term by applying a transformation on all constructors.

`eqConsPattern :: Term a → Term a → Bool`

    Checks whether the constructor pattern of the first term is equal to the constructor pattern of the second term. Returns `True` if both terms have the same constructor and the same arity.

A.3.25 Library Rewriting.Unification

Library for representation of unification on first-order terms.
This library implements a unification algorithm using reference tables.

Exported functions:

`unify :: [(Term a,Term a)] → Either (UnificationError a) (FM Int (Term a))`

Unifies a list of term equations. Returns either a unification error or a substitution.

`unifiable :: [(Term a,Term a)] → Bool`

Checks whether a list of term equations can be unified.

A.3.26 Library Rewriting.UnificationSpec

Library for specifying the unification on first-order terms.
This library implements a general unification algorithm. Because the algorithm is easy to understand, but rather slow, it serves as a specification for more elaborate implementations.

Exported types:

`data UnificationError`

Representation of a unification error, parameterized over the kind of function symbols, e.g., strings.

Exported constructors:

- `Clash :: (Term a) → (Term a) → UnificationError a`

  `Clash t1 t2`

  – The constructor term `t1` is supposed to be equal to the constructor term `t2` but has a different constructor.

- `OccurCheck :: Int → (Term a) → UnificationError a`

  `OccurCheck v t`

  – The variable `v` is supposed to be equal to the term `t` in which it occurs as a subterm.

Exported functions:

`showUnificationError :: (a → String) → UnificationError a → String`

Transforms a unification error into a string representation.

`unify :: [(Term a,Term a)] → Either (UnificationError a) (FM Int (Term a))`

Unifies a list of term equations. Returns either a unification error or a substitution.

`unifiable :: [(Term a,Term a)] → Bool`

Checks whether a list of term equations can be unified.

A.4 Libraries for Database Access and Manipulation

A.4.1 Library KeyDatabaseSQLite

This module provides a general interface for databases (persistent predicates) where each entry consists of a key and an info part. The key is an integer and the info is arbitrary. All functions are parameterized with a dynamic predicate that takes an integer key as a first parameter.

This module reimplements the interface of the module `KeyDatabase` based on the SQLite database engine. In order to use it you need to have `sqlite3` in your `PATH` environment variable or adjust the value of the constant `path<code>to</code>sqlite3`.

Programs that use the `KeyDatabase` module can be adjusted to use this module instead by replacing the imports of `Dynamic`, `Database`, and `KeyDatabase` with this module and changing the declarations of database predicates to use the function `persistentSQLite` instead of `dynamic` or `persistent`. This module redefines the types `Dynamic`, `Query`, and `Transaction` and although both implementations can be used in the same program (by importing modules qualified) they cannot be mixed.

Compared with the interface of `KeyDatabase`, this module lacks definitions for `index`, `sortByIndex`, `groupByIndex`, and `runTNA` and adds the functions `deleteDBEntries` and `closeDBHandles`.

Exported types:

```
type Key = Int
```

```
type KeyPred a = Int → a → Dynamic
```

```
data Query
```

> Queries can read but not write to the database.

   Exported constructors:

```
data Transaction
```

> Transactions can modify the database and are executed atomically.

   Exported constructors:

```
data Dynamic
```

> Result type of database predicates.

   Exported constructors:

```
data ColVal
```

> Abstract type for value restrictions

Exported constructors:

data TError

The type of errors that might occur during a transaction.

Exported constructors:

- TError :: TErrorKind → String → TError

data TErrorKind

The various kinds of transaction errors.

Exported constructors:

- KeyNotExistsError :: TErrorKind

- NoRelationshipError :: TErrorKind

- DuplicateKeyError :: TErrorKind

- KeyRequiredError :: TErrorKind

- UniqueError :: TErrorKind

- MinError :: TErrorKind

- MaxError :: TErrorKind

- UserDefinedError :: TErrorKind

- ExecutionError :: TErrorKind

Exported functions:

runQ :: Query a → IO a

Runs a database query in the IO monad.

transformQ :: (a → b) → Query a → Query b

Applies a function to the result of a database query.

runT :: Transaction a → IO (Either a TError)

Runs a transaction atomically in the IO monad.

Transactions are *immediate*, which means that locks are acquired on all databases as soon as the transaction is started. After one transaction is started, no other database connection will be able to write to the database or start a transaction. Other connections *can* read the database during a transaction of another process.

The choice to use immediate rather than deferred transactions is conservative. It might also be possible to allow multiple simultaneous transactions that lock tables on the first database access (which is the default in SQLite). However this leads to unpredictable order in which locks are taken when multiple databases are involved. The current implementation fixes the locking order by sorting databases by their name and locking them in order immediately when a transaction begins.

More information on [10] _transaction.html">transactions in SQLite is available online.

`runJustT :: Transaction a → IO a`

Executes a possibly composed transaction on the current state of dynamic predicates as a single transaction. Similar to `runT` but a run-time error is raised if the execution of the transaction fails.

`getDB :: Query a → Transaction a`

Lifts a database query to the transaction type such that it can be composed with other transactions. Run-time errors that occur during the execution of the given query are transformed into transaction errors.

`returnT :: a → Transaction a`

Returns the given value in a transaction that does not access the database.

`doneT :: Transaction ()`

Returns the unit value in a transaction that does not access the database. Useful to ignore results when composing transactions.

`errorT :: TError → Transaction a`

Aborts a transaction with an error.

`failT :: String → Transaction a`

Aborts a transaction with a user-defined error message.

`(|>>=) :: Transaction a → (a → Transaction b) → Transaction b`

Combines two transactions into a single transaction that executes both in sequence. The first transaction is executed, its result passed to the function which computes the second transaction, which is then executed to compute the final result.

If the first transaction is aborted with an error, the second transaction is not executed.

`(|>>) :: Transaction a → Transaction b → Transaction b`

Combines two transactions to execute them in sequence. The result of the first transaction is ignored.

---

[10]http://sqlite.org/lang

`sequenceT :: [Transaction a] → Transaction [a]`

Executes a list of transactions sequentially and computes a list of all results.

`sequenceT_ :: [Transaction a] → Transaction ()`

Executes a list of transactions sequentially, ignoring their results.

`mapT :: (a → Transaction b) → [a] → Transaction [b]`

Applies a function that yields transactions to all elements of a list, executes the transaction sequentially, and collects their results.

`mapT_ :: (a → Transaction b) → [a] → Transaction ()`

Applies a function that yields transactions to all elements of a list, executes the transactions sequentially, and ignores their results.

`persistentSQLite :: String → String → [String] → Int → a → Dynamic`

This function is used instead of `dynamic` or `persistent` to declare predicates whose facts are stored in an SQLite database.

If the provided database or the table do not exist they are created automatically when the declared predicate is accessed for the first time.

Multiple column names can be provided if the second argument of the predicate is a tuple with a matching arity. Other record types are not supported. If no column names are provided a table with a single column called `info` is created. Columns of name *rowid* are not supported and lead to a run-time error.

`existsDBKey :: (Int → a → Dynamic) → Int → Query Bool`

Checks whether the predicate has an entry with the given key.

`allDBKeys :: (Int → a → Dynamic) → Query [Int]`

Returns a list of all stored keys. Do not use this function unless the database is small.

`allDBInfos :: (Int → a → Dynamic) → Query [a]`

Returns a list of all info parts of stored entries. Do not use this function unless the database is small.

`allDBKeyInfos :: (Int → a → Dynamic) → Query [(Int,a)]`

Returns a list of all stored entries. Do not use this function unless the database is small.

`(@=) :: Int → a → ColVal`

Constructs a value restriction for the column given as first argument

`someDBKeys :: (Int → a → Dynamic) → [ColVal] → Query [Int]`

Returns a list of those stored keys where the corresponding info part matches the gioven value restriction. Safe to use even on large databases if the number of results is small.

`someDBInfos :: (Int → a → Dynamic) → [ColVal] → Query [a]`

Returns a list of those info parts of stored entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

`someDBKeyInfos :: (Int → a → Dynamic) → [ColVal] → Query [(Int,a)]`

Returns a list of those entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

`someDBKeyProjections :: (Int → a → Dynamic) → [Int] → [ColVal] → Query [(Int,b)]`

Returns a list of column projections on those entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

`getDBInfo :: (Int → a → Dynamic) → Int → Query (Maybe a)`

Queries the information stored under the given key. Yields `Nothing` if the given key is not present.

`getDBInfos :: (Int → a → Dynamic) → [Int] → Query (Maybe [a])`

Queries the information stored under the given keys. Yields `Nothing` if a given key is not present.

`deleteDBEntry :: (Int → a → Dynamic) → Int → Transaction ()`

Deletes the information stored under the given key. If the given key does not exist this transaction is silently ignored and no error is raised.

`deleteDBEntries :: (Int → a → Dynamic) → [Int] → Transaction ()`

Deletes the information stored under the given keys. No error is raised if (some of) the keys do not exist.

`updateDBEntry :: (Int → a → Dynamic) → Int → a → Transaction ()`

Updates the information stored under the given key. The transaction is aborted with a `KeyNotExistsError` if the given key is not present in the database.

`newDBEntry :: (Int → a → Dynamic) → a → Transaction Int`

Stores new information in the database and yields the newly generated key.

`newDBKeyEntry :: (Int → a → Dynamic) → Int → a → Transaction ()`

Stores a new entry in the database under a given key. The transaction fails if the key already exists.

```
cleanDB :: (Int → a → Dynamic) → Transaction ()
```

Deletes all entries from the database associated with a predicate.

```
closeDBHandles :: IO ()
```

Closes all database connections. Should be called when no more database access will be necessary.

```
showTError :: TError → String
```

Transforms a transaction error into a string.

A.4.2 Library Database.CDBI.Connection

This module defines basis data types and functions for accessing database systems using SQL. Currently, only SQLite3 is supported, but this is easy to extend. It also provides execution of SQL-Queries with types. Allowed datatypes for these queries are defined and the conversion to standard SQL-Queries is provided.

Exported types:

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

The result of SQL-related actions. It is either a `DBError` or some value.

```
type DBAction a = Connection → IO (Either DBError a)
```

A DBAction takes a connection and returns an `IO (SQLResult a)`.

```
data DBError
```

`DBErrors` are composed of an `DBErrorKind` and a `String` describing the error more explicitly.

Exported constructors:

- `DBError :: DBErrorKind → String → DBError`

```
data DBErrorKind
```

The different kinds of errors.

Exported constructors:

- `TableDoesNotExist :: DBErrorKind`

- `ParameterError :: DBErrorKind`

- `ConstraintViolation :: DBErrorKind`

- `SyntaxError :: DBErrorKind`

220

- NoLineError :: DBErrorKind

- LockedDBError :: DBErrorKind

- UnknownError :: DBErrorKind

data SQLValue

Data type for SQL values, used during the communication with the database.

Exported constructors:

- SQLString :: String → SQLValue

- SQLInt :: Int → SQLValue

- SQLFloat :: Float → SQLValue

- SQLChar :: Char → SQLValue

- SQLBool :: Bool → SQLValue

- SQLDate :: ClockTime → SQLValue

- SQLNull :: SQLValue

data SQLType

Type identifiers for SQLValues, necessary to determine the type of the value a column should be converted to.

Exported constructors:

- SQLTypeString :: SQLType

- SQLTypeInt :: SQLType

- SQLTypeFloat :: SQLType

- SQLTypeChar :: SQLType

- SQLTypeBool :: SQLType

- SQLTypeDate :: SQLType

data Connection

Data type for database connections. Currently, only connections to a SQLite3 database are supported, but other types of connections could easily be added. List of functions that would need to be implemented: A function to connect to the database, disconnect, writeConnection readRawConnection, parseLines, begin, commit, rollback and getColumnNames

Exported constructors:

- SQLiteConnection :: Handle → Connection

Exported functions:

`fromSQLResult :: Either DBError a → a`

> Gets the value of an `SQLResult`. If there is no result value but a database error, the error is raised.

`printSQLResults :: Either DBError [a] → IO ()`

> Print an `SQLResult` list, i.e., print either the `DBError` or the list of result elements.

`runInTransaction :: (Connection → IO (Either DBError a)) → Connection → IO (Either DBError a)`

> Run a `DBAction` as a transaction. In case of an `Error` it will rollback all changes, otherwise the changes are committed.

`(>+=) :: (Connection → IO (Either DBError a)) → (a → Connection → IO (Either DBError b)) → Connection → IO (Either DBError b)`

> Connect two `DBAction`s. When executed this function will execute the first `DBAction` and then execute the second applied to the first result An `Error` will stop either action.

`(>+) :: (Connection → IO (Either DBError a)) → (Connection → IO (Either DBError b)) → Connection → IO (Either DBError b)`

> Connect two DBActions, but ignore the result of the first.

`fail :: DBError → Connection → IO (Either DBError a)`

> Failing action.

`ok :: a → Connection → IO (Either DBError a)`

> Successful action.

`select :: String → [SQLValue] → [SQLType] → Connection → IO (Either DBError [[SQLValue]])`

> execute a query where the result of the execution is returned

`execute :: String → [SQLValue] → Connection → IO (Either DBError ())`

> execute a query without a result

`executeMultipleTimes :: String → [[SQLValue]] → Connection → IO (Either DBError ())`

> execute a query multiple times with different SQLValues without a result

`connectSQLite :: String → IO Connection`

> Connect to a SQLite Database

`disconnect :: Connection → IO ()`

> Disconnect from a database.

`begin :: Connection → IO ()`

> Begin a Transaction.

`commit :: Connection → IO ()`

> Commit a Transaction.

`rollback :: Connection → IO ()`

> Rollback a Transaction.

`runWithDB :: String → (Connection → IO a) → IO a`

> Executes an action dependent on a connection on a database by connecting to the datebase. The connection will be kept open and re-used for the next action to this database.

`executeRaw :: String → [String] → Connection → IO (Either DBError [[String]])`

> Execute a SQL statement. The statement may contain ? placeholders and a list of parameters which should be inserted at the respective positions. The result is a list of list of strings where every single list represents a row of the result.

`getColumnNames :: String → Connection → IO (Either DBError [String])`

> Returns a list with the names of every column in a table The parameter is the name of the table and a connection

`valueToString :: SQLValue → String`

A.4.3   Library Database.CDBI.Criteria

This module provides datatypes, constructor functions and translation functions to specify SQL criteria including options(group-by, having, order-by)

Exported types:

`type CColumn = Column ()`

> Type for columns used inside a constraint.

`type CValue = Value ()`

> Type for values used inside a constraint.

`data Criteria`

Criterias for queries that can have a constraint and a group-by clause

Exported constructors:

- Criteria :: Constraint → (Maybe GroupBy) → Criteria

data Specifier

specifier for queries

Exported constructors:

- Distinct :: Specifier

- All :: Specifier

data Option

datatype to represent order-by statement

Exported constructors:

data GroupBy

datatype to represent group-by statement

Exported constructors:

data Condition

datatype for conditions inside a having-clause

Exported constructors:

- Con :: Constraint → Condition

- Fun :: String → Specifier → Constraint → Condition

- HAnd :: [Condition] → Condition

- HOr :: [Condition] → Condition

- Neg :: Condition → Condition

data Value

A datatype to compare values. Can be either a SQLValue or a Column with an additional Integer (rename-number). The Integer is for dealing with renamed tables in queries (i.e. Students as 1Students). If the Integer n is 0 the column will be named as usual ("Table"."Column"), otherwise it will be named "nTable"."Column" in the query This is for being able to do complex "where exists" constraints

224

Exported constructors:

- `Val :: SQLValue → Value a`

- `Col :: (Column a) → Int → Value a`

data ColVal

A datatype thats a combination between a Column and a Value (Needed for update queries)

Exported constructors:

- `ColVal :: (Column ()) → (Value ()) → ColVal`

data Constraint

Constraints for queries Every constructor with at least one value has a function as a constructor and only that function will be exported to assure type-safety Most of these are just like the Sql-where-commands Exists needs the table-name, an integer and maybe a constraint (where exists (select *from table where constraint)) The integer n will rename the table if it has a different value than 0 (where exists (select *from table as ntable where...))*

Exported constructors:

- `IsNull :: (Value ()) → Constraint`

- `IsNotNull :: (Value ()) → Constraint`

- `BinaryRel :: RelOp → (Value ()) → (Value ()) → Constraint`

- `Between :: (Value ()) → (Value ()) → (Value ()) → Constraint`

- `IsIn :: (Value ()) → [Value ()] → Constraint`

- `Not :: Constraint → Constraint`

- `And :: [Constraint] → Constraint`

- `Or :: [Constraint] → Constraint`

- `Exists :: String → Int → Constraint → Constraint`

- `None :: Constraint`

Exported functions:

emptyCriteria :: Criteria

    An empty criteria

int :: Int → Value Int

    Constructor for a Value Val of type Int

float :: Float → Value Float

    Constructor for a Value Val of type Float

char :: Char → Value Char

    Constructor for a Value Val of type Char

string :: String → Value String

    Constructor for a Value Val of type String

bool :: Bool → Value Bool

    Constructor for a Value Val of type Bool

date :: ClockTime → Value ClockTime

    Constructor for a Value Val of type ClockTime

idVal :: Int → Value a

    Constructor for Values of ID-types Should just be used internally!

col :: Column a → Value a

    Constructor for a Value Col without a rename-number

colNum :: Column a → Int → Value a

    Constructor for a Value Col with a rename-number

colVal :: Column a → Value a → ColVal

    A constructor for ColVal needed for typesafety

colValAlt :: String → String → SQLValue → ColVal

    Alternative ColVal constructor without typesafety

isNull :: Value a → Constraint

    IsNull construnctor

isNotNull :: Value a → Constraint

IsNotNull construnctor

`equal :: Value a → Value a → Constraint`

Equal construnctor

`(.=.) :: Value a → Value a → Constraint`

Infix Equal

`notEqual :: Value a → Value a → Constraint`

NotEqual construnctor

`(./=.) :: Value a → Value a → Constraint`

Infix NotEqual

`greaterThan :: Value a → Value a → Constraint`

GreatherThan construnctor

`(.>.) :: Value a → Value a → Constraint`

Infix GreaterThan

`lessThan :: Value a → Value a → Constraint`

LessThan construnctor

`(.<.) :: Value a → Value a → Constraint`

Infix LessThan

`greaterThanEqual :: Value a → Value a → Constraint`

GreaterThanEqual construnctor

`(.>=.) :: Value a → Value a → Constraint`

Infix GreaterThanEqual

`lessThanEqual :: Value a → Value a → Constraint`

LessThanEqual construnctor

`(.<=.) :: Value a → Value a → Constraint`

Infix LessThanEqual

`like :: Value a → Value a → Constraint`

Like construnctor

`(.~.) :: Value a → Value a → Constraint`

Infix Like

```
between :: Value a → Value a → Value a → Constraint
```

Between construnctor

```
isIn :: Value a → [Value a] → Constraint
```

IsIn construnctor

```
(.<->.) :: Value a → [Value a] → Constraint
```

Infix IsIn

```
ascOrder :: Value a → Option
```

Constructor for the option: Ascending Order by Column

```
descOrder :: Value a → Option
```

Constructor for the option: Descending Order by Column

```
groupBy :: Value a → GroupByTail → GroupBy
```

```
groupByCol :: Value a → GroupByTail → GroupByTail
```

Constructor to specifiy more than one column for group-by

```
having :: Condition → GroupByTail
```

```
noHave :: GroupByTail
```

Constructor for empty having-Clause

```
condition :: Constraint → Condition
```

```
sumIntCol :: Specifier → Value Int → Value Int → (Value () → Value () →
Constraint) → Condition
```

having-clauses.

```
sumFloatCol :: Specifier → Value Float → Value Float → (Value () → Value () →
Constraint) → Condition
```

Constructor for aggregation function sum for columns of type float in having-clauses.

```
avgIntCol :: Specifier → Value Int → Value Float → (Value () → Value () →
Constraint) → Condition
```

Constructor for aggregation function avg for columns of type Int in having-clauses.

```
avgFloatCol :: Specifier → Value Float → Value Float → (Value () → Value () →
Constraint) → Condition
```

Constructor for aggregation function avg for columns of type float in having-clauses.

```
countCol :: Specifier → Value a → Value Int → (Value () → Value () →
Constraint) → Condition
```

```
minCol :: Specifier → Value a → Value a → (Value () → Value () → Constraint)
→ Condition
```

Constructor for aggregation function min in having-clauses.

```
maxCol :: Specifier → Value a → Value a → (Value () → Value () → Constraint)
→ Condition
```

Constructor for aggregation function max in having-clauses.

```
toCColumn :: Column a → Column ()
```

```
toCValue :: Value a → Value ()
```

```
trCriteria :: Criteria → String
```

```
trOption :: [Option] → String
```

```
trCondition :: Condition → String
```

```
trConstraint :: Constraint → String
```

```
trValue :: Value a → String
```

```
trColumn :: String → Int → String
```

```
trSpecifier :: Specifier → String
```

A.4.4 Library Database.CDBI.Description

This module contains basic datatypes and operations to represent a relational data model in a type-safe manner. This representation is used by the library `Database.CDBI.ER` to provide type safety when working with relational databases. The tool `erd2cdbi` generates from an entity-relationship model a Curry program that represents all entities and relationships by the use of this module.

Exported types:

`type Table = String`

> A type representing tablenames

`data EntityDescription`

> The datatype EntityDescription is a description of a database entity type including the name, the types the entity consists of, a function transforming an instance of this entity to a list of SQLValues, a second function doing the same but converting the key value always to SQLNull to ensure that keys are auto incrementing and a function transforming a list of SQLValues to an instance of this entity

> *Exported constructors:*

> - `ED :: String → [SQLType] → (a → [SQLValue]) → (a → [SQLValue]) →`
>   `([SQLValue] → a) → EntityDescription a`

`data CombinedDescription`

> Entity-types can be combined (For Example Student and Lecture could be combined to Data StuLec = StuLec Student Lecture). If a description for this new type is written CDBI can look up that type in the database The description is a list of Tuples consisting of a String (The name of the entity type that will be combined), a "rename-number" n which will rename the table to "table as ntable" and a list of SQLTypes (The types that make up that entity type). Furthermore there has to be a function that transform a list of SQLValues into this combined type, and two functions that transform the combined type into a list of SQLValues, the first one for updates, the second one for insertion. The list of sqlvalues needs to match what is returned by the database.

> *Exported constructors:*

> - `CD :: [(String,Int,[SQLType])] → ([SQLValue] → a) → (a → [[SQLValue]]) →`
>   `(a → [[SQLValue]]) → CombinedDescription a`

`data Column`

> A datatype representing column names. The first string is the simple name of the column (for example the column Name of the row Student). The second string is the name of the column combined with the name of the row (for example Student.Name). These names should always be in quotes (for example "Student"."Name") so no errors emerge (the name "Group" for example would result in errors if not in quotes). Has a phantom-type for the value the column represents.

Exported constructors:

- Column :: String → String → Column a

data ColumnDescription

Datatype representing columns for selection. This datatype has to be distinguished from type Column which is just for definition of conditions. The type definition consists of the complete name (including tablename), the SQLType of the column and two functions for the mapping from SQLValue into the resulttype and the other way around

Exported constructors:

- ColDesc :: String → SQLType → (a → SQLValue) → (SQLValue → a) → ColumnDescription a

Exported functions:

combineDescriptions :: EntityDescription a → Int → EntityDescription b → Int → (a → b → c) → (c → (a,b)) → CombinedDescription c

A constructor for CombinedDescription.

addDescription :: EntityDescription a → Int → (a → b → b) → (b → a) → CombinedDescription b → CombinedDescription b

Adds another ED to an already existing CD.

getTable :: EntityDescription a → String

getTypes :: EntityDescription a → [SQLType]

getToValues :: EntityDescription a → a → [SQLValue]

getToInsertValues :: EntityDescription a → a → [SQLValue]

getToEntity :: EntityDescription a → [SQLValue] → a

getColumnSimple :: Column a → String

getColumnFull :: Column a → String

```
getColumnName :: ColumnDescription a → String


getColumnTableName :: ColumnDescription a → String


getColumnTyp :: ColumnDescription a → SQLType


getColumnValueBuilder :: ColumnDescription a → a → SQLValue


getColumnValueSelector :: ColumnDescription a → SQLValue → a


toValueOrNull :: (a → SQLValue) → Maybe a → SQLValue


sqlIntOrNull :: Maybe Int → SQLValue


sqlFloatOrNull :: Maybe Float → SQLValue


sqlCharOrNull :: Maybe Char → SQLValue


sqlStringOrNull :: Maybe String → SQLValue


sqlString :: String → SQLValue


sqlBoolOrNull :: Maybe Bool → SQLValue


sqlDateOrNull :: Maybe ClockTime → SQLValue


intOrNothing :: SQLValue → Maybe Int
```

```
floatOrNothing :: SQLValue → Maybe Float
```

```
charOrNothing :: SQLValue → Maybe Char
```

```
stringOrNothing :: SQLValue → Maybe String
```

```
fromStringOrNull :: SQLValue → String
```

```
boolOrNothing :: SQLValue → Maybe Bool
```

```
dateOrNothing :: SQLValue → Maybe ClockTime
```

A.4.5   Library Database.CDBI.ER

This is the main CDBI-module. It provides datatypes and functions to do Database-Queries working with Entities (ER-Model)

Exported functions:

```
insertEntry :: a → EntityDescription a → Connection → IO (Either DBError ())
```

Inserts an entry into the database.

```
saveEntry :: a → EntityDescription a → Connection → IO (Either DBError ())
```

Saves an entry to the database (only for backward compatibility).

```
insertEntries :: [a] → EntityDescription a → Connection → IO (Either DBError
())
```

Inserts several entries into the database.

```
saveMultipleEntries :: [a] → EntityDescription a → Connection → IO (Either
DBError ())
```

Saves multiple entries to the database (only for backward compatibility).

```
restoreEntries :: [a] → EntityDescription a → Connection → IO (Either DBError
())
```

Stores entries with their current keys in the database. It is an error if entries with the same key are already in the database. Thus, this operation is useful only to restore a database with saved data.

getEntries :: Specifier → EntityDescription a → Criteria → [Option] → Maybe Int → Connection → IO (Either DBError [a])

Gets entries from the database.

getColumn :: [SetOp] → [SingleColumnSelect a] → [Option] → Maybe Int → Connection → IO (Either DBError [a])

Gets a single Column from the database.

getColumnTuple :: [SetOp] → [TupleColumnSelect a b] → [Option] → Maybe Int → Connection → IO (Either DBError [(a,b)])

Gets two Columns from the database.

getColumnTriple :: [SetOp] → [TripleColumnSelect a b c] → [Option] → Maybe Int → Connection → IO (Either DBError [(a,b,c)])

Gets three Columns from the database.

getColumnFourTuple :: [SetOp] → [FourColumnSelect a b c d] → [Option] → Maybe Int → Connection → IO (Either DBError [(a,b,c,d)])

Gets four Columns from the database.

getColumnFiveTuple :: [SetOp] → [FiveColumnSelect a b c d e] → [Option] → Maybe Int → Connection → IO (Either DBError [(a,b,c,d,e)])

Gets five Columns from the database.

getEntriesCombined :: Specifier → CombinedDescription a → [Join] → Criteria → [Option] → Maybe Int → Connection → IO (Either DBError [a])

Gets combined entries from the database.

insertEntryCombined :: a → CombinedDescription a → Connection → IO (Either DBError ())

Inserts combined entries.

saveEntryCombined :: a → CombinedDescription a → Connection → IO (Either DBError ())

Saves combined entries (for backward compatibility).

updateEntries :: EntityDescription a → [ColVal] → Constraint → Connection → IO (Either DBError ())

Updates entries depending on wether they fulfill the criteria or not

`updateEntry :: a → EntityDescription a → Connection → IO (Either DBError ())`

>   Updates an entry by ID. Works for Entities that have a primary key as first value. Function will update the entry in the database with the ID of the entry that is given as parameter with the values of the entry given as parameter

`updateEntryCombined :: a → CombinedDescription a → Connection → IO (Either DBError ())`

>   Same as updateEntry but for combined Data

`deleteEntries :: EntityDescription a → Maybe Constraint → Connection → IO (Either DBError ())`

>   Deletes entries depending on wether they fulfill the criteria or not

A.4.6 Library Database.CDBI.QueryTypes

This module contains datatype declarations, constructor functions selectors and translation functions for complex select queries in particular for those selecting (1 to 5) single columns.

Exported types:

`type ColumnTupleCollection a b = (ColumnSingleCollection a,ColumnSingleCollection b)`

>   Datatype to select two different columns which can be of different types and from different tables.

`type ColumnTripleCollection a b c = (ColumnSingleCollection a,ColumnSingleCollection b,ColumnSingleCollection c)`

>   Datatype to select three different columns which can be of different types and from different tables.

`type ColumnFourTupleCollection a b c d = (ColumnSingleCollection a,ColumnSingleCollection b,ColumnSingleCollection c,ColumnSingleCollection d)`

>   Datatype to select four different columns which can be of different types and from different tables.

`type ColumnFiveTupleCollection a b c d e = (ColumnSingleCollection a,ColumnSingleCollection b,ColumnSingleCollection c,ColumnSingleCollection d,ColumnSingleCollection e)`

>   Datatype to select five different columns which can be of different types and from different tables.

`data SetOp`

Exported constructors:

- `Union :: SetOp`

- `Intersect :: SetOp`

- `Except :: SetOp`

data Join

datatype for joins

Exported constructors:

- `Cross :: Join`

- `Inner :: Constraint → Join`

data TableClause

data structure to represent a table-clause (tables and joins) in a way that at least one table has to be specified

Exported constructors:

- `TC :: String → Int → (Maybe (Join,TableClause)) → TableClause`

data ColumnSingleCollection

Datatype representing a single column in a select-clause. Can be just a column connected with an alias and an optional aggregation function(String) or a Case-when-then-statement

Exported constructors:

- `ResultColumnDescription :: (ColumnDescription a) → Int → String → ColumnSingleCollection a`

- `Case :: Condition → (Value (),Value ()) → (SQLType,SQLValue → a) → ColumnSingleCollection a`

data SingleColumnSelect

Datatype to describe all parts of a select-query without Setoperators order-by and limit (selecthead) for a single column.

Exported constructors:

- `SingleCS :: Specifier → (ColumnSingleCollection a) → TableClause → Criteria → SingleColumnSelect a`

data TupleColumnSelect

Datatype to describe all parts of a select-query without Setoperators order-by and limit (selecthead) for two columns.

Exported constructors:

- `TupleCS :: Specifier →` (`ColumnSingleCollection a,ColumnSingleCollection b`) `→ TableClause → Criteria → TupleColumnSelect a b`

data TripleColumnSelect

Datatype to describe all parts of a select-query without Setoperators order-by and limit (selecthead) for three columns.

Exported constructors:

- `TripleCS :: Specifier →` (`ColumnSingleCollection a,ColumnSingleCollection b,ColumnSingleCollection c`) `→ TableClause → Criteria → TripleColumnSelect a b c`

data FourColumnSelect

Datatype to describe all parts of a select-query without Setoperators order-by and limit (selecthead) for four columns.

Exported constructors:

- `FourCS :: Specifier →` (`ColumnSingleCollection a,ColumnSingleCollection b,ColumnSingleCollection c,ColumnSingleCollection d`) `→ TableClause → Criteria → FourColumnSelect a b c d`

data FiveColumnSelect

Datatype to describe all parts of a select-query without Setoperators order-by and limit (selecthead) for five columns.

Exported constructors:

- `FiveCS :: Specifier →` (`ColumnSingleCollection a,ColumnSingleCollection b,ColumnSingleCollection c,ColumnSingleCollection d,ColumnSingleCollection e`) `→ TableClause → Criteria → FiveColumnSelect a b c d e`

Exported functions:

`innerJoin :: Constraint → Join`

Constructorfunction for an inner join

`crossJoin :: Join`

Constructorfunction for cross join

sum :: Specifier → ColumnDescription a → (String,ColumnDescription Float)

> Constructor for aggregation function sum in select-clauses. A pseudo-ResultColumnDescription of type float is created for correct return type.

avg :: Specifier → ColumnDescription a → (String,ColumnDescription Float)

> Constructor for aggregation function avg in select-clauses. A pseudo-ResultColumnDescription of type float is created for correct return type.

count :: Specifier → ColumnDescription a → (String,ColumnDescription Int)

> Constructor for aggregation function count in select-clauses. A pseudo-ResultColumnDescription of type float is created for correct return type.

minV :: ColumnDescription a → (String,ColumnDescription a)

> Constructor for aggregation function min in select-clauses.

maxV :: ColumnDescription a → (String,ColumnDescription a)

> Constructor for aggregation function max in select-clauses.

none :: ColumnDescription a → (String,ColumnDescription a)

> Constructor function in case no aggregation function is specified.

caseResultInt :: (SQLType,SQLValue → Int)

caseResultFloat :: (SQLType,SQLValue → Float)

caseResultString :: (SQLType,SQLValue → String)

caseResultBool :: (SQLType,SQLValue → Bool)

caseResultChar :: (SQLType,SQLValue → Char)

caseThen :: Condition → Value a → Value a → (SQLType,SQLValue → a) →
ColumnSingleCollection a

> Constructor function for representation of statement: CASE WHEN condition THEN val1 ELSE val2 END. It does only work for the same type in then and else branch.

singleCol :: ColumnDescription a → Int → (ColumnDescription a →
(String,ColumnDescription b)) → ColumnSingleCollection b

Constructorfunction for ColumnSingleCollection.

```
tupleCol :: ColumnSingleCollection a → ColumnSingleCollection b →
(ColumnSingleCollection a,ColumnSingleCollection b)


tripleCol :: ColumnSingleCollection a → ColumnSingleCollection b →
ColumnSingleCollection c → (ColumnSingleCollection a,ColumnSingleCollection
b,ColumnSingleCollection c)


fourCol :: ColumnSingleCollection a → ColumnSingleCollection b →
ColumnSingleCollection c → ColumnSingleCollection d → (ColumnSingleCollection
a,ColumnSingleCollection b,ColumnSingleCollection c,ColumnSingleCollection d)


fiveCol :: ColumnSingleCollection a → ColumnSingleCollection b →
ColumnSingleCollection c → ColumnSingleCollection d → ColumnSingleCollection
e → (ColumnSingleCollection a,ColumnSingleCollection b,ColumnSingleCollection
c,ColumnSingleCollection d,ColumnSingleCollection e)


getSingleType :: SingleColumnSelect a → [SQLType]


getSingleValFunc :: SingleColumnSelect a → SQLValue → a


getTupleTypes :: TupleColumnSelect a b → [SQLType]


getTupleValFuncs :: TupleColumnSelect a b → (SQLValue → a,SQLValue → b)


getTripleTypes :: TripleColumnSelect a b c → [SQLType]


getTripleValFuncs :: TripleColumnSelect a b c → (SQLValue → a,SQLValue →
b,SQLValue → c)


getFourTupleTypes :: FourColumnSelect a b c d → [SQLType]
```

```haskell
getFourTupleValFuncs :: FourColumnSelect a b c d → (SQLValue → a,SQLValue →
b,SQLValue → c,SQLValue → d)


getFiveTupleTypes :: FiveColumnSelect a b c d e → [SQLType]


getFiveTupleValFuncs :: FiveColumnSelect a b c d e → (SQLValue → a,SQLValue →
b,SQLValue → c,SQLValue → d,SQLValue → e)


trSingleSelectQuery :: SingleColumnSelect a → String


trTupleSelectQuery :: TupleColumnSelect a b → String


trTripleSelectQuery :: TripleColumnSelect a b c → String


trFourTupleSelectQuery :: FourColumnSelect a b c d → String


trFiveTupleSelectQuery :: FiveColumnSelect a b c d e → String


trSetOp :: SetOp → String


trLimit :: Maybe Int → String


asTable :: String → Int → String


trJoinPart1 :: Join → String


trJoinPart2 :: Join → String
```

A.4.7 Library Database.ERD

This module contains the definition of data types to represent entity/relationship diagrams and an I/O operation to read them from a term file.

Exported types:

```
type ERDName = String
```

```
type EName = String
```

```
type AName = String
```

```
type Null = Bool
```

```
type RName = String
```

```
type Role = String
```

```
data ERD
```

>   Data type to represent entity/relationship diagrams.

  Exported constructors:

  • ERD :: String → [Entity] → [Relationship] → ERD

```
data Entity
```

  Exported constructors:

  • Entity :: String → [Attribute] → Entity

```
data Attribute
```

  Exported constructors:

  • Attribute :: String → Domain → Key → Bool → Attribute

```
data Key
```

Exported constructors:

- NoKey :: Key

- PKey :: Key

- Unique :: Key

```
data Domain
```

Exported constructors:

- IntDom :: (Maybe Int) → Domain

- FloatDom :: (Maybe Float) → Domain

- CharDom :: (Maybe Char) → Domain

- StringDom :: (Maybe String) → Domain

- BoolDom :: (Maybe Bool) → Domain

- DateDom :: (Maybe CalendarTime) → Domain

- UserDefined :: String → (Maybe String) → Domain

- KeyDom :: String → Domain

```
data Relationship
```

Exported constructors:

- Relationship :: String → [REnd] → Relationship

```
data REnd
```

Exported constructors:

- REnd :: String → String → Cardinality → REnd

```
data Cardinality
```

Cardinality of a relationship w.r.t. some entity. The cardinality is either a fixed number (e.g., (Exactly 1) representing the cardinality (1,1)) or an interval (e.g., (Between 1 (Max 4)) representing the cardinality (1,4), or (Between 0 Infinite) representing the cardinality (0,n)).

Exported constructors:

- Exactly :: Int → Cardinality

- Between :: Int → MaxValue → Cardinality

data MaxValue

The upper bound of a cardinality which is either a finite number or infinite.

Exported constructors:

- Max :: Int → MaxValue

- Infinite :: MaxValue

Exported functions:

readERDTermFile :: String → IO ERD

Read an ERD specification from a file containing a single ERD term.

A.4.8 Library Database.ERDGoodies

This module contains some useful operations on the data types representing entity/relationship diagrams

Exported functions:

erdName :: ERD → String

The name of an ERD.

entityName :: Entity → String

The name of an entity.

isEntityNamed :: String → Entity → Bool

Is this an entity with a given name?

hasForeignKey :: String → Entity → Bool

Has the entity an attribute with a foreign key for a given entity name?

foreignKeyAttributes :: String → [Attribute] → [Attribute]

Returns the attributes that are a foreign key of a given entity name.

`entityAttributes :: Entity → [Attribute]`

> The attributes of an entity

`attributeName :: Attribute → String`

> The name of an attribute.

`attributeDomain :: Attribute → Domain`

> The domain of an attribute.

`hasDefault :: Domain → Bool`

> Has an attribute domain a default value?

`isForeignKey :: Attribute → Bool`

`isNullAttribute :: Attribute → Bool`

> Has an attribute a null value?

`cardMinimum :: Cardinality → Int`

> The minimum value of a cardinality.

`cardMaximum :: Cardinality → Int`

> The maximum value of a cardinality (provided that it is not infinite).

`showERD :: Int → ERD → String`

> A simple pretty printer for ERDs.

`combineIds :: [String] → String`

> Combines a non-empty list of identifiers into a single identifier. Used in ERD trans-
> formation and code generation to create names for combined objects, e.g., relationships
> and foreign keys.

`storeERDFromProgram :: String → IO String`

> Writes the ERD defined in a Curry program (as a top-level operation of type
> `Database.ERD.ERD`) in a term file and return the name of the term file.

`writeERDTermFile :: ERD → IO ()`

> Writes an ERD term into a file with name `ERDMODELNAME.erdterm` and prints the name
> of the generated file.

A.5   Libraries for Web Applications

A.5.1   Library Bootstrap3Style

This library contains some operations to generate web pages rendered with Bootstrap

Exported functions:

bootstrapForm :: String → [String] → String → (String,[HtmlExp]) → [[HtmlExp]]
→ [[HtmlExp]] → Int → [HtmlExp] → [HtmlExp] → [HtmlExp] → [HtmlExp] →
HtmlForm

     An HTML form rendered with bootstrap.

bootstrapPage :: String → [String] → String → (String,[HtmlExp]) → [[HtmlExp]]
→ [[HtmlExp]] → Int → [HtmlExp] → [HtmlExp] → [HtmlExp] → [HtmlExp] →
HtmlPage

     An HTML page rendered with bootstrap.

titledSideMenu :: String → [[HtmlExp]] → [HtmlExp]

defaultButton :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp

     Default input button.

smallButton :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp

     Small input button.

primButton :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp

     Primary input button.

hrefButton :: String → [HtmlExp] → HtmlExp

     Hypertext reference rendered as a button.

hrefBlock :: String → [HtmlExp] → HtmlExp

     Hypertext reference rendered as a block level button.

hrefInfoBlock :: String → [HtmlExp] → HtmlExp

     Hypertext reference rendered as an info block level button.

glyphicon :: String → HtmlExp

homeIcon :: HtmlExp

```
userIcon :: HtmlExp
```

```
loginIcon :: HtmlExp
```

```
logoutIcon :: HtmlExp
```

A.5.2 Library CategorizedHtmlList

This library provides functions to categorize a list of entities into a HTML page with an index access (e.g., "A-Z") to these entities.

Exported functions:

```
list2CategorizedHtml :: [(a,[HtmlExp])] → [(b,String)] → (a → b → Bool) →
[HtmlExp]
```

> General categorization of a list of entries.
>
> The item will occur in every category for which the boolean function categoryFun yields True.

```
categorizeByItemKey :: [(String,[HtmlExp])] → [HtmlExp]
```

> Categorize a list of entries with respect to the inial keys.
>
> The categories are named as all initial characters of the keys of the items.

```
stringList2ItemList :: [String] → [(String,[HtmlExp])]
```

> Convert a string list into an key-item list The strings are used as keys and for the simple text layout.

A.5.3 Library HTML

Library for HTML and CGI programming. This paper contains a description of the basic ideas behind this library.
The installation of a cgi script written with this library can be done by the command

```
curry makecgi -m initialForm -o /home/joe/public_html/prog.cgi prog
```

where **prog** is the name of the Curry program with the cgi script, /home/joe/public_html/prog.cgi is the desired location of the compiled cgi script, and initialForm is the Curry expression (of type IO HtmlForm) computing the HTML form (where curry is the shell command calling the Curry system PAKCS or KiCS2).

246

Exported types:

`type CgiEnv = CgiRef → String`

> The type for representing cgi environments (i.e., mappings from cgi references to the corresponding values of the input elements).

`type HtmlHandler = (CgiRef → String) → IO HtmlForm`

> The type of event handlers in HTML forms.

`data CgiRef`

> The (abstract) data type for representing references to input elements in HTML forms.

Exported constructors:

`data HtmlExp`

> The data type for representing HTML expressions.

Exported constructors:

- `HtmlText :: String → HtmlExp`

  `HtmlText s`

  – a text string without any further structure

- `HtmlStruct :: String → [(String,String)] → [HtmlExp] → HtmlExp`

  `HtmlStruct t as hs`

  – a structure with a tag, attributes, and HTML expressions inside the structure

- `HtmlCRef :: HtmlExp → CgiRef → HtmlExp`

  `HtmlCRef h ref`

  – an input element (described by the first argument) with a cgi reference

- `HtmlEvent :: HtmlExp → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

  `HtmlEvent h hdlr`

  – an input element (first arg) with an associated event handler (tpyically, a submit button)

`data HtmlForm`

> The data type for representing HTML forms (active web pages) and return values of HTML forms.

Exported constructors:

- `HtmlForm :: String → [FormParam] → [HtmlExp] → HtmlForm`

  `HtmlForm t ps hs`

    – an HTML form with title t, optional parameters (e.g., cookies) ps, and contents hs

- `HtmlAnswer :: String → String → HtmlForm`

  `HtmlAnswer t c`

    – an answer in an arbitrary format where t is the content type (e.g., "text/plain") and c is the contents

`data FormParam`

The possible parameters of an HTML form. The parameters of a cookie (FormCookie) are its name and value and optional parameters (expiration date, domain, path (e.g., the path "/" makes the cookie valid for all documents on the server), security) which are collected in a list.

Exported constructors:

- `FormCookie :: String → String → [CookieParam] → FormParam`

  `FormCookie name value params`

    – a cookie to be sent to the client's browser

- `FormCSS :: String → FormParam`

  `FormCSS s`

    – a URL for a CSS file for this form

- `FormJScript :: String → FormParam`

  `FormJScript s`

    – a URL for a Javascript file for this form

- `FormOnSubmit :: String → FormParam`

  `FormOnSubmit s`

    – a JavaScript statement to be executed when the form is submitted (i.e., <form ... onsubmit="s">)

- `FormTarget :: String → FormParam`

  `FormTarget s`

    – a name of a target frame where the output of the script should be represented (should only be used for scripts running in a frame)

- `FormEnc :: String → FormParam`

  `FormEnc`

- the encoding scheme of this form

- FormMeta :: [(String,String)] → FormParam

  FormMeta as

    - meta information (in form of attributes) for this form

- HeadInclude :: HtmlExp → FormParam

  HeadInclude he

    - HTML expression to be included in form header

- MultipleHandlers :: FormParam

  MultipleHandlers

    - indicates that the event handlers of the form can be multiply used (i.e., are not deleted if the form is submitted so that they are still available when going back in the browser; but then there is a higher risk that the web server process might overflow with unused events); the default is a single use of event handlers, i.e., one cannot use the back button in the browser and submit the same form again (which is usually a reasonable behavior to avoid double submissions of data).

- BodyAttr :: (String,String) → FormParam

  BodyAttr ps

    - optional attribute for the body element (more than one occurrence is allowed)

data CookieParam

The possible parameters of a cookie.

Exported constructors:

- CookieExpire :: ClockTime → CookieParam

- CookieDomain :: String → CookieParam

- CookiePath :: String → CookieParam

- CookieSecure :: CookieParam

data HtmlPage

The data type for representing HTML pages. The constructor arguments are the title, the parameters, and the contents (body) of the web page.

Exported constructors:

- HtmlPage :: String → [PageParam] → [HtmlExp] → HtmlPage

```
data PageParam
```

The possible parameters of an HTML page.

Exported constructors:

- `PageEnc :: String → PageParam`

  `PageEnc`

  – the encoding scheme of this page

- `PageCSS :: String → PageParam`

  `PageCSS s`

  – a URL for a CSS file for this page

- `PageJScript :: String → PageParam`

  `PageJScript s`

  – a URL for a Javascript file for this page

- `PageMeta :: [(String,String)] → PageParam`

  `PageMeta as`

  – meta information (in form of attributes) for this page

- `PageLink :: [(String,String)] → PageParam`

  `PageLink as`

  – link information (in form of attributes) for this page

- `PageBodyAttr :: (String,String) → PageParam`

  `PageBodyAttr attr`

  – optional attribute for the body element of the page (more than one occurrence is allowed)

Exported functions:

`defaultEncoding :: String`

The default encoding used in generated web pages.

`idOfCgiRef :: CgiRef → String`

Internal identifier of a CgiRef (intended only for internal use in other libraries!).

`formEnc :: String → FormParam`

An encoding scheme for a HTML form.

`formCSS :: String → FormParam`

A URL for a CSS file for a HTML form.

`formMetaInfo :: [(String,String)] → FormParam`

Meta information for a HTML form. The argument is a list of attributes included in the `meta`-tag in the header for this form.

`formBodyAttr :: (String,String) → FormParam`

Optional attribute for the body element of the HTML form. More than one occurrence is allowed, i.e., all such attributes are collected.

`form :: String → [HtmlExp] → HtmlForm`

A basic HTML form for active web pages with the default encoding and a default background.

`standardForm :: String → [HtmlExp] → HtmlForm`

A standard HTML form for active web pages where the title is included in the body as the first header.

`cookieForm :: String → [(String,String)] → [HtmlExp] → HtmlForm`

An HTML form with simple cookies. The cookies are sent to the client's browser together with this form.

`addCookies :: [(String,String)] → HtmlForm → HtmlForm`

Add simple cookie to HTML form. The cookies are sent to the client's browser together with this form.

`answerText :: String → HtmlForm`

A textual result instead of an HTML form as a result for active web pages.

`answerEncText :: String → String → HtmlForm`

A textual result instead of an HTML form as a result for active web pages where the encoding is given as the first parameter.

`addFormParam :: HtmlForm → FormParam → HtmlForm`

Adds a parameter to an HTML form.

`redirect :: Int → String → HtmlForm → HtmlForm`

Adds redirection to given HTML form.

`expires :: Int → HtmlForm → HtmlForm`

Adds expire time to given HTML form.

`addSound :: String → Bool → HtmlForm → HtmlForm`

Adds sound to given HTML form. The functions adds two different declarations for sound, one invented by Microsoft for the internet explorer, one introduced for netscape. As neither is an official part of HTML, addsound might not work on all systems and browsers. The greatest chance is by using sound files in MID-format.

pageEnc :: String → PageParam

An encoding scheme for a HTML page.

pageCSS :: String → PageParam

A URL for a CSS file for a HTML page.

pageMetaInfo :: [(String,String)] → PageParam

Meta information for a HTML page. The argument is a list of attributes included in the meta-tag in the header for this page.

pageLinkInfo :: [(String,String)] → PageParam

Link information for a HTML page. The argument is a list of attributes included in the link-tag in the header for this page.

pageBodyAttr :: (String,String) → PageParam

Optional attribute for the body element of the web page. More than one occurrence is allowed, i.e., all such attributes are collected.

page :: String → [HtmlExp] → HtmlPage

A basic HTML web page with the default encoding.

standardPage :: String → [HtmlExp] → HtmlPage

A standard HTML web page where the title is included in the body as the first header.

addPageParam :: HtmlPage → PageParam → HtmlPage

Adds a parameter to an HTML page.

htxt :: String → HtmlExp

Basic text as HTML expression. The text may contain special HTML chars (like <,>,&,") which will be quoted so that they appear as in the parameter string.

htxts :: [String] → [HtmlExp]

A list of strings represented as a list of HTML expressions. The strings may contain special HTML chars that will be quoted.

hempty :: HtmlExp

An empty HTML expression.

```
nbsp :: HtmlExp
```

Non breaking Space

```
h1 :: [HtmlExp] → HtmlExp
```

Header 1

```
h2 :: [HtmlExp] → HtmlExp
```

Header 2

```
h3 :: [HtmlExp] → HtmlExp
```

Header 3

```
h4 :: [HtmlExp] → HtmlExp
```

Header 4

```
h5 :: [HtmlExp] → HtmlExp
```

Header 5

```
par :: [HtmlExp] → HtmlExp
```

Paragraph

```
section :: [HtmlExp] → HtmlExp
```

Section

```
header :: [HtmlExp] → HtmlExp
```

Header

```
footer :: [HtmlExp] → HtmlExp
```

Footer

```
emphasize :: [HtmlExp] → HtmlExp
```

Emphasize

```
strong :: [HtmlExp] → HtmlExp
```

Strong (more emphasized) text.

```
bold :: [HtmlExp] → HtmlExp
```

Boldface

```
italic :: [HtmlExp] → HtmlExp
```

Italic

`nav :: [HtmlExp] → HtmlExp`

> Navigation

`code :: [HtmlExp] → HtmlExp`

> Program code

`center :: [HtmlExp] → HtmlExp`

> Centered text

`blink :: [HtmlExp] → HtmlExp`

> Blinking text

`teletype :: [HtmlExp] → HtmlExp`

> Teletype font

`pre :: [HtmlExp] → HtmlExp`

> Unformatted input, i.e., keep spaces and line breaks and don't quote special characters.

`verbatim :: String → HtmlExp`

> Verbatim (unformatted), special characters ($<$,$>$,&,") are quoted.

`address :: [HtmlExp] → HtmlExp`

> Address

`href :: String → [HtmlExp] → HtmlExp`

> Hypertext reference

`anchor :: String → [HtmlExp] → HtmlExp`

> An anchored text with a hypertext reference inside a document.

`ulist :: [[HtmlExp]] → HtmlExp`

> Unordered list

`olist :: [[HtmlExp]] → HtmlExp`

> Ordered list

`litem :: [HtmlExp] → HtmlExp`

> A single list item (usually not explicitly used)

`dlist :: [([HtmlExp],[HtmlExp])] → HtmlExp`

> Description list

`table :: [[[HtmlExp]]] → HtmlExp`

    Table with a matrix of items where each item is a list of HTML expressions.

`headedTable :: [[[HtmlExp]]] → HtmlExp`

    Similar to `table` but introduces header tags for the first row.

`addHeadings :: HtmlExp → [[HtmlExp]] → HtmlExp`

    Add a row of items (where each item is a list of HTML expressions) as headings to a table. If the first argument is not a table, the headings are ignored.

`hrule :: HtmlExp`

    Horizontal rule

`breakline :: HtmlExp`

    Break a line

`image :: String → String → HtmlExp`

    Image

`styleSheet :: String → HtmlExp`

    Defines a style sheet to be used in this HTML document.

`style :: String → [HtmlExp] → HtmlExp`

    Provides a style for HTML elements. The style argument is the name of a style class defined in a style definition (see `styleSheet`) or in an external style sheet (see form and page parameters `FormCSS` and `PageCSS`).

`textstyle :: String → String → HtmlExp`

    Provides a style for a basic text. The style argument is the name of a style class defined in an external style sheet.

`blockstyle :: String → [HtmlExp] → HtmlExp`

    Provides a style for a block of HTML elements. The style argument is the name of a style class defined in an external style sheet. This element is used (in contrast to "style") for larger blocks of HTML elements since a line break is placed before and after these elements.

`inline :: [HtmlExp] → HtmlExp`

    Joins a list of HTML elements into a single HTML element. Although this construction has no rendering, it is sometimes useful for programming when several HTML elements must be put together.

`block :: [HtmlExp] → HtmlExp`

Joins a list of HTML elements into a block. A line break is placed before and after these elements.

`button :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

Submit button with a label string and an event handler

`resetbutton :: String → HtmlExp`

Reset button with a label string

`imageButton :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

Submit button in form of an imag.

`textfield :: CgiRef → String → HtmlExp`

Input text field with a reference and an initial contents

`password :: CgiRef → HtmlExp`

Input text field (where the entered text is obscured) with a reference

`textarea :: CgiRef → (Int,Int) → String → HtmlExp`

Input text area with a reference, height/width, and initial contents

`checkbox :: CgiRef → String → HtmlExp`

A checkbox with a reference and a value. The value is returned if checkbox is on, otherwise "" is returned.

`checkedbox :: CgiRef → String → HtmlExp`

A checkbox that is initially checked with a reference and a value. The value is returned if checkbox is on, otherwise "" is returned.

`radio_main :: CgiRef → String → HtmlExp`

A main button of a radio (initially "on") with a reference and a value. The value is returned of this button is on. A complete radio button suite always consists of a main button (radio*main) and some further buttons (radio*others) with the same reference. Initially, the main button is selected (or nothing is selected if one uses radio*main*off instead of radio_main). The user can select another button but always at most one button of the radio can be selected. The value corresponding to the selected button is returned in the environment for this radio reference.

`radio_main_off :: CgiRef → String → HtmlExp`

A main button of a radio (initially "off") with a reference and a value. The value is returned of this button is on.

`radio_other :: CgiRef → String → HtmlExp`

A further button of a radio (initially "off") with a reference (identical to the main button of this radio) and a value. The value is returned of this button is on.

selection :: CgiRef → [(String,String)] → HtmlExp

A selection button with a reference and a list of name/value pairs. The names are shown in the selection and the value is returned for the selected name.

selectionInitial :: CgiRef → [(String,String)] → Int → HtmlExp

A selection button with a reference, a list of name/value pairs, and a preselected item in this list. The names are shown in the selection and the value is returned for the selected name.

multipleSelection :: CgiRef → [(String,String,Bool)] → HtmlExp

A selection button with a reference and a list of name/value/flag pairs. The names are shown in the selection and the value is returned if the corresponding name is selected. If flag is True, the corresonding name is initially selected. If more than one name has been selected, all values are returned in one string where the values are separated by newline (<code>\n</code>) characters.

hiddenfield :: String → String → HtmlExp

A hidden field to pass a value referenced by a fixed name. This function should be used with care since it may cause conflicts with the CGI-based implementation of this library.

htmlQuote :: String → String

Quotes special characters ($<$,$>$,&,", umlauts) in a string as HTML special characters.

htmlIsoUmlauts :: String → String

Translates umlauts in iso-8859-1 encoding into HTML special characters.

addAttr :: HtmlExp → (String,String) → HtmlExp

Adds an attribute (name/value pair) to an HTML element.

addAttrs :: HtmlExp → [(String,String)] → HtmlExp

Adds a list of attributes (name/value pair) to an HTML element.

addClass :: HtmlExp → String → HtmlExp

Adds a class attribute to an HTML element.

showHtmlExps :: [HtmlExp] → String

Transforms a list of HTML expressions into string representation.

showHtmlExp :: HtmlExp → String

Transforms a single HTML expression into string representation.

`showHtmlPage :: HtmlPage → String`

Transforms HTML page into string representation.

`getUrlParameter :: IO String`

Gets the parameter attached to the URL of the script. For instance, if the script is called with URL "http://.../script.cgi?parameter", then "parameter" is returned by this I/O action. Note that an URL parameter should be "URL encoded" to avoid the appearance of characters with a special meaning. Use the functions "urlencoded2string" and "string2urlencoded" to decode and encode such parameters, respectively.

`urlencoded2string :: String → String`

Translates urlencoded string into equivalent ASCII string.

`string2urlencoded :: String → String`

Translates arbitrary strings into equivalent urlencoded string.

`getCookies :: IO [(String,String)]`

Gets the cookies sent from the browser for the current CGI script. The cookies are represented in the form of name/value pairs since no other components are important here.

`coordinates :: (CgiRef → String) → Maybe (Int,Int)`

For image buttons: retrieve the coordinates where the user clicked within the image.

`runFormServerWithKey :: String → String → IO HtmlForm → IO ()`

The server implementing an HTML form (possibly containing input fields). It receives a message containing the environment of the client's web browser, translates the HTML form w.r.t. this environment into a string representation of the complete HTML document and sends the string representation back to the client's browser by binding the corresponding message argument.

`runFormServerWithKeyAndFormParams :: String → String → [FormParam] → IO HtmlForm → IO ()`

The server implementing an HTML form (possibly containing input fields). It receives a message containing the environment of the client's web browser, translates the HTML form w.r.t. this environment into a string representation of the complete HTML document and sends the string representation back to the client's browser by binding the corresponding message argument.

`showLatexExps :: [HtmlExp] → String`

Transforms HTML expressions into LaTeX string representation.

`showLatexExp :: HtmlExp → String`

Transforms an HTML expression into LaTeX string representation.

`htmlSpecialChars2tex :: String → String`

Convert special HTML characters into their LaTeX representation, if necessary.

`showLatexDoc :: [HtmlExp] → String`

Transforms HTML expressions into a string representation of a complete LaTeX document.

`showLatexDocWithPackages :: [HtmlExp] → [String] → String`

Transforms HTML expressions into a string representation of a complete LaTeX document. The variable "packages" holds the packages to add to the latex document e.g. "ngerman"

`showLatexDocs :: [[HtmlExp]] → String`

Transforms a list of HTML expressions into a string representation of a complete LaTeX document where each list entry appears on a separate page.

`showLatexDocsWithPackages :: [[HtmlExp]] → [String] → String`

Transforms a list of HTML expressions into a string representation of a complete LaTeX document where each list entry appears on a separate page. The variable "packages" holds the packages to add to the latex document (e.g., "ngerman").

`germanLatexDoc :: [HtmlExp] → String`

show german latex document

`intForm :: IO HtmlForm → IO ()`

Execute an HTML form in "interactive" mode.

`intFormMain :: String → String → String → String → Bool → String → IO HtmlForm → IO ()`

Execute an HTML form in "interactive" mode with various parameters.

A.5.4 Library HtmlCgi

Library to support CGI programming in the HTML library. It is only intended as an auxiliary library to implement dynamic web pages according to the HTML library. It contains a simple script that is installed for a dynamic web page and which sends the user input to the real application server implementing the application.

Exported types:

`data CgiServerMsg`

> The messages to comunicate between the cgi script and the server program. CgiSubmit env cgienv nextpage - pass the environment and show next page, where env are the values of the environment variables of the web script (e.g., QUERY$STRING, REMOTE$HOST, REMOTE_ADDR), cgienv are the values in the current form submitted by the client, and nextpage is the answer text to be shown in the next web page

Exported constructors:

- `CgiSubmit :: [(String,String)]` → `[(String,String)]` → `CgiServerMsg`

- `GetLoad :: CgiServerMsg`

  `GetLoad`

    – get info about the current load of the server process

- `SketchStatus :: CgiServerMsg`

  `SketchStatus`

    – get a sketch of the status of the server

- `SketchHandlers :: CgiServerMsg`

  `SketchHandlers`

    – get a sketch of all event handlers of the server

- `ShowStatus :: CgiServerMsg`

  `ShowStatus`

    – show the status of the server with all event handlers

- `CleanServer :: CgiServerMsg`

  `CleanServer`

    – clean up the server (with possible termination)

- `StopCgiServer :: CgiServerMsg`

  `StopCgiServer`

    – stop the server

Exported functions:

readCgiServerMsg :: Handle → IO (Maybe CgiServerMsg)

    Reads a line from a handle and check whether it is a syntactically correct cgi server message.

submitForm :: IO ()

runCgiServerCmd :: String → CgiServerMsg → IO ()

    Executes a specific command for a cgi server.

noHandlerPage :: String → String → String

cgiServerRegistry :: String

    The name of the file to register all cgi servers.

registerCgiServer :: String → String → IO ()

unregisterCgiServer :: String → IO ()

A.5.5 Library HtmlParser

This module contains a very simple parser for HTML documents.

Exported functions:

readHtmlFile :: String → IO [HtmlExp]

    Reads a file with HTML text and returns the corresponding HTML expressions.

parseHtmlString :: String → [HtmlExp]

    Transforms an HTML string into a list of HTML expressions. If the HTML string is a well structured document, the list of HTML expressions should contain exactly one element.

A.5.6 Library Mail

This library contains functions for sending emails. The implementation might need to be adapted to the local environment.

Exported types:

`data MailOption`

> Options for sending emails.

> *Exported constructors:*

- `CC :: String → MailOption`

  `CC`

    – recipient of a carbon copy

- `BCC :: String → MailOption`

  `BCC`

    – recipient of a blind carbon copy

- `TO :: String → MailOption`

  `TO`

    – recipient of the email

Exported functions:

`sendMail :: String → String → String → String → IO ()`

> Sends an email via mailx command.

`sendMailWithOptions :: String → String → [MailOption] → String → IO ()`

> Sends an email via mailx command and various options. Note that multiple options are allowed, e.g., more than one CC option for multiple recipient of carbon copies.

> Important note: The implementation of this operation is based on the command "mailx" and must be adapted according to your local environment!

A.5.7 Library Markdown

Library to translate markdown documents into HTML or LaTeX. The slightly restricted subset of the markdown syntax recognized by this implementation is documented in this page.

Exported types:

`type MarkdownDoc = [MarkdownElem]`

> A markdown document is a list of markdown elements.

`data MarkdownElem`

> The data type for representing the different elements occurring in a markdown document.

Exported constructors:

- `Text :: String → MarkdownElem`

  `Text s`

  – a simple text in a markdown document

- `Emph :: String → MarkdownElem`

  `Emph s`

  – an emphasized text in a markdown document

- `Strong :: String → MarkdownElem`

  `Strong s`

  – a strongly emphaszed text in a markdown document

- `Code :: String → MarkdownElem`

  `Code s`

  – a code string in a markdown document

- `HRef :: String → String → MarkdownElem`

  `HRef s u`

  – a reference to URL `u` with text `s` in a markdown document

- `Par :: [MarkdownElem] → MarkdownElem`

  `Par md`

  – a paragraph in a markdown document

- `CodeBlock :: String → MarkdownElem`

  `CodeBlock s`

  – a code block in a markdown document

- `UList :: [[MarkdownElem]] → MarkdownElem`

  `UList mds`

  – an unordered list in a markdown document

- `OList :: [[MarkdownElem]] → MarkdownElem`

  `OList mds`

  – an ordered list in a markdown document

- `Quote :: [MarkdownElem] → MarkdownElem`

  `Quote md`

– a quoted paragraph in a markdown document

- `HRule :: MarkdownElem`

  `HRule`

  – a hoirzontal rule in a markdown document

- `Header :: Int → String → MarkdownElem`

  `Header l s`

  – a level `l` header with title `s` in a markdown document

Exported functions:

`fromMarkdownText :: String → [MarkdownElem]`

  Parse markdown document from its textual representation.

`removeEscapes :: String → String`

  Remove the backlash of escaped markdown characters in a string.

`markdownEscapeChars :: String`

  Escape characters supported by markdown.

`markdownText2HTML :: String → [HtmlExp]`

  Translate a markdown text into a (partial) HTML document.

`markdownText2CompleteHTML :: String → String → String`

  Translate a markdown text into a complete HTML text that can be viewed as a standalone document by a browser. The first argument is the title of the document.

`markdownText2LaTeX :: String → String`

  Translate a markdown text into a (partial) LaTeX document. All characters with a special meaning in LaTeX, like dollar or ampersand signs, are quoted.

`markdownText2LaTeXWithFormat :: (String → String) → String → String`

  Translate a markdown text into a (partial) LaTeX document where the first argument is a function to translate the basic text occurring in markdown elements to a LaTeX string. For instance, one can use a translation operation that supports passing mathematical formulas in LaTeX style instead of quoting all special characters.

`markdownText2CompleteLaTeX :: String → String`

  Translate a markdown text into a complete LaTeX document that can be formatted as a standalone document.

`formatMarkdownInputAsPDF :: IO ()`

  Format the standard input (containing markdown text) as PDF.

`formatMarkdownFileAsPDF :: String → IO ()`

  Format a file containing markdown text as PDF.

A.5.8 Library URL

Library for dealing with URLs (Uniform Resource Locators).

Exported functions:

`getContentsOfUrl :: String → IO String`

> Reads the contents of a document located by a URL. This action requires that the program "wget" is in your path, otherwise the implementation must be adapted to the local installation.

A.5.9 Library WUI

A library to support the type-oriented construction of Web User Interfaces (WUIs).
The ideas behind the application and implementation of WUIs are described in a paper that is available via this web page.

Exported types:

`type Rendering = [HtmlExp] → HtmlExp`

> A rendering is a function that combines the visualization of components of a data structure into some HTML expression.

`data WuiHandler`

> A handler for a WUI is an event handler for HTML forms possibly with some specific code attached (for future extensions).

Exported constructors:

`data WuiSpec`

> The type of WUI specifications. The first component are parameters specifying the behavior of this WUI type (rendering, error message, and constraints on inputs). The second component is a "show" function returning an HTML expression for the edit fields and a WUI state containing the CgiRefs to extract the values from the edit fields. The third component is "read" function to extract the values from the edit fields for a given cgi environment (returned as (Just v)). If the value is not legal, Nothing is returned. The second component of the result contains an HTML edit expression together with a WUI state to edit the value again.

Exported constructors:

`data WTree`

> A simple tree structure to demonstrate the construction of WUIs for tree types.

Exported constructors:

- `WLeaf :: a → WTree a`

- `WNode :: [WTree a] → WTree a`

Exported functions:

`wuiHandler2button :: String → WuiHandler → HtmlExp`

   Transform a WUI handler into a submit button with a given label string.

`withRendering :: WuiSpec a → ([HtmlExp] → HtmlExp) → WuiSpec a`

   Puts a new rendering function into a WUI specification.

`withError :: WuiSpec a → String → WuiSpec a`

   Puts a new error message into a WUI specification.

`withCondition :: WuiSpec a → (a → Bool) → WuiSpec a`

   Puts a new condition into a WUI specification.

`transformWSpec :: (a → b,b → a) → WuiSpec a → WuiSpec b`

   Transforms a WUI specification from one type to another.

`adaptWSpec :: (a → b) → WuiSpec a → WuiSpec b`

   Adapt a WUI specification to a new type. For this purpose, the first argument must
   be a transformation mapping values from the old type to the new type. This function
   must be bijective and operationally invertible (i.e., the inverse must be computable by
   narrowing). Otherwise, use `transformWSpec`!

`wHidden :: WuiSpec a`

   A hidden widget for a value that is not shown in the WUI. Usually, this is used in
   components of larger structures, e.g., internal identifiers, data base keys.

`wConstant :: (a → HtmlExp) → WuiSpec a`

   A widget for values that are shown but cannot be modified. The first argument is a
   mapping of the value into a HTML expression to show this value.

`wInt :: WuiSpec Int`

   A widget for editing integer values.

`wString :: WuiSpec String`

   A widget for editing string values.

`wStringSize :: Int → WuiSpec String`

   A widget for editing string values with a size attribute.

`wRequiredString :: WuiSpec String`

   A widget for editing string values that are required to be non-empty.

`wRequiredStringSize :: Int → WuiSpec String`

A widget with a size attribute for editing string values that are required to be non-empty.

`wTextArea :: (Int,Int) → WuiSpec String`

A widget for editing string values in a text area. The argument specifies the height and width of the text area.

`wSelect :: (a → String) → [a] → WuiSpec a`

A widget to select a value from a given list of values. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into strings to be shown in the selection widget.

`wSelectInt :: [Int] → WuiSpec Int`

A widget to select a value from a given list of integers (provided as the argument). The current value should be contained in the value list and is preselected.

`wSelectBool :: String → String → WuiSpec Bool`

A widget to select a Boolean value via a selection box. The arguments are the strings that are shown for the values True and False in the selection box, respectively.

`wCheckBool :: [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a check box. The first argument are HTML expressions that are shown after the check box. The result is True if the box is checked.

`wMultiCheckSelect :: (a → [HtmlExp]) → [a] → WuiSpec [a]`

A widget to select a list of values from a given list of values via check boxes. The current values should be contained in the value list and are preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the check box.

`wRadioSelect :: (a → [HtmlExp]) → [a] → WuiSpec a`

A widget to select a value from a given list of values via a radio button. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the radio button.

`wRadioBool :: [HtmlExp] → [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a radio button. The arguments are the lists of HTML expressions that are shown after the True and False radio buttons, respectively.

`wPair :: WuiSpec a → WuiSpec b → WuiSpec (a,b)`

WUI combinator for pairs.

```
wCons2 :: (a → b → c) → WuiSpec a → WuiSpec b → WuiSpec c
```

WUI combinator for constructors of arity 2. The first argument is the binary constructor.
The second and third arguments are the WUI specifications for the argument types.

```
wTriple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec (a,b,c)
```

WUI combinator for triples.

```
wCons3 :: (a → b → c → d) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d
```

WUI combinator for constructors of arity 3. The first argument is the ternary construc-
tor. The further arguments are the WUI specifications for the argument types.

```
w4Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec (a,b,c,d)
```

WUI combinator for tuples of arity 4.

```
wCons4 :: (a → b → c → d → e) → WuiSpec a → WuiSpec b → WuiSpec c →
WuiSpec d → WuiSpec e
```

WUI combinator for constructors of arity 4. The first argument is the ternary construc-
tor. The further arguments are the WUI specifications for the argument types.

```
w5Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec (a,b,c,d,e)
```

WUI combinator for tuples of arity 5.

```
wCons5 :: (a → b → c → d → e → f) → WuiSpec a → WuiSpec b → WuiSpec c →
WuiSpec d → WuiSpec e → WuiSpec f
```

WUI combinator for constructors of arity 5. The first argument is the ternary construc-
tor. The further arguments are the WUI specifications for the argument types.

```
w6Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec (a,b,c,d,e,f)
```

WUI combinator for tuples of arity 6.

```
wCons6 :: (a → b → c → d → e → f → g) → WuiSpec a → WuiSpec b → WuiSpec c
→ WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g
```

WUI combinator for constructors of arity 6. The first argument is the ternary construc-
tor. The further arguments are the WUI specifications for the argument types.

```
w7Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec (a,b,c,d,e,f,g)
```

WUI combinator for tuples of arity 7.

```
wCons7 :: (a → b → c → d → e → f → g → h) → WuiSpec a → WuiSpec b →
WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h
```

WUI combinator for constructors of arity 7. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
w8Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec (a,b,c,d,e,f,g,h)
```

WUI combinator for tuples of arity 8.

```
wCons8 :: (a → b → c → d → e → f → g → h → i) → WuiSpec a → WuiSpec b
→ WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h →
WuiSpec i
```

WUI combinator for constructors of arity 8. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
w9Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec (a,b,c,d,e,f,g,h,i)
```

WUI combinator for tuples of arity 9.

```
wCons9 :: (a → b → c → d → e → f → g → h → i → j) → WuiSpec a → WuiSpec
b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h →
WuiSpec i → WuiSpec j
```

WUI combinator for constructors of arity 9. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
w10Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e
→ WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec
(a,b,c,d,e,f,g,h,i,j)
```

WUI combinator for tuples of arity 10.

```
wCons10 :: (a → b → c → d → e → f → g → h → i → j → k) → WuiSpec a →
WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g →
WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k
```

WUI combinator for constructors of arity 10. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
w11Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →
WuiSpec (a,b,c,d,e,f,g,h,i,j,k)
```

WUI combinator for tuples of arity 11.

```
wCons11 :: (a → b → c → d → e → f → g → h → i → j → k → l) → WuiSpec a
→ WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g →
WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k → WuiSpec l
```

WUI combinator for constructors of arity 11. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

w12Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →
WuiSpec l → WuiSpec (a,b,c,d,e,f,g,h,i,j,k,l)

WUI combinator for tuples of arity 12.

wCons12 :: (a → b → c → d → e → f → g → h → i → j → k → l → m) →
WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f →
WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k → WuiSpec l →
WuiSpec m

WUI combinator for constructors of arity 12. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

wJoinTuple :: WuiSpec a → WuiSpec b → WuiSpec (a,b)

WUI combinator to combine two tuples into a joint tuple. It is similar to wPair but renders both components as a single tuple provided that the components are already rendered as tuples, i.e., by the rendering function renderTuple. This combinator is useful to define combinators for large tuples.

wList :: WuiSpec a → WuiSpec [a]

WUI combinator for list structures where the list elements are vertically aligned in a table.

wListWithHeadings :: [String] → WuiSpec a → WuiSpec [a]

Add headings to a standard WUI for list structures:

wHList :: WuiSpec a → WuiSpec [a]

WUI combinator for list structures where the list elements are horizontally aligned in a table.

wMatrix :: WuiSpec a → WuiSpec [[a]]

WUI for matrices, i.e., list of list of elements visualized as a matrix.

wMaybe :: WuiSpec Bool → WuiSpec a → a → WuiSpec (Maybe a)

WUI for Maybe values. It is constructed from a WUI for Booleans and a WUI for the potential values. Nothing corresponds to a selection of False in the Boolean WUI. The value WUI is shown after the Boolean WUI.

wCheckMaybe :: WuiSpec a → [HtmlExp] → a → WuiSpec (Maybe a)

A WUI for Maybe values where a check box is used to select Just. The value WUI is shown after the check box.

`wRadioMaybe :: WuiSpec a → [HtmlExp] → [HtmlExp] → a → WuiSpec (Maybe a)`

A WUI for Maybe values where radio buttons are used to switch between Nothing and Just. The value WUI is shown after the radio button WUI.

`wEither :: WuiSpec a → WuiSpec b → WuiSpec (Either a b)`

WUI for union types. Here we provide only the implementation for Either types since other types with more alternatives can be easily reduced to this case.

`wTree :: WuiSpec a → WuiSpec (WTree a)`

WUI for tree types. The rendering specifies the rendering of inner nodes. Leaves are shown with their default rendering.

`renderTuple :: [HtmlExp] → HtmlExp`

Standard rendering of tuples as a table with a single row. Thus, the elements are horizontally aligned.

`renderTaggedTuple :: [String] → [HtmlExp] → HtmlExp`

Standard rendering of tuples with a tag for each element. Thus, each is preceded by a tag, that is set in bold, and all elements are vertically aligned.

`renderList :: [HtmlExp] → HtmlExp`

Standard rendering of lists as a table with a row for each item: Thus, the elements are vertically aligned.

`mainWUI :: WuiSpec a → a → (a → IO HtmlForm) → IO HtmlForm`

Generates an HTML form from a WUI data specification, an initial value and an update form.

`wui2html :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp,WuiHandler)`

Generates HTML editors and a handler from a WUI data specification, an initial value and an update form.

`wuiInForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO HtmlForm) → IO HtmlForm`

Puts a WUI into a HTML form containing "holes" for the WUI and the handler.

`wuiWithErrorForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO HtmlForm) → (HtmlExp,WuiHandler)`

Generates HTML editors and a handler from a WUI data specification, an initial value and an update form. In addition to wui2html, we can provide a skeleton form used to show illegal inputs.

A.5.10 Library WUIjs

A library to support the type-oriented construction of Web User Interfaces (WUIs).
The ideas behind the application and implementation of WUIs are described in a paper that is available via this web page.
In addition to the original library, this version provides also support for JavaScript.

Exported types:

`type Rendering = [HtmlExp] → HtmlExp`

> A rendering is a function that combines the visualization of components of a data structure into some HTML expression.

`data WuiHandler`

> A handler for a WUI is an event handler for HTML forms possibly with some specific JavaScript code attached.

Exported constructors:

`data WuiSpec`

> The type of WUI specifications. The first component are parameters specifying the behavior of this WUI type (rendering, error message, and constraints on inputs). The second component is a "show" function returning an HTML expression for the edit fields and a WUI state containing the CgiRefs to extract the values from the edit fields. The third component is "read" function to extract the values from the edit fields for a given cgi environment (returned as (Just v)). If the value is not legal, Nothing is returned. The second component of the result contains an HTML edit expression together with a WUI state to edit the value again.

Exported constructors:

`data WTree`

> A simple tree structure to demonstrate the construction of WUIs for tree types.

Exported constructors:

- `WLeaf :: a → WTree a`

- `WNode :: [WTree a] → WTree a`

Exported functions:

`wuiHandler2button :: String → WuiHandler → HtmlExp`

    Transform a WUI handler into a submit button with a given label string.

`withRendering :: WuiSpec a → ([HtmlExp] → HtmlExp) → WuiSpec a`

    Puts a new rendering function into a WUI specification.

`withError :: WuiSpec a → String → WuiSpec a`

    Puts a new error message into a WUI specification.

`withCondition :: WuiSpec a → (a → Bool) → WuiSpec a`

    Puts a new condition into a WUI specification.

`withConditionJS :: WuiSpec a → (a → Bool) → WuiSpec a`

    Puts a new JavaScript implementation of the condition into a WUI specification.

`withConditionJSName :: WuiSpec a → (a → Bool,String) → WuiSpec a`

    Puts a new JavaScript implementation of the condition into a WUI specification.

`transformWSpec :: (a → b,b → a) → WuiSpec a → WuiSpec b`

    Transforms a WUI specification from one type to another.

`adaptWSpec :: (a → b) → WuiSpec a → WuiSpec b`

    Adapt a WUI specification to a new type. For this purpose, the first argument must be a transformation mapping values from the old type to the new type. This function must be bijective and operationally invertible (i.e., the inverse must be computable by narrowing). Otherwise, use `transformWSpec`!

`wHidden :: WuiSpec a`

    A hidden widget for a value that is not shown in the WUI. Usually, this is used in components of larger structures, e.g., internal identifiers, data base keys.

`wConstant :: (a → HtmlExp) → WuiSpec a`

    A widget for values that are shown but cannot be modified. The first argument is a mapping of the value into a HTML expression to show this value.

`wInt :: WuiSpec Int`

    A widget for editing integer values.

`wString :: WuiSpec String`

    A widget for editing string values.

`wStringSize :: Int → WuiSpec String`

> A widget for editing string values with a size attribute.

`wRequiredString :: WuiSpec String`

> A widget for editing string values that are required to be non-empty.

`wRequiredStringSize :: Int → WuiSpec String`

> A widget with a size attribute for editing string values that are required to be non-empty.

`wTextArea :: (Int,Int) → WuiSpec String`

> A widget for editing string values in a text area. The argument specifies the height and width of the text area.

`wSelect :: (a → String) → [a] → WuiSpec a`

> A widget to select a value from a given list of values. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into strings to be shown in the selection widget.

`wSelectInt :: [Int] → WuiSpec Int`

> A widget to select a value from a given list of integers (provided as the argument). The current value should be contained in the value list and is preselected.

`wSelectBool :: String → String → WuiSpec Bool`

> A widget to select a Boolean value via a selection box. The arguments are the strings that are shown for the values True and False in the selection box, respectively.

`wCheckBool :: [HtmlExp] → WuiSpec Bool`

> A widget to select a Boolean value via a check box. The first argument are HTML expressions that are shown after the check box. The result is True if the box is checked.

`wMultiCheckSelect :: (a → [HtmlExp]) → [a] → WuiSpec [a]`

> A widget to select a list of values from a given list of values via check boxes. The current values should be contained in the value list and are preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the check box.

`wRadioSelect :: (a → [HtmlExp]) → [a] → WuiSpec a`

> A widget to select a value from a given list of values via a radio button. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the radio button.

`wRadioBool :: [HtmlExp] → [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a radio button. The arguments are the lists of HTML expressions that are shown after the True and False radio buttons, respectively.

wJoinTuple :: WuiSpec a → WuiSpec b → WuiSpec (a,b)

WUI combinator to combine two tuples into a joint tuple. It is similar to wPair but renders both components as a single tuple provided that the components are already rendered as tuples, i.e., by the rendering function renderTuple. This combinator is useful to define combinators for large tuples.

wPair :: WuiSpec a → WuiSpec b → WuiSpec (a,b)

WUI combinator for pairs.

wCons2 :: (a → b → c) → WuiSpec a → WuiSpec b → WuiSpec c

WUI combinator for constructors of arity 2. The first argument is the binary constructor. The second and third arguments are the WUI specifications for the argument types.

wCons2JS :: Maybe ([JSExp] → JSExp) → (a → b → c) → WuiSpec a → WuiSpec b → WuiSpec c

wTriple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec (a,b,c)

WUI combinator for triples.

wCons3 :: (a → b → c → d) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d

WUI combinator for constructors of arity 3. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

wCons3JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d

w4Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec (a,b,c,d)

WUI combinator for tuples of arity 4.

wCons4 :: (a → b → c → d → e) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e

WUI combinator for constructors of arity 4. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

wCons4JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e

w5Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec (a,b,c,d,e)

> WUI combinator for tuples of arity 5.

wCons5 :: (a → b → c → d → e → f) → WuiSpec a → WuiSpec b → WuiSpec c →
WuiSpec d → WuiSpec e → WuiSpec f

> WUI combinator for constructors of arity 5. The first argument is the ternary construc-
> tor. The further arguments are the WUI specifications for the argument types.

wCons5JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f) → WuiSpec a
→ WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f

w6Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec (a,b,c,d,e,f)

> WUI combinator for tuples of arity 6.

wCons6 :: (a → b → c → d → e → f → g) → WuiSpec a → WuiSpec b → WuiSpec c
→ WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g

> WUI combinator for constructors of arity 6. The first argument is the ternary construc-
> tor. The further arguments are the WUI specifications for the argument types.

wCons6JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g) →
WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f →
WuiSpec g

w7Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec (a,b,c,d,e,f,g)

> WUI combinator for tuples of arity 7.

wCons7 :: (a → b → c → d → e → f → g → h) → WuiSpec a → WuiSpec b →
WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h

> WUI combinator for constructors of arity 7. The first argument is the ternary construc-
> tor. The further arguments are the WUI specifications for the argument types.

wCons7JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h)
→ WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f →
WuiSpec g → WuiSpec h

w8Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec (a,b,c,d,e,f,g,h)

WUI combinator for tuples of arity 8.

wCons8 :: (a → b → c → d → e → f → g → h → i) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i

> WUI combinator for constructors of arity 8. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

wCons8JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h → i) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i

w9Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec (a,b,c,d,e,f,g,h,i)

> WUI combinator for tuples of arity 9.

wCons9 :: (a → b → c → d → e → f → g → h → i → j) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j

> WUI combinator for constructors of arity 9. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

wCons9JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h → i → j) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j

w10Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec (a,b,c,d,e,f,g,h,i,j)

> WUI combinator for tuples of arity 10.

wCons10 :: (a → b → c → d → e → f → g → h → i → j → k) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k

> WUI combinator for constructors of arity 10. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

wCons10JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h → i → j → k) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k

```
w11Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →
WuiSpec (a,b,c,d,e,f,g,h,i,j,k)
```

WUI combinator for tuples of arity 11.

```
wCons11 :: (a → b → c → d → e → f → g → h → i → j → k → l) → WuiSpec a
→ WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f → WuiSpec g →
WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k → WuiSpec l
```

WUI combinator for constructors of arity 11. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons11JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h →
i → j → k → l) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e
→ WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →
WuiSpec l
```

```
w12Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e →
WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k →
WuiSpec l → WuiSpec (a,b,c,d,e,f,g,h,i,j,k,l)
```

WUI combinator for tuples of arity 12.

```
wCons12 :: (a → b → c → d → e → f → g → h → i → j → k → l → m) →
WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec f →
WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j → WuiSpec k → WuiSpec l →
WuiSpec m
```

WUI combinator for constructors of arity 12. The first argument is the ternary constructor. The further arguments are the WUI specifications for the argument types.

```
wCons12JS :: Maybe ([JSExp] → JSExp) → (a → b → c → d → e → f → g → h
→ i → j → k → l → m) → WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d →
WuiSpec e → WuiSpec f → WuiSpec g → WuiSpec h → WuiSpec i → WuiSpec j →
WuiSpec k → WuiSpec l → WuiSpec m
```

```
wList :: WuiSpec a → WuiSpec [a]
```

WUI combinator for list structures where the list elements are vertically aligned in a table.

```
wListWithHeadings :: [String] → WuiSpec a → WuiSpec [a]
```

Add headings to a standard WUI for list structures:

```
wHList :: WuiSpec a → WuiSpec [a]
```

WUI combinator for list structures where the list elements are horizontally aligned in a table.

wMatrix :: WuiSpec a → WuiSpec [[a]]

WUI for matrices, i.e., list of list of elements visualized as a matrix.

wMaybe :: WuiSpec Bool → WuiSpec a → a → WuiSpec (Maybe a)

WUI for Maybe values. It is constructed from a WUI for Booleans and a WUI for the potential values. Nothing corresponds to a selection of False in the Boolean WUI. The value WUI is shown after the Boolean WUI.

wCheckMaybe :: WuiSpec a → [HtmlExp] → a → WuiSpec (Maybe a)

A WUI for Maybe values where a check box is used to select Just. The value WUI is shown after the check box.

wRadioMaybe :: WuiSpec a → [HtmlExp] → [HtmlExp] → a → WuiSpec (Maybe a)

A WUI for Maybe values where radio buttons are used to switch between Nothing and Just. The value WUI is shown after the radio button WUI.

wEither :: WuiSpec a → WuiSpec b → WuiSpec (Either a b)

WUI for union types. Here we provide only the implementation for Either types since other types with more alternatives can be easily reduced to this case.

wTree :: WuiSpec a → WuiSpec (WTree a)

WUI for tree types. The rendering specifies the rendering of inner nodes. Leaves are shown with their default rendering.

renderTuple :: [HtmlExp] → HtmlExp

Standard rendering of tuples as a table with a single row. Thus, the elements are horizontally aligned.

renderTaggedTuple :: [String] → [HtmlExp] → HtmlExp

Standard rendering of tuples with a tag for each element. Thus, each is preceded by a tag, that is set in bold, and all elements are vertically aligned.

renderList :: [HtmlExp] → HtmlExp

Standard rendering of lists as a table with a row for each item: Thus, the elements are vertically aligned.

mainWUI :: WuiSpec a → a → (a → IO HtmlForm) → IO HtmlForm

Generates an HTML form from a WUI data specification, an initial value and an update form.

`wui2html :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp,WuiHandler)`

> Generates HTML editors and a handler from a WUI data specification, an initial value and an update form.

`wuiInForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO HtmlForm) → IO HtmlForm`

> Puts a WUI into a HTML form containing "holes" for the WUI and the handler.

`wuiWithErrorForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO HtmlForm) → (HtmlExp,WuiHandler)`

> Generates HTML editors and a handler from a WUI data specification, an initial value and an update form. In addition to wui2html, we can provide a skeleton form used to show illegal inputs.

A.5.11 Library XML

Library for processing XML data.
Warning: the structure of this library is not stable and might be changed in the future!

Exported types:

`data XmlExp`

> The data type for representing XML expressions.

> *Exported constructors:*

> - `XText :: String → XmlExp`
>   `XText`
>
>   – a text string (PCDATA)

> - `XElem :: String → [(String,String)] → [XmlExp] → XmlExp`
>   `XElem`
>
>   – an XML element with tag field, attributes, and a list of XML elements as contents

`data Encoding`

> The data type for encodings used in the XML document.

> *Exported constructors:*

> - `StandardEnc :: Encoding`

> - `Iso88591Enc :: Encoding`

`data XmlDocParams`

The data type for XML document parameters.

Exported constructors:

- Enc :: Encoding → XmlDocParams

  Enc

    – the encoding for a document

- DtdUrl :: String → XmlDocParams

  DtdUrl

    – the url of the DTD for a document

Exported functions:

tagOf :: XmlExp → String

Returns the tag of an XML element (or empty for a textual element).

elemsOf :: XmlExp → [XmlExp]

Returns the child elements an XML element.

textOf :: [XmlExp] → String

Extracts the textual contents of a list of XML expressions. Useful auxiliary function when transforming XML expressions into other data structures.

For instance, textOf [XText "xy", XElem "a" [] [], XText "bc"] == "xy bc"

textOfXml :: [XmlExp] → String

Included for backward compatibility, better use textOf!

xtxt :: String → XmlExp

Basic text (maybe containing special XML chars).

xml :: String → [XmlExp] → XmlExp

XML element without attributes.

writeXmlFile :: String → XmlExp → IO ()

Writes a file with a given XML document.

writeXmlFileWithParams :: String → [XmlDocParams] → XmlExp → IO ()

Writes a file with a given XML document and XML parameters.

showXmlDoc :: XmlExp → String

Show an XML document in indented format as a string.

```
showXmlDocWithParams :: [XmlDocParams] → XmlExp → String
```

```
readXmlFile :: String → IO XmlExp
```

Reads a file with an XML document and returns the corresponding XML expression.

```
readUnsafeXmlFile :: String → IO (Maybe XmlExp)
```

Tries to read a file with an XML document and returns the corresponding XML expression, if possible. If file or parse errors occur, Nothing is returned.

```
readFileWithXmlDocs :: String → IO [XmlExp]
```

Reads a file with an arbitrary sequence of XML documents and returns the list of corresponding XML expressions.

```
parseXmlString :: String → [XmlExp]
```

Transforms an XML string into a list of XML expressions. If the XML string is a well structured document, the list of XML expressions should contain exactly one element.

```
updateXmlFile :: (XmlExp → XmlExp) → String → IO ()
```

An action that updates the contents of an XML file by some transformation on the XML document.

A.5.12 Library XmlConv

Provides type-based combinators to construct XML converters. Arbitrary XML data can be represented as algebraic datatypes and vice versa. See here[11] for a description of this library.

Exported types:

```
type XmlReads a = ([(String,String)],[XmlExp]) → (a,([(String,String)],[XmlExp]))
```

Type of functions that consume some XML data to compute a result

```
type XmlShows a = a → ([(String,String)],[XmlExp]) → ([(String,String)],[XmlExp])
```

Type of functions that extend XML data corresponding to a given value

```
type XElemConv a = XmlConv Repeatable Elem a
```

Type of converters for XML elements

```
type XAttrConv a = XmlConv NotRepeatable NoElem a
```

Type of converters for attributes

```
type XPrimConv a = XmlConv NotRepeatable NoElem a
```

---
[11]http://www-ps.informatik.uni-kiel.de/~sebf/projects/xmlconv/

Type of converters for primitive values

```
type XOptConv a = XmlConv NotRepeatable NoElem a
```

Type of converters for optional values

```
type XRepConv a = XmlConv NotRepeatable NoElem a
```

Type of converters for repetitions

Exported functions:

```
xmlReads :: XmlConv a b c → ([(String,String)],[XmlExp]) →
(c,([(String,String)],[XmlExp]))
```

Takes an XML converter and returns a function that consumes XML data and returns
the remaining data along with the result.

```
xmlShows :: XmlConv a b c → c → ([(String,String)],[XmlExp]) →
([(String,String)],[XmlExp])
```

Takes an XML converter and returns a function that extends XML data with the rep-
resentation of a given value.

```
xmlRead :: XmlConv a Elem b → XmlExp → b
```

Takes an XML converter and an XML expression and returns a corresponding Curry
value.

```
xmlShow :: XmlConv a Elem b → b → XmlExp
```

Takes an XML converter and a value and returns a corresponding XML expression.

```
int :: XmlConv NotRepeatable NoElem Int
```

Creates an XML converter for integer values. Integer values must not be used in repe-
titions and do not represent XML elements.

```
float :: XmlConv NotRepeatable NoElem Float
```

Creates an XML converter for float values. Float values must not be used in repetitions
and do not represent XML elements.

```
char :: XmlConv NotRepeatable NoElem Char
```

Creates an XML converter for character values. Character values must not be used in
repetitions and do not represent XML elements.

```
string :: XmlConv NotRepeatable NoElem String
```

Creates an XML converter for string values. String values must not be used in repetitions
and do not represent XML elements.

```
(!) :: XmlConv a b c → XmlConv a b c → XmlConv a b c
```

Parallel composition of XML converters.

```
element :: String → XmlConv a b c → XmlConv Repeatable Elem c
```

Takes an arbitrary XML converter and returns a converter representing an XML element that contains the corresponding data. XML elements may be used in repetitions.

```
empty :: a → XmlConv NotRepeatable NoElem a
```

Takes a value and returns an XML converter for this value which is not represented as XML data. Empty XML data must not be used in repetitions and does not represent an XML element.

```
attr :: String → (String → a,a → String) → XmlConv NotRepeatable NoElem a
```

Takes a name and string conversion functions and returns an XML converter that represents an attribute. Attributes must not be used in repetitions and do not represent an XML element.

```
adapt :: (a → b,b → a) → XmlConv c d a → XmlConv c d b
```

Converts between arbitrary XML converters for different types.

```
opt :: XmlConv a b c → XmlConv NotRepeatable NoElem (Maybe c)
```

Creates a converter for arbitrary optional XML data. Optional XML data must not be used in repetitions and does not represent an XML element.

```
rep :: XmlConv Repeatable a b → XmlConv NotRepeatable NoElem [b]
```

Takes an XML converter representing repeatable data and returns an XML converter that represents repetitions of this data. Repetitions must not be used in other repetitions and do not represent XML elements.

```
aInt :: String → XmlConv NotRepeatable NoElem Int
```

Creates an XML converter for integer attributes. Integer attributes must not be used in repetitions and do not represent XML elements.

```
aFloat :: String → XmlConv NotRepeatable NoElem Float
```

Creates an XML converter for float attributes. Float attributes must not be used in repetitions and do not represent XML elements.

```
aChar :: String → XmlConv NotRepeatable NoElem Char
```

Creates an XML converter for character attributes. Character attributes must not be used in repetitions and do not represent XML elements.

```
aString :: String → XmlConv NotRepeatable NoElem String
```

Creates an XML converter for string attributes. String attributes must not be used in repetitions and do not represent XML elements.

`aBool :: String → String → String → XmlConv NotRepeatable NoElem Bool`

Creates an XML converter for boolean attributes. Boolean attributes must not be used in repetitions and do not represent XML elements.

`eInt :: String → XmlConv Repeatable Elem Int`

Creates an XML converter for integer elements. Integer elements may be used in repetitions.

`eFloat :: String → XmlConv Repeatable Elem Float`

Creates an XML converter for float elements. Float elements may be used in repetitions.

`eChar :: String → XmlConv Repeatable Elem Char`

Creates an XML converter for character elements. Character elements may be used in repetitions.

`eString :: String → XmlConv Repeatable Elem String`

Creates an XML converter for string elements. String elements may be used in repetitions.

`eBool :: String → String → XmlConv Repeatable Elem Bool`

Creates an XML converter for boolean elements. Boolean elements may be used in repetitions.

`eEmpty :: String → a → XmlConv Repeatable Elem a`

Takes a name and a value and creates an empty XML element that represents the given value. The created element may be used in repetitions.

`eOpt :: String → XmlConv a b c → XmlConv Repeatable Elem (Maybe c)`

Creates an XML converter that represents an element containing optional XML data. The created element may be used in repetitions.

`eRep :: String → XmlConv Repeatable a b → XmlConv Repeatable Elem [b]`

Creates an XML converter that represents an element containing repeated XML data. The created element may be used in repetitions.

`seq1 :: (a → b) → XmlConv c d a → XmlConv c NoElem b`

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

`repSeq1 :: (a → b) → XmlConv Repeatable c a → XmlConv NotRepeatable NoElem [b]`

285

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions but does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

eSeq1 :: String → (a → b) → XmlConv c d a → XmlConv Repeatable Elem b

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

eRepSeq1 :: String → (a → b) → XmlConv Repeatable c a → XmlConv Repeatable Elem [b]

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

seq2 :: (a → b → c) → XmlConv d e a → XmlConv f g b → XmlConv NotRepeatable NoElem c

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

repSeq2 :: (a → b → c) → XmlConv Repeatable d a → XmlConv Repeatable e b → XmlConv NotRepeatable NoElem [c]

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

eSeq2 :: String → (a → b → c) → XmlConv d e a → XmlConv f g b → XmlConv Repeatable Elem c

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

eRepSeq2 :: String → (a → b → c) → XmlConv Repeatable d a → XmlConv Repeatable e b → XmlConv Repeatable Elem [c]

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

seq3 :: (a → b → c → d) → XmlConv e f a → XmlConv g h b → XmlConv i j c → XmlConv NotRepeatable NoElem d

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

repSeq3 :: (a → b → c → d) → XmlConv Repeatable e a → XmlConv Repeatable f b → XmlConv Repeatable g c → XmlConv NotRepeatable NoElem [d]

> Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

eSeq3 :: String → (a → b → c → d) → XmlConv e f a → XmlConv g h b → XmlConv i j c → XmlConv Repeatable Elem d

> Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

eRepSeq3 :: String → (a → b → c → d) → XmlConv Repeatable e a → XmlConv Repeatable f b → XmlConv Repeatable g c → XmlConv Repeatable Elem [d]

> Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

seq4 :: (a → b → c → d → e) → XmlConv f g a → XmlConv h i b → XmlConv j k c → XmlConv l m d → XmlConv NotRepeatable NoElem e

> Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

repSeq4 :: (a → b → c → d → e) → XmlConv Repeatable f a → XmlConv Repeatable g b → XmlConv Repeatable h c → XmlConv Repeatable i d → XmlConv NotRepeatable NoElem [e]

> Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

eSeq4 :: String → (a → b → c → d → e) → XmlConv f g a → XmlConv h i b → XmlConv j k c → XmlConv l m d → XmlConv Repeatable Elem e

> Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

eRepSeq4 :: String → (a → b → c → d → e) → XmlConv Repeatable f a → XmlConv Repeatable g b → XmlConv Repeatable h c → XmlConv Repeatable i d → XmlConv Repeatable Elem [e]

> Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

```
seq5 :: (a → b → c → d → e → f) → XmlConv g h a → XmlConv i j b → XmlConv
k l c → XmlConv m n d → XmlConv o p e → XmlConv NotRepeatable NoElem f
```

> Creates an XML converter representing a sequence of arbitrary XML data. The sequence
> must not be used in repetitions and does not represent an XML element.

```
repSeq5 :: (a → b → c → d → e → f) → XmlConv Repeatable g a → XmlConv
Repeatable h b → XmlConv Repeatable i c → XmlConv Repeatable j d → XmlConv
Repeatable k e → XmlConv NotRepeatable NoElem [f]
```

> Creates an XML converter that represents a repetition of a sequence of repeatable XML
> data. The repetition may be used in other repetitions and does not represent an XML
> element. This combinator is provided because converters for repeatable sequences cannot
> be constructed by the seq combinators.

```
eSeq5 :: String → (a → b → c → d → e → f) → XmlConv g h a → XmlConv i j b
→ XmlConv k l c → XmlConv m n d → XmlConv o p e → XmlConv Repeatable Elem f
```

> Creates an XML converter for compound values represented as an XML element with
> children that correspond to the values components. The element can be used in repeti-
> tions.

```
eRepSeq5 :: String → (a → b → c → d → e → f) → XmlConv Repeatable g a →
XmlConv Repeatable h b → XmlConv Repeatable i c → XmlConv Repeatable j d →
XmlConv Repeatable k e → XmlConv Repeatable Elem [f]
```

> Creates an XML converter for repetitions of sequences represented as an XML element
> that can be used in repetitions.

```
seq6 :: (a → b → c → d → e → f → g) → XmlConv h i a → XmlConv j k b →
XmlConv l m c → XmlConv n o d → XmlConv p q e → XmlConv r s f → XmlConv
NotRepeatable NoElem g
```

> Creates an XML converter representing a sequence of arbitrary XML data. The sequence
> must not be used in repetitions and does not represent an XML element.

```
repSeq6 :: (a → b → c → d → e → f → g) → XmlConv Repeatable h a → XmlConv
Repeatable i b → XmlConv Repeatable j c → XmlConv Repeatable k d → XmlConv
Repeatable l e → XmlConv Repeatable m f → XmlConv NotRepeatable NoElem [g]
```

> Creates an XML converter that represents a repetition of a sequence of repeatable XML
> data. The repetition may be used in other repetitions and does not represent an XML
> element. This combinator is provided because converters for repeatable sequences cannot
> be constructed by the seq combinators.

```
eSeq6 :: String → (a → b → c → d → e → f → g) → XmlConv h i a → XmlConv j
k b → XmlConv l m c → XmlConv n o d → XmlConv p q e → XmlConv r s f → XmlConv
Repeatable Elem g
```

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

eRepSeq6 :: String → (a → b → c → d → e → f → g) → XmlConv Repeatable h a → XmlConv Repeatable i b → XmlConv Repeatable j c → XmlConv Repeatable k d → XmlConv Repeatable l e → XmlConv Repeatable m f → XmlConv Repeatable Elem [g]

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

A.6 Libraries for Meta-Programming

A.6.1 Library AbstractCurry.Types

This library contains a definition for representing Curry programs in Curry and an I/O action to read Curry programs and transform them into this abstract representation.
Note this defines a slightly new format for AbstractCurry in comparison to the first proposal of 2003.
Assumption: an abstract Curry program is stored in file with extension .acy

Exported types:

type MName = String

A module name.

type QName = (String,String)

The data type for representing qualified names. In AbstractCurry all names are qualified to avoid name clashes. The first component is the module name and the second component the unqualified name as it occurs in the source program. An exception are locally defined names where the module name is the empty string (to avoid name clashes with a globally defined name).

type CTVarIName = (Int,String)

The type for representing type variables. They are represented by (i,n) where i is a type variable index which is unique inside a function and n is a name (if possible, the name written in the source program).

type CField a = ((String,String),a)

Labeled record fields

type Arity = Int

Function arity

type CVarIName = (Int,String)

Data types for representing object variables. Object variables occurring in expressions are represented by (Var i) where i is a variable index.

data CVisibility

Data type to specify the visibility of various entities.

Exported constructors:

- Public :: CVisibility

- Private :: CVisibility

data CurryProg

Data type for representing a Curry module in the intermediate form. A value of this data type has the form

```
(CurryProg modname imports typedecls functions opdecls)
```

where modname: name of this module, imports: list of modules names that are imported, typedecls: Type declarations functions: Function declarations opdecls: Operator precedence declarations

Exported constructors:

- CurryProg :: String $\rightarrow$ [String] $\rightarrow$ [CTypeDecl] $\rightarrow$ [CFuncDecl] $\rightarrow$ [COpDecl] $\rightarrow$ CurryProg

data CTypeDecl

Data type for representing definitions of algebraic data types and type synonyms.

A data type definition of the form

```
data t x1...xn = ...| c t1....tkc |...
```

is represented by the Curry term

```
(CType t v [i1,...,in] [...(CCons c kc v [t1,...,tkc])...])
```

where each `ij` is the index of the type variable `xj`.

Note: the type variable indices are unique inside each type declaration and are usually numbered from 0

Thus, a data type declaration consists of the name of the data type, a list of type parameters and a list of constructor declarations.

Exported constructors:

- CType :: (String,String) → CVisibility → [(Int,String)] → [CConsDecl] → CTypeDecl

- CTypeSyn :: (String,String) → CVisibility → [(Int,String)] → CTypeExpr → CTypeDecl

- CNewType :: (String,String) → CVisibility → [(Int,String)] → CConsDecl → CTypeDecl

data CConsDecl

A constructor declaration consists of the name of the constructor and a list of the argument types of the constructor. The arity equals the number of types.

Exported constructors:

- CCons :: (String,String) → CVisibility → [CTypeExpr] → CConsDecl

- CRecord :: (String,String) → CVisibility → [CFieldDecl] → CConsDecl

data CFieldDecl

A record field declaration consists of the name of the the label, the visibility and its corresponding type.

Exported constructors:

- CField :: (String,String) → CVisibility → CTypeExpr → CFieldDecl

data CTypeExpr

Type expression. A type expression is either a type variable, a function type, or a type constructor application.

Note: the names of the predefined type constructors are "Int", "Float", "Bool", "Char", "IO", "()" (unit type), "(,...,)" (tuple types), "[]" (list type)

Exported constructors:

- CTVar :: (Int,String) → CTypeExpr

- CFuncType :: CTypeExpr → CTypeExpr → CTypeExpr

- CTCons :: (String,String) → [CTypeExpr] → CTypeExpr

data COpDecl

Data type for operator declarations. An operator declaration "fix p n" in Curry corresponds to the AbstractCurry term (COp n fix p).

Exported constructors:

- COp :: (String,String) → CFixity → Int → COpDecl

data CFixity

Data type for operator associativity

Exported constructors:

- CInfixOp :: CFixity

- CInfixlOp :: CFixity

- CInfixrOp :: CFixity

data CFuncDecl

Data type for representing function declarations.

A function declaration in AbstractCurry is a term of the form

(CFunc name arity visibility type (CRules eval [CRule rule1,...,rulek]))

and represents the function `name` defined by the rules `rule1,...,rulek`.

Note: the variable indices are unique inside each rule

Thus, a function declaration consists of the name, arity, type, and a list of rules.

A function declaration with the constructor `CmtFunc` is similarly to `CFunc` but has a comment as an additional first argument. This comment could be used by pretty printers that generate a readable Curry program containing documentation comments.

Exported constructors:

- CFunc :: (String,String) → Int → CVisibility → CTypeExpr → [CRule] → CFuncDecl

- CmtFunc :: String → (String,String) → Int → CVisibility → CTypeExpr → [CRule] → CFuncDecl

data CRule

The general form of a function rule. It consists of a list of patterns (left-hand side) and the right-hand side for these patterns.

Exported constructors:

- CRule :: [CPattern] → CRhs → CRule

data CRhs

Right-hand-side of a `CRule` or a `case` expression. It is either a simple unconditional right-hand side or a list of guards with their corresponding right-hand sides, and a list of local declarations.

Exported constructors:

- CSimpleRhs :: CExpr → [CLocalDecl] → CRhs

- CGuardedRhs :: [(CExpr,CExpr)] → [CLocalDecl] → CRhs

data CLocalDecl

Data type for representing local (let/where) declarations

Exported constructors:

- CLocalFunc :: CFuncDecl → CLocalDecl

- CLocalPat :: CPattern → CRhs → CLocalDecl

- CLocalVars :: [(Int,String)] → CLocalDecl

data CPattern

Data type for representing pattern expressions.

Exported constructors:

- CPVar :: (Int,String) → CPattern

- CPLit :: CLiteral → CPattern

- CPComb :: (String,String) → [CPattern] → CPattern

- CPAs :: (Int,String) → CPattern → CPattern

- CPFuncComb :: (String,String) → [CPattern] → CPattern

- CPLazy :: CPattern → CPattern

- CPRecord :: (String,String) → [((String,String),CPattern)] → CPattern

data CExpr

Data type for representing Curry expressions.

Exported constructors:

- CVar :: (Int,String) → CExpr

- CLit :: CLiteral → CExpr

- CSymbol :: (String,String) → CExpr

- CApply :: CExpr → CExpr → CExpr

- CLambda :: [CPattern] → CExpr → CExpr

293

- CLetDecl :: [CLocalDecl] → CExpr → CExpr

- CDoExpr :: [CStatement] → CExpr

- CListComp :: CExpr → [CStatement] → CExpr

- CCase :: CCaseType → CExpr → [(CPattern,CRhs)] → CExpr

- CTyped :: CExpr → CTypeExpr → CExpr

- CRecConstr :: (String,String) → [((String,String),CExpr)] → CExpr

- CRecUpdate :: CExpr → [((String,String),CExpr)] → CExpr

data CLiteral

Data type for representing literals occurring in an expression. It is either an integer, a float, or a character constant.

Exported constructors:

- CIntc :: Int → CLiteral

- CFloatc :: Float → CLiteral

- CCharc :: Char → CLiteral

- CStringc :: String → CLiteral

data CStatement

Data type for representing statements in do expressions and list comprehensions.

Exported constructors:

- CSExpr :: CExpr → CStatement

- CSPat :: CPattern → CExpr → CStatement

- CSLet :: [CLocalDecl] → CStatement

data CCaseType

Type of case expressions

Exported constructors:

- CRigid :: CCaseType

- CFlex :: CCaseType

Exported functions:

`version :: String`

> Current version of AbstractCurry

`preludeName :: String`

> The name of the standard prelude.

`pre :: String → (String,String)`

> Converts a string into a qualified name of the Prelude.

A.6.2 Library AbstractCurry.Files

This library defines various I/O actions to read Curry programs and transform them into the AbstractCurry representation and to write AbstractCurry files.
Assumption: an abstract Curry program is stored in file with extension `.acy` in the subdirectory `.curry`

Exported functions:

`readCurry :: String → IO CurryProg`

> I/O action which parses a Curry program and returns the corresponding typed Abstract Curry program. Thus, the argument is the file name without suffix ".curry" or ".lcurry") and the result is a Curry term representing this program.

`readCurryWithImports :: String → IO [CurryProg]`

> Read an AbstractCurry file with all its imports.

`tryReadCurryWithImports :: String → IO (Either [String] [CurryProg])`

`tryReadCurryFile :: String → IO (Either String CurryProg)`

`tryParse :: String → IO (Either String CurryProg)`

> Try to parse an AbstractCurry file.

`readUntypedCurry :: String → IO CurryProg`

> I/O action which parses a Curry program and returns the corresponding untyped Abstract Curry program. Thus, the argument is the file name without suffix ".curry" or ".lcurry") and the result is a Curry term representing this program.

`readCurryWithParseOptions :: String → FrontendParams → IO CurryProg`

I/O action which reads a typed Curry program from a file (with extension ".acy") with respect to some parser options. This I/O action is used by the standard action `readCurry`. It is currently predefined only in Curry2Prolog.

`readUntypedCurryWithParseOptions :: String → FrontendParams → IO CurryProg`

I/O action which reads an untyped Curry program from a file (with extension ".uacy") with respect to some parser options. For more details see function **readCurryWithParseOptions**

`abstractCurryFileName :: String → String`

Transforms a name of a Curry program (with or without suffix ".curry" or ".lcurry") into the name of the file containing the corresponding AbstractCurry program.

`untypedAbstractCurryFileName :: String → String`

Transforms a name of a Curry program (with or without suffix ".curry" or ".lcurry") into the name of the file containing the corresponding untyped AbstractCurry program.

`readAbstractCurryFile :: String → IO CurryProg`

I/O action which reads an AbstractCurry program from a file in ".acy" format. In contrast to `readCurry`, this action does not parse a source program. Thus, the argument must be the name of an existing file (with suffix ".acy") containing an AbstractCurry program in ".acy" format and the result is a Curry term representing this program. It is currently predefined only in Curry2Prolog.

`tryReadACYFile :: String → IO (Maybe CurryProg)`

Tries to read an AbstractCurry file and returns

- Left err , where err specifies the error occurred
- Right prog, where prog is the AbstractCurry program

`writeAbstractCurryFile :: String → CurryProg → IO ()`

Writes an AbstractCurry program into a file in ".acy" format. The first argument must be the name of the target file (with suffix ".acy").

A.6.3 Library AbstractCurry.Select

This library provides some useful operations to select components in AbstractCurry programs, i.e., it provides a collection of selector functions for AbstractCurry.

Exported functions:

`progName :: CurryProg → String`

`imports :: CurryProg → [String]`

      Returns the imports (module names) of a given Curry program.

`functions :: CurryProg → [CFuncDecl]`

      Returns the function declarations of a given Curry program.

`constructors :: CurryProg → [CConsDecl]`

      Returns all constructors of given Curry program.

`types :: CurryProg → [CTypeDecl]`

      Returns the type declarations of a given Curry program.

`publicFuncNames :: CurryProg → [(String,String)]`

      Returns the names of all visible functions in given Curry program.

`publicConsNames :: CurryProg → [(String,String)]`

      Returns the names of all visible constructors in given Curry program.

`publicTypeNames :: CurryProg → [(String,String)]`

      Returns the names of all visible types in given Curry program.

`typeName :: CTypeDecl → (String,String)`

      Returns the name of a given type declaration

`typeVis :: CTypeDecl → CVisibility`

      Returns the visibility of a given type declaration

`typeCons :: CTypeDecl → [CConsDecl]`

      Returns the constructors of a given type declaration.

`consName :: CConsDecl → (String,String)`

      Returns the name of a given constructor declaration.

`consVis :: CConsDecl → CVisibility`

      Returns the visibility of a given constructor declaration.

`isBaseType :: CTypeExpr → Bool`

Returns true if the type expression is a base type.

`isPolyType :: CTypeExpr → Bool`

Returns true if the type expression contains type variables.

`isFunctionalType :: CTypeExpr → Bool`

Returns true if the type expression is a functional type.

`isIOType :: CTypeExpr → Bool`

Returns true if the type expression is (IO t).

`isIOReturnType :: CTypeExpr → Bool`

Returns true if the type expression is (IO t) with t/=() and t is not functional

`argTypes :: CTypeExpr → [CTypeExpr]`

Returns all argument types from a functional type

`resultType :: CTypeExpr → CTypeExpr`

Return the result type from a (nested) functional type

`tvarsOfType :: CTypeExpr → [(Int,String)]`

Returns all type variables occurring in a type expression.

`tconsOfType :: CTypeExpr → [(String,String)]`

Returns all type constructors used in the given type.

`modsOfType :: CTypeExpr → [String]`

Returns all modules used in the given type.

`funcName :: CFuncDecl → (String,String)`

Returns the name of a given function declaration.

`funcArity :: CFuncDecl → Int`

`funcComment :: CFuncDecl → String`

Returns the documentation comment of a given function declaration.

`funcVis :: CFuncDecl → CVisibility`

Returns the visibility of a given function declaration.

`funcType :: CFuncDecl → CTypeExpr`

Returns the type of a given function declaration.

`funcRules :: CFuncDecl → [CRule]`

Returns the rules of a given function declaration.

`ruleRHS :: CRule → CRhs`

Returns the right-hand side of a rules.

`ldeclsOfRule :: CRule → [CLocalDecl]`

Returns the local declarations of given rule.

`varsOfPat :: CPattern → [(Int,String)]`

Returns list of all variables occurring in a pattern. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfExp :: CExpr → [(Int,String)]`

Returns list of all variables occurring in an expression. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfRhs :: CRhs → [(Int,String)]`

Returns list of all variables occurring in a right-hand side. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfStat :: CStatement → [(Int,String)]`

Returns list of all variables occurring in a statement. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfLDecl :: CLocalDecl → [(Int,String)]`

Returns list of all variables occurring in a local declaration. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfFDecl :: CFuncDecl → [(Int,String)]`

Returns list of all variables occurring in a function declaration. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`varsOfRule :: CRule → [(Int,String)]`

Returns list of all variables occurring in a rule. Each occurrence corresponds to one element, i.e., the list might contain multiple elements.

`funcNamesOfLDecl :: CLocalDecl → [(String,String)]`

`funcNamesOfFDecl :: CFuncDecl → [(String,String)]`

`funcNamesOfStat :: CStatement → [(String,String)]`

`isPrelude :: String → Bool`

Tests whether a module name is the prelude.

A.6.4   Library AbstractCurry.Build

This library provides some useful operations to write programs that generate AbstractCurry programs in a more compact and readable way.

Exported functions:

(~>) :: CTypeExpr → CTypeExpr → CTypeExpr

> A function type.

baseType :: (String,String) → CTypeExpr

> A base type.

listType :: CTypeExpr → CTypeExpr

> Constructs a list type from an element type.

tupleType :: [CTypeExpr] → CTypeExpr

> Constructs a tuple type from list of component types.

ioType :: CTypeExpr → CTypeExpr

> Constructs an IO type from a type.

maybeType :: CTypeExpr → CTypeExpr

> Constructs a Maybe type from element type.

stringType :: CTypeExpr

> The type expression of the String type.

intType :: CTypeExpr

> The type expression of the Int type.

floatType :: CTypeExpr

> The type expression of the Float type.

boolType :: CTypeExpr

> The type expression of the Bool type.

charType :: CTypeExpr

> The type expression of the Char type.

unitType :: CTypeExpr

> The type expression of the unit type.

```
dateType :: CTypeExpr
```

The type expression of the Time.CalendarTime type.

```
cfunc :: (String,String) → Int → CVisibility → CTypeExpr → [CRule] →
CFuncDecl
```

Constructs a function declaration from a given qualified function name, arity, visibility, type expression and list of defining rules.

```
cmtfunc :: String → (String,String) → Int → CVisibility → CTypeExpr → [CRule]
→ CFuncDecl
```

Constructs a function declaration from a given comment, qualified function name, arity, visibility, type expression and list of defining rules.

```
simpleRule :: [CPattern] → CExpr → CRule
```

Constructs a simple rule with a pattern list and an unconditional right-hand side.

```
simpleRuleWithLocals :: [CPattern] → CExpr → [CLocalDecl] → CRule
```

Constructs a simple rule with a pattern list, an unconditional right-hand side, and local declarations.

```
guardedRule :: [CPattern] → [(CExpr,CExpr)] → [CLocalDecl] → CRule
```

Constructs a rule with a possibly guarded right-hand side and local declarations. A simple right-hand side is constructed if there is only one True condition.

```
noGuard :: CExpr → (CExpr,CExpr)
```

Constructs a guarded expression with the trivial guard.

```
applyF :: (String,String) → [CExpr] → CExpr
```

An application of a qualified function name to a list of arguments.

```
applyE :: CExpr → [CExpr] → CExpr
```

An application of an expression to a list of arguments.

```
constF :: (String,String) → CExpr
```

A constant, i.e., an application without arguments.

```
applyV :: (Int,String) → [CExpr] → CExpr
```

An application of a variable to a list of arguments.

```
applyJust :: CExpr → CExpr
```

```
applyMaybe :: CExpr → CExpr → CExpr → CExpr
```


```
tupleExpr :: [CExpr] → CExpr
```

Constructs a tuple expression from list of component expressions.

```
letExpr :: [CLocalDecl] → CExpr → CExpr
```


```
cBranch :: CPattern → CExpr → (CPattern,CRhs)
```

Constructs from a pattern and an expression a branch for a case expression.

```
tuplePattern :: [CPattern] → CPattern
```

Constructs a tuple pattern from list of component patterns.

```
pVars :: Int → [CPattern]
```

Constructs, for given n, a list of n PVars starting from 0.

```
pInt :: Int → CPattern
```

Converts an integer into an AbstractCurry expression.

```
pFloat :: Float → CPattern
```

Converts a float into an AbstractCurry expression.

```
pChar :: Char → CPattern
```

Converts a character into a pattern.

```
pNil :: CPattern
```

Constructs an empty list pattern.

```
listPattern :: [CPattern] → CPattern
```

Constructs a list pattern from list of component patterns.

```
stringPattern :: String → CPattern
```

Converts a string into a pattern representing this string.

```
list2ac :: [CExpr] → CExpr
```

Converts a list of AbstractCurry expressions into an AbstractCurry representation of this list.

```
cInt :: Int → CExpr
```

Converts an integer into an AbstractCurry expression.

```
cFloat :: Float → CExpr
```

> Converts a float into an AbstractCurry expression.

```
cChar :: Char → CExpr
```

> Converts a character into an AbstractCurry expression.

```
string2ac :: String → CExpr
```

> Converts a string into an AbstractCurry represention of this string.

```
toVar :: Int → CExpr
```

> Converts an index i into a variable named xi.

```
cvar :: String → CExpr
```

> Converts a string into a variable with index 1.

```
cpvar :: String → CPattern
```

> Converts a string into a pattern variable with index 1.

```
ctvar :: String → CTypeExpr
```

> Converts a string into a type variable with index 1.

A.6.5 Library AbstractCurry.Pretty

Pretty-printing of AbstractCurry.
This library provides a pretty-printer for AbstractCurry modules.

Exported types:

```
data Qualification
```


Exported constructors:

```
data LayoutChoice
```

> The choice for a generally preferred layout.

Exported constructors:

- ```
  PreferNestedLayout :: LayoutChoice
  ```
  ```
  PreferNestedLayout
  ```

    – prefer a layout where the arguments of long expressions are vertically aligned

- `PreferFilledLayout :: LayoutChoice`

  `PreferFilledLayout`

    - prefer a layout where the arguments of long expressions are filled as long as possible into one line

`data Options`

Exported constructors:

Exported functions:

`defaultOptions :: Options`

The default options to pretty print a module. These are:

- page width: 78 characters
- indentation width: 2 characters
- qualification method: qualify all imported names (except prelude names)
- layout choice: prefer nested layout (see `LayoutChoice`)

These options can be changed by corresponding setters (`setPageWith`, `setIndentWith`, `set...Qualification`, `setLayoutChoice`).

Note: If these default options are used for pretty-print operations other than `prettyCurryProg` or `ppCurryProg`, then one has to set the current module name explicitly by `setModName`!

`setPageWith :: Int → Options → Options`

Sets the page width of the pretty printer options.

`setIndentWith :: Int → Options → Options`

Sets the indentation width of the pretty printer options.

`setImportQualification :: Options → Options`

Sets the qualification method to be used to print identifiers to "import qualification" (which is the default). In this case, all identifiers imported from other modules (except for the identifiers of the prelude) are fully qualified.

`setNoQualification :: Options → Options`

Sets the qualification method to be used to print identifiers to "unqualified". In this case, no identifiers is printed with its module qualifier. This might lead to name conflicts or unintended references if some identifiers in the pretty-printed module are in conflict with imported identifiers.

setFullQualification :: Options → Options

Sets the qualification method to be used to print identifiers to "fully qualified". In this case, every identifiers, including those of the processed module and the prelude, are fully qualified.

setOnDemandQualification :: [CurryProg] → Options → Options

Sets the qualification method to be used to print identifiers to "qualification on demand". In this case, an identifier is qualified only if it is necessary to avoid a name conflict, e.g., if a local identifier has the same names as an imported identifier. Since it is necessary to know the names of all identifiers defined in the current module (to be pretty printed) and imported from other modules, the first argument is the list of modules consisting of the current module and all imported modules (including the prelude). The current module must always be the head of this list.

setModName :: String → Options → Options

Sets the name of the current module in the pretty printer options.

setLayoutChoice :: LayoutChoice → Options → Options

Sets the preferred layout in the pretty printer options.

showCProg :: CurryProg → String

Shows a pretty formatted version of an abstract Curry Program. The options for pretty-printing are the `defaultOptions` (and therefore the restrictions mentioned there apply here too).

prettyCurryProg :: Options → CurryProg → String

Pretty-print the document generated by `ppCurryProg`, using the page width specified by given options.

ppCurryProg :: Options → CurryProg → Doc

Pretty-print a CurryProg (the representation of a program, written in Curry, using AbstractCurry) according to given options. This function will overwrite the module name given by options with the name specified as the first component of `CurryProg`. The list of imported modules is extended to all modules mentioned in the program if qualified pretty printing is used. This is necessary to avoid errors w.r.t. names re-exported by modules.

ppMName :: String → Doc

Pretty-print a module name (just a string).

ppExports :: Options → [CTypeDecl] → [CFuncDecl] → Doc

Pretty-print exports, i.e. all type and function declarations which are public. extract the type and function declarations which are public and gather their qualified names in a list.

ppImports :: Options → [String] → Doc

Pretty-print imports (list of module names) by prepending the word "import" to the module name. If the qualification mode is `Imports` or `Full`, then the imports are declared as `qualified`.

ppCOpDecl :: Options → COpDecl → Doc

Pretty-print operator precedence declarations.

ppCTypeDecl :: Options → CTypeDecl → Doc

Pretty-print type declarations, like `data ...  = ...`, `type ...  = ...` or `newtype ...  = ...`.

ppCFuncDecl :: Options → CFuncDecl → Doc

Pretty-print a function declaration.

ppCFuncDeclWithoutSig :: Options → CFuncDecl → Doc

Pretty-print a function declaration without signature.

ppCFuncSignature :: Options → (String,String) → CTypeExpr → Doc

Pretty-print a function signature according to given options.

ppCTypeExpr :: Options → CTypeExpr → Doc

Pretty-print a type expression.

ppCRules :: Options → (String,String) → [CRule] → Doc

Pretty-print a list of function rules, concatenated vertically.

ppCRule :: Options → (String,String) → CRule → Doc

Pretty-print a rule of a function. Given a function `f x y = x * y`, then `x y = x * y` is a rule consisting of `x y` as list of patterns and `x * y` as right hand side.

ppCPattern :: Options → CPattern → Doc

Pretty-print a pattern expression.

ppCLiteral :: Options → CLiteral → Doc

Pretty-print given literal (Int, Float, ...).

ppCRhs :: Doc → Options → CRhs → Doc

Pretty-print the right hand side of a rule (or case expression), including the d sign, where d is the relation (as doc) between the left hand side and the right hand side – usually this is one of =, ->. If the right hand side contains local declarations, they will be pretty printed too, further indented.

ppCExpr :: Options → CExpr → Doc

Pretty-print an expression.

ppCStatement :: Options → CStatement → Doc

ppQFunc :: Options → (String,String) → Doc

Pretty-print a function name or constructor name qualified according to given options. Use ppQType or ppType for pretty-printing type names.

ppFunc :: (String,String) → Doc

Pretty-print a function name or constructor name non-qualified. Use ppQType or ppType for pretty-printing type names.

ppQType :: Options → (String,String) → Doc

Pretty-print a type (QName) qualified according to given options.

ppType :: (String,String) → Doc

Pretty-print a type (QName) non-qualified.

A.6.6   Library FlatCurry.Types

This library supports meta-programming, i.e., the manipulation of Curry programs in Curry. For this purpose, the library contains definitions of data types for the representation of so-called FlatCurry programs.

Exported types:

type QName = (String,String)

The data type for representing qualified names. In FlatCurry all names are qualified to avoid name clashes. The first component is the module name and the second component the unqualified name as it occurs in the source program.

type TVarIndex = Int

The data type for representing type variables. They are represented by (TVar i) where i is a type variable index.

type VarIndex = Int

Data type for representing object variables. Object variables occurring in expressions are represented by (`Var i`) where `i` is a variable index.

type Arity = Int

Arity of a function.

data Prog

Data type for representing a Curry module in the intermediate form. A value of this data type has the form

```
(Prog modname imports typedecls functions opdecls)
```

where `modname` is the name of this module, `imports` is the list of modules names that are imported, and `typedecls`, `functions`, and `opdecls` are the list of data type, function, and operator declarations contained in this module, respectively.

Exported constructors:

- Prog :: String → [String] → [TypeDecl] → [FuncDecl] → [OpDecl] → Prog

data Visibility

Data type to specify the visibility of various entities.

Exported constructors:

- Public :: Visibility

- Private :: Visibility

data TypeDecl

Data type for representing definitions of algebraic data types and type synonyms.

A data type definition of the form

```
data t x1...xn = ...| c t1....tkc |...
```

is represented by the FlatCurry term

```
(Type t [i1,...,in] [...(Cons c kc [t1,...,tkc])...])
```

where each `ij` is the index of the type variable `xj`.

Note: the type variable indices are unique inside each type declaration and are usually numbered from 0

Thus, a data type declaration consists of the name of the data type, a list of type parameters and a list of constructor declarations.

Exported constructors:

- Type :: (String,String) → Visibility → [Int] → [ConsDecl] → TypeDecl

- TypeSyn :: (String,String) → Visibility → [Int] → TypeExpr → TypeDecl

data ConsDecl

A constructor declaration consists of the name and arity of the constructor and a list of the argument types of the constructor.

Exported constructors:

- Cons :: (String,String) → Int → Visibility → [TypeExpr] → ConsDecl

data TypeExpr

Data type for type expressions. A type expression is either a type variable, a function type, or a type constructor application.

Note: the names of the predefined type constructors are "Int", "Float", "Bool", "Char", "IO", "()" (unit type), "(,...,)" (tuple types), "[]" (list type)

Exported constructors:

- TVar :: Int → TypeExpr

- FuncType :: TypeExpr → TypeExpr → TypeExpr

- TCons :: (String,String) → [TypeExpr] → TypeExpr

data OpDecl

Data type for operator declarations. An operator declaration fix p n in Curry corresponds to the FlatCurry term (Op n fix p).

Exported constructors:

- Op :: (String,String) → Fixity → Int → OpDecl

data Fixity

Data types for the different choices for the fixity of an operator.

Exported constructors:

- InfixOp :: Fixity

- InfixlOp :: Fixity

- InfixrOp :: Fixity

data FuncDecl

Data type for representing function declarations.

A function declaration in FlatCurry is a term of the form

```
(Func name k type (Rule [i1,...,ik] e))
```

and represents the function **name** with definition

```
name :: type
name x1...xk = e
```

where each **ij** is the index of the variable **xj**.

Note: the variable indices are unique inside each function declaration and are usually numbered from 0

External functions are represented as

```
(Func name arity type (External s))
```

where s is the external name associated to this function.

Thus, a function declaration consists of the name, arity, type, and rule.

Exported constructors:

- Func :: (String,String) → Int → Visibility → TypeExpr → Rule → FuncDecl

data Rule

A rule is either a list of formal parameters together with an expression or an "External" tag.

Exported constructors:

- Rule :: [Int] → Expr → Rule

- External :: String → Rule

data CaseType

Data type for classifying case expressions. Case expressions can be either flexible or rigid in Curry.

Exported constructors:

- Rigid :: CaseType

- Flex :: CaseType

data CombType

Data type for classifying combinations (i.e., a function/constructor applied to some arguments).

Exported constructors:

- `FuncCall :: CombType`

  `FuncCall`

  – a call to a function where all arguments are provided

- `ConsCall :: CombType`

  `ConsCall`

  – a call with a constructor at the top, all arguments are provided

- `FuncPartCall :: Int → CombType`

  `FuncPartCall`

  – a partial call to a function (i.e., not all arguments are provided) where the parameter is the number of missing arguments

- `ConsPartCall :: Int → CombType`

  `ConsPartCall`

  – a partial call to a constructor (i.e., not all arguments are provided) where the parameter is the number of missing arguments

`data Expr`

Data type for representing expressions.

Remarks:

if-then-else expressions are represented as rigid case expressions:

`(if e1 then e2 else e3)`

is represented as

`(case e1 of { True -> e2; False -> e3})`

Higher-order applications are represented as calls to the (external) function `apply`. For instance, the rule

`app f x = f x`

is represented as

```
(Rule  [0,1] (Comb FuncCall ("Prelude","apply") [Var 0, Var 1]))
```

A conditional rule is represented as a call to an external function `cond` where the first argument is the condition (a constraint). For instance, the rule

```
equal2 x | x=:=2 = True
```

is represented as

```
(Rule [0]
      (Comb FuncCall ("Prelude","cond")
            [Comb FuncCall ("Prelude","=:=") [Var 0, Lit (Intc 2)],
             Comb FuncCall ("Prelude","True") []]))
```

Exported constructors:

- `Var :: Int → Expr`

  `Var`

    – variable (represented by unique index)

- `Lit :: Literal → Expr`

  `Lit`

    – literal (Int/Float/Char constant)

- `Comb :: CombType → (String,String) → [Expr] → Expr`

  `Comb`

    – application (`f e1 ...  en`) of function/constructor `f` with `n<=`arity(`f`)

- `Let :: [(Int,Expr)] → Expr → Expr`

  `Let`

    – introduction of local variables via (recursive) let declarations

- `Free :: [Int] → Expr → Expr`

  `Free`

    – introduction of free local variables

- `Or :: Expr → Expr → Expr`

  `Or`

    – disjunction of two expressions (used to translate rules with overlapping left-hand sides)

```

- `Case :: CaseType → Expr → [BranchExpr] → Expr`

  Case

    – case distinction (rigid or flex)

- `Typed :: Expr → TypeExpr → Expr`

  Typed

    – typed expression to represent an expression with a type declaration

## data BranchExpr

Data type for representing branches in a case expression.

Branches "(m.c x1...xn) -> e" in case expressions are represented as

`(Branch (Pattern (m,c) [i1,...,in]) e)`

where each `ij` is the index of the pattern variable `xj`, or as

`(Branch (LPattern (Intc i)) e)`

for integers as branch patterns (similarly for other literals like float or character constants).

*Exported constructors:*

- `Branch :: Pattern → Expr → BranchExpr`

## data Pattern

Data type for representing patterns in case expressions.

*Exported constructors:*

- `Pattern :: (String,String) → [Int] → Pattern`

- `LPattern :: Literal → Pattern`

## data Literal

Data type for representing literals occurring in an expression or case branch. It is either an integer, a float, or a character constant.

*Exported constructors:*

- `Intc :: Int → Literal`

- `Floatc :: Float → Literal`

- `Charc :: Char → Literal`

**Exported functions:**

`showQNameInModule :: String → (String,String) → String`

> Translates a given qualified type name into external name relative to a module. Thus, names not defined in this module (except for names defined in the prelude) are prefixed with their module name.

### A.6.7 Library FlatCurry.Files

This library supports meta-programming, i.e., the manipulation of Curry programs in Curry. This library defines I/O actions

**Exported functions:**

`readFlatCurry :: String → IO Prog`

> I/O action which parses a Curry program and returns the corresponding FlatCurry program. Thus, the argument is the module path (without suffix ".curry" or ".lcurry") and the result is a FlatCurry term representing this program.

`readFlatCurryWithParseOptions :: String → FrontendParams → IO Prog`

> I/O action which parses a Curry program with respect to some parser options and returns the corresponding FlatCurry program. This I/O action is used by the standard action `readFlatCurry`.

`flatCurryFileName :: String → String`

> Transforms a name of a Curry program (with or without suffix ".curry" or ".lcurry") into the name of the file containing the corresponding FlatCurry program.

`flatCurryIntName :: String → String`

> Transforms a name of a Curry program (with or without suffix ".curry" or ".lcurry") into the name of the file containing the corresponding FlatCurry program.

`readFlatCurryFile :: String → IO Prog`

> I/O action which reads a FlatCurry program from a file in ".fcy" format. In contrast to `readFlatCurry`, this action does not parse a source program. Thus, the argument must be the name of an existing file (with suffix ".fcy") containing a FlatCurry program in ".fcy" format and the result is a FlatCurry term representing this program.

`readFlatCurryInt :: String → IO Prog`

> I/O action which returns the interface of a Curry module, i.e., a FlatCurry program containing only "Public" entities and function definitions without rules (i.e., external functions). The argument is the file name without suffix ".curry" (or ".lcurry") and the result is a FlatCurry term representing the interface of this module.

`readFlatCurryIntWithParseOptions :: String → FrontendParams → IO Prog`

> I/O action which parses Curry program with respect to some parser options and returns the FlatCurry interface of this program, i.e., a FlatCurry program containing only "Public" entities and function definitions without rules (i.e., external functions). The argument is the file name without suffix ".curry" (or ".lcurry") and the result is a FlatCurry term representing the interface of this module.

`writeFCY :: String → Prog → IO ()`

> Writes a FlatCurry program into a file in ".fcy" format. The first argument must be the name of the target file (with suffix ".fcy").

`lookupFlatCurryFileInLoadPath :: String → IO (Maybe String)`

> Returns the name of the FlatCurry file of a module in the load path, if this file exists.

`getFlatCurryFileInLoadPath :: String → IO String`

> Returns the name of the FlatCurry file of a module in the load path, if this file exists.

### A.6.8   Library FlatCurry.Goodies

This library provides selector functions, test and update operations as well as some useful auxiliary functions for FlatCurry data terms. Most of the provided functions are based on general transformation functions that replace constructors with user-defined functions. For recursive datatypes the transformations are defined inductively over the term structure. This is quite usual for transformations on FlatCurry terms, so the provided functions can be used to implement specific transformations without having to explicitly state the recursion. Essentially, the tedious part of such transformations - descend in fairly complex term structures - is abstracted away, which hopefully makes the code more clear and brief.

**Exported types:**

`type Update a b = (b → b) → a → a`

**Exported functions:**

`trProg :: (String → [String] → [TypeDecl] → [FuncDecl] → [OpDecl] → a) →`
`Prog → a`

> transform program

`progName :: Prog → String`

> get name from program

`progImports :: Prog → [String]`

get imports from program

`progTypes :: Prog → [TypeDecl]`

get type declarations from program

`progFuncs :: Prog → [FuncDecl]`

get functions from program

`progOps :: Prog → [OpDecl]`

get infix operators from program

`updProg :: (String → String) → ([String] → [String]) → ([TypeDecl] → [TypeDecl]) → ([FuncDecl] → [FuncDecl]) → ([OpDecl] → [OpDecl]) → Prog → Prog`

update program

`updProgName :: (String → String) → Prog → Prog`

update name of program

`updProgImports :: ([String] → [String]) → Prog → Prog`

update imports of program

`updProgTypes :: ([TypeDecl] → [TypeDecl]) → Prog → Prog`

update type declarations of program

`updProgFuncs :: ([FuncDecl] → [FuncDecl]) → Prog → Prog`

update functions of program

`updProgOps :: ([OpDecl] → [OpDecl]) → Prog → Prog`

update infix operators of program

`allVarsInProg :: Prog → [Int]`

get all program variables (also from patterns)

`updProgExps :: (Expr → Expr) → Prog → Prog`

lift transformation on expressions to program

`rnmAllVarsInProg :: (Int → Int) → Prog → Prog`

rename programs variables

`updQNamesInProg :: ((String,String) → (String,String)) → Prog → Prog`

update all qualified names in program

rnmProg :: String → Prog → Prog

     rename program (update name of and all qualified names in program)

trType :: ((String,String) → Visibility → [Int] → [ConsDecl] → a) →
((String,String) → Visibility → [Int] → TypeExpr → a) → TypeDecl → a

     transform type declaration

typeName :: TypeDecl → (String,String)

     get name of type declaration

typeVisibility :: TypeDecl → Visibility

     get visibility of type declaration

typeParams :: TypeDecl → [Int]

     get type parameters of type declaration

typeConsDecls :: TypeDecl → [ConsDecl]

     get constructor declarations from type declaration

typeSyn :: TypeDecl → TypeExpr

     get synonym of type declaration

isTypeSyn :: TypeDecl → Bool

     is type declaration a type synonym?

updType :: ((String,String) → (String,String)) → (Visibility → Visibility)
→ ([Int] → [Int]) → ([ConsDecl] → [ConsDecl]) → (TypeExpr → TypeExpr) →
TypeDecl → TypeDecl

     update type declaration

updTypeName :: ((String,String) → (String,String)) → TypeDecl → TypeDecl

     update name of type declaration

updTypeVisibility :: (Visibility → Visibility) → TypeDecl → TypeDecl

     update visibility of type declaration

updTypeParams :: ([Int] → [Int]) → TypeDecl → TypeDecl

     update type parameters of type declaration

updTypeConsDecls :: ([ConsDecl] → [ConsDecl]) → TypeDecl → TypeDecl

     update constructor declarations of type declaration

317

```
updTypeSynonym :: (TypeExpr → TypeExpr) → TypeDecl → TypeDecl
```

  update synonym of type declaration

```
updQNamesInType :: ((String,String) → (String,String)) → TypeDecl → TypeDecl
```

  update all qualified names in type declaration

```
trCons :: ((String,String) → Int → Visibility → [TypeExpr] → a) → ConsDecl →
a
```

  transform constructor declaration

```
consName :: ConsDecl → (String,String)
```

  get name of constructor declaration

```
consArity :: ConsDecl → Int
```

  get arity of constructor declaration

```
consVisibility :: ConsDecl → Visibility
```

  get visibility of constructor declaration

```
consArgs :: ConsDecl → [TypeExpr]
```

  get arguments of constructor declaration

```
updCons :: ((String,String) → (String,String)) → (Int → Int) → (Visibility →
Visibility) → ([TypeExpr] → [TypeExpr]) → ConsDecl → ConsDecl
```

  update constructor declaration

```
updConsName :: ((String,String) → (String,String)) → ConsDecl → ConsDecl
```

  update name of constructor declaration

```
updConsArity :: (Int → Int) → ConsDecl → ConsDecl
```

  update arity of constructor declaration

```
updConsVisibility :: (Visibility → Visibility) → ConsDecl → ConsDecl
```

  update visibility of constructor declaration

```
updConsArgs :: ([TypeExpr] → [TypeExpr]) → ConsDecl → ConsDecl
```

  update arguments of constructor declaration

```
updQNamesInConsDecl :: ((String,String) → (String,String)) → ConsDecl →
ConsDecl
```

  update all qualified names in constructor declaration

`tVarIndex :: TypeExpr → Int`

> get index from type variable

`domain :: TypeExpr → TypeExpr`

> get domain from functional type

`range :: TypeExpr → TypeExpr`

> get range from functional type

`tConsName :: TypeExpr → (String,String)`

> get name from constructed type

`tConsArgs :: TypeExpr → [TypeExpr]`

> get arguments from constructed type

`trTypeExpr :: (Int → a) → ((String,String) → [a] → a) → (a → a → a) →`
`TypeExpr → a`

> transform type expression

`isTVar :: TypeExpr → Bool`

> is type expression a type variable?

`isTCons :: TypeExpr → Bool`

> is type declaration a constructed type?

`isFuncType :: TypeExpr → Bool`

> is type declaration a functional type?

`updTVars :: (Int → TypeExpr) → TypeExpr → TypeExpr`

> update all type variables

`updTCons :: ((String,String) → [TypeExpr] → TypeExpr) → TypeExpr → TypeExpr`

> update all type constructors

`updFuncTypes :: (TypeExpr → TypeExpr → TypeExpr) → TypeExpr → TypeExpr`

> update all functional types

`argTypes :: TypeExpr → [TypeExpr]`

> get argument types from functional type

`resultType :: TypeExpr → TypeExpr`

> get result type from (nested) functional type

`rnmAllVarsInTypeExpr :: (Int → Int) → TypeExpr → TypeExpr`

   rename variables in type expression

`updQNamesInTypeExpr :: ((String,String) → (String,String)) → TypeExpr →`
`TypeExpr`

   update all qualified names in type expression

`trOp :: ((String,String) → Fixity → Int → a) → OpDecl → a`

   transform operator declaration

`opName :: OpDecl → (String,String)`

   get name from operator declaration

`opFixity :: OpDecl → Fixity`

   get fixity of operator declaration

`opPrecedence :: OpDecl → Int`

   get precedence of operator declaration

`updOp :: ((String,String) → (String,String)) → (Fixity → Fixity) → (Int →`
`Int) → OpDecl → OpDecl`

   update operator declaration

`updOpName :: ((String,String) → (String,String)) → OpDecl → OpDecl`

   update name of operator declaration

`updOpFixity :: (Fixity → Fixity) → OpDecl → OpDecl`

   update fixity of operator declaration

`updOpPrecedence :: (Int → Int) → OpDecl → OpDecl`

   update precedence of operator declaration

`trFunc :: ((String,String) → Int → Visibility → TypeExpr → Rule → a) →`
`FuncDecl → a`

   transform function

`funcName :: FuncDecl → (String,String)`

   get name of function

`funcArity :: FuncDecl → Int`

   get arity of function

```
funcVisibility :: FuncDecl → Visibility
```

get visibility of function

```
funcType :: FuncDecl → TypeExpr
```

get type of function

```
funcRule :: FuncDecl → Rule
```

get rule of function

```
updFunc :: ((String,String) → (String,String)) → (Int → Int) → (Visibility →
Visibility) → (TypeExpr → TypeExpr) → (Rule → Rule) → FuncDecl → FuncDecl
```

update function

```
updFuncName :: ((String,String) → (String,String)) → FuncDecl → FuncDecl
```

update name of function

```
updFuncArity :: (Int → Int) → FuncDecl → FuncDecl
```

update arity of function

```
updFuncVisibility :: (Visibility → Visibility) → FuncDecl → FuncDecl
```

update visibility of function

```
updFuncType :: (TypeExpr → TypeExpr) → FuncDecl → FuncDecl
```

update type of function

```
updFuncRule :: (Rule → Rule) → FuncDecl → FuncDecl
```

update rule of function

```
isExternal :: FuncDecl → Bool
```

is function externally defined?

```
allVarsInFunc :: FuncDecl → [Int]
```

get variable names in a function declaration

```
funcArgs :: FuncDecl → [Int]
```

get arguments of function, if not externally defined

```
funcBody :: FuncDecl → Expr
```

get body of function, if not externally defined

```
funcRHS :: FuncDecl → [Expr]
```

`rnmAllVarsInFunc :: (Int → Int) → FuncDecl → FuncDecl`

    rename all variables in function

`updQNamesInFunc :: ((String,String) → (String,String)) → FuncDecl → FuncDecl`

    update all qualified names in function

`updFuncArgs :: ([Int] → [Int]) → FuncDecl → FuncDecl`

    update arguments of function, if not externally defined

`updFuncBody :: (Expr → Expr) → FuncDecl → FuncDecl`

    update body of function, if not externally defined

`trRule :: ([Int] → Expr → a) → (String → a) → Rule → a`

    transform rule

`ruleArgs :: Rule → [Int]`

    get rules arguments if it's not external

`ruleBody :: Rule → Expr`

    get rules body if it's not external

`ruleExtDecl :: Rule → String`

    get rules external declaration

`isRuleExternal :: Rule → Bool`

    is rule external?

`updRule :: ([Int] → [Int]) → (Expr → Expr) → (String → String) → Rule → Rule`

    update rule

`updRuleArgs :: ([Int] → [Int]) → Rule → Rule`

    update rules arguments

`updRuleBody :: (Expr → Expr) → Rule → Rule`

    update rules body

`updRuleExtDecl :: (String → String) → Rule → Rule`

    update rules external declaration

`allVarsInRule :: Rule → [Int]`

    get variable names in a functions rule

```
rnmAllVarsInRule :: (Int → Int) → Rule → Rule
```

      rename all variables in rule

```
updQNamesInRule :: ((String,String) → (String,String)) → Rule → Rule
```

      update all qualified names in rule

```
trCombType :: a → (Int → a) → a → (Int → a) → CombType → a
```

      transform combination type

```
isCombTypeFuncCall :: CombType → Bool
```

      is type of combination FuncCall?

```
isCombTypeFuncPartCall :: CombType → Bool
```

      is type of combination FuncPartCall?

```
isCombTypeConsCall :: CombType → Bool
```

      is type of combination ConsCall?

```
isCombTypeConsPartCall :: CombType → Bool
```

      is type of combination ConsPartCall?

```
missingArgs :: CombType → Int
```


```
varNr :: Expr → Int
```

      get internal number of variable

```
literal :: Expr → Literal
```

      get literal if expression is literal expression

```
combType :: Expr → CombType
```

      get combination type of a combined expression

```
combName :: Expr → (String,String)
```

      get name of a combined expression

```
combArgs :: Expr → [Expr]
```

      get arguments of a combined expression

```
missingCombArgs :: Expr → Int
```

      get number of missing arguments if expression is combined

```
letBinds :: Expr → [(Int,Expr)]
```

get indices of variables in let declaration

```
letBody :: Expr → Expr
```

get body of let declaration

```
freeVars :: Expr → [Int]
```

get variable indices from declaration of free variables

```
freeExpr :: Expr → Expr
```

get expression from declaration of free variables

```
orExps :: Expr → [Expr]
```

get expressions from or-expression

```
caseType :: Expr → CaseType
```

get case-type of case expression

```
caseExpr :: Expr → Expr
```

get scrutinee of case expression

```
caseBranches :: Expr → [BranchExpr]
```

get branch expressions from case expression

```
isVar :: Expr → Bool
```

is expression a variable?

```
isLit :: Expr → Bool
```

is expression a literal expression?

```
isComb :: Expr → Bool
```

is expression combined?

```
isLet :: Expr → Bool
```

is expression a let expression?

```
isFree :: Expr → Bool
```

is expression a declaration of free variables?

```
isOr :: Expr → Bool
```

is expression an or-expression?

`isCase :: Expr → Bool`

>   is expression a case expression?

`trExpr :: (Int → a) → (Literal → a) → (CombType → (String,String) → [a] → a) → ([(Int,a)] → a → a) → ([Int] → a → a) → (a → a → a) → (CaseType → a → [b] → a) → (Pattern → a → b) → (a → TypeExpr → a) → Expr → a`

>   transform expression

`updVars :: (Int → Expr) → Expr → Expr`

>   update all variables in given expression

`updLiterals :: (Literal → Expr) → Expr → Expr`

>   update all literals in given expression

`updCombs :: (CombType → (String,String) → [Expr] → Expr) → Expr → Expr`

>   update all combined expressions in given expression

`updLets :: ([(Int,Expr)] → Expr → Expr) → Expr → Expr`

>   update all let expressions in given expression

`updFrees :: ([Int] → Expr → Expr) → Expr → Expr`

>   update all free declarations in given expression

`updOrs :: (Expr → Expr → Expr) → Expr → Expr`

>   update all or expressions in given expression

`updCases :: (CaseType → Expr → [BranchExpr] → Expr) → Expr → Expr`

>   update all case expressions in given expression

`updBranches :: (Pattern → Expr → BranchExpr) → Expr → Expr`

>   update all case branches in given expression

`updTypeds :: (Expr → TypeExpr → Expr) → Expr → Expr`

>   update all typed expressions in given expression

`isFuncCall :: Expr → Bool`

>   is expression a call of a function where all arguments are provided?

`isFuncPartCall :: Expr → Bool`

>   is expression a partial function call?

`isConsCall :: Expr → Bool`

is expression a call of a constructor?

`isConsPartCall :: Expr → Bool`

is expression a partial constructor call?

`isGround :: Expr → Bool`

is expression fully evaluated?

`allVars :: Expr → [Int]`

get all variables (also pattern variables) in expression

`rnmAllVars :: (Int → Int) → Expr → Expr`

rename all variables (also in patterns) in expression

`updQNames :: ((String,String) → (String,String)) → Expr → Expr`

update all qualified names in expression

`trBranch :: (Pattern → Expr → a) → BranchExpr → a`

transform branch expression

`branchPattern :: BranchExpr → Pattern`

get pattern from branch expression

`branchExpr :: BranchExpr → Expr`

get expression from branch expression

`updBranch :: (Pattern → Pattern) → (Expr → Expr) → BranchExpr → BranchExpr`

update branch expression

`updBranchPattern :: (Pattern → Pattern) → BranchExpr → BranchExpr`

update pattern of branch expression

`updBranchExpr :: (Expr → Expr) → BranchExpr → BranchExpr`

update expression of branch expression

`trPattern :: ((String,String) → [Int] → a) → (Literal → a) → Pattern → a`

transform pattern

`patCons :: Pattern → (String,String)`

get name from constructor pattern

`patArgs :: Pattern → [Int]`

get arguments from constructor pattern

`patLiteral :: Pattern → Literal`

get literal from literal pattern

`isConsPattern :: Pattern → Bool`

is pattern a constructor pattern?

`updPattern :: ((String,String) → (String,String)) → ([Int] → [Int]) → (Literal → Literal) → Pattern → Pattern`

update pattern

`updPatCons :: ((String,String) → (String,String)) → Pattern → Pattern`

update constructors name of pattern

`updPatArgs :: ([Int] → [Int]) → Pattern → Pattern`

update arguments of constructor pattern

`updPatLiteral :: (Literal → Literal) → Pattern → Pattern`

update literal of pattern

`patExpr :: Pattern → Expr`

build expression from pattern

A.6.9 Library FlatCurry.Pretty

This library provides pretty-printers for FlatCurry modules and all substructures (e.g., expressions).

Exported types:

`data Options`

Options for pretty printing

Exported constructors:

- `Options :: Int → QualMode → String → Options`

`data QualMode`

Qualification mode, determines whether identifiers are printed qualified or unqualified. While `QualNone` and `QualImports` aim at readability, there may be ambiguities due to shadowing. On the contrary, `QualImports` and `QualAll` produce correct output at the cost of readability.

Exported constructors:

- QualNone :: QualMode

  QualNone

  – no qualification, only unqualified names

- QualImportsButPrelude :: QualMode

  QualImportsButPrelude

  – qualify all imports except those from the module Prelude

- QualImports :: QualMode

  QualImports

  – qualify all imports, including Prelude

- QualAll :: QualMode

  QualAll

  – qualify all names

Exported functions:

indentWidth :: Options → Int

qualMode :: Options → QualMode

currentModule :: Options → String

defaultOptions :: Options

   Default Options for pretty-printing.

ppProg :: Options → Prog → Doc

   pretty-print a FlatCurry module

ppHeader :: Options → String → [TypeDecl] → [FuncDecl] → Doc

   pretty-print the module header

ppExports :: Options → [TypeDecl] → [FuncDecl] → Doc

   pretty-print the export list

ppTypeExport :: Options → TypeDecl → Doc

   pretty-print a type export

```
ppConsExports :: Options → [ConsDecl] → [Doc]
```

pretty-print the export list of constructors

```
ppFuncExports :: Options → [FuncDecl] → [Doc]
```

pretty-print the export list of functions

```
ppImports :: Options → [String] → Doc
```

pretty-print a list of import statements

```
ppImport :: Options → String → Doc
```

pretty-print a single import statement

```
ppOpDecls :: Options → [OpDecl] → Doc
```

pretty-print a list of operator fixity declarations

```
ppOpDecl :: Options → OpDecl → Doc
```

pretty-print a single operator fixity declaration

```
ppFixity :: Fixity → Doc
```

pretty-print the associativity keyword

```
ppTypeDecls :: Options → [TypeDecl] → Doc
```

pretty-print a list of type declarations

```
ppTypeDecl :: Options → TypeDecl → Doc
```

pretty-print a type declaration

```
ppConsDecls :: Options → [ConsDecl] → Doc
```

pretty-print the constructor declarations

```
ppConsDecl :: Options → ConsDecl → Doc
```

pretty print a single constructor

```
ppTypeExp :: Options → TypeExpr → Doc
```

pretty a top-level type expression

```
ppTypeExpr :: Options → Int → TypeExpr → Doc
```

pretty-print a type expression

```
ppTVarIndex :: Int → Doc
```

pretty-print a type variable

```
ppFuncDecls :: Options → [FuncDecl] → Doc
```

   pretty-print a list of function declarations

```
ppFuncDecl :: Options → FuncDecl → Doc
```

   pretty-print a function declaration

```
ppRule :: Options → Rule → Doc
```

   pretty-print a function rule

```
ppExp :: Options → Expr → Doc
```

   Pretty-print a top-level expression.

```
ppExpr :: Options → Int → Expr → Doc
```

   pretty-print an expression

```
ppVarIndex :: Int → Doc
```

   pretty-print a variable

```
ppLiteral :: Literal → Doc
```

   pretty-print a literal

```
ppComb :: Options → Int → (String,String) → [Expr] → Doc
```

   Pretty print a constructor or function call

```
ppDecls :: Options → [(Int,Expr)] → Doc
```

   pretty-print a list of declarations

```
ppDecl :: Options → (Int,Expr) → Doc
```

   pretty-print a single declaration

```
ppCaseType :: CaseType → Doc
```

   Pretty print the type of a case expression

```
ppBranch :: Options → BranchExpr → Doc
```

   Pretty print a case branch

```
ppPattern :: Options → Pattern → Doc
```

   Pretty print a pattern

```
ppPrefixQOp :: Options → (String,String) → Doc
```

   pretty-print a qualified prefix operator.

```
ppPrefixOp :: (String,String) → Doc
```

>   pretty-print a prefix operator unqualified.

```
ppInfixQOp :: Options → (String,String) → Doc
```

>   pretty-print an infix operator

```
ppQName :: Options → (String,String) → Doc
```

>   Pretty-print a qualified name

```
ppName :: (String,String) → Doc
```

>   Pretty-print a qualified name unqualified (e.g., for type definitions).

```
isInfixOp :: (String,String) → Bool
```

>   Check whether an operator is an infix operator

```
isConsId :: (String,String) → Bool
```

>   Check whether an identifier represents the : list constructor.

```
isListId :: (String,String) → Bool
```

>   Check whether an identifier represents a list

```
isTupleId :: (String,String) → Bool
```

>   Check whether an identifier represents a tuple

```
indent :: Options → Doc → Doc
```

>   Indentation

A.6.10   Library FlatCurry.Read

This library defines operations to read a FlatCurry programs or interfaces together with all its imported modules in the current load path.

Exported functions:

```
readFlatCurryInPath :: [String] → String → IO Prog
```

>   Reads a FlatCurry program together in a given load path. The arguments are a load path and the name of the module.

```
readFlatCurryWithImports :: String → IO [Prog]
```

>   Reads a FlatCurry program together with all its imported modules. The argument is the name of the main module, possibly with a directory prefix.

```
readFlatCurryWithImportsInPath :: [String] → String → IO [Prog]
```

331

Reads a FlatCurry program together with all its imported modules in a given load path. The arguments are a load path and the name of the main module.

`readFlatCurryIntWithImports :: String → IO [Prog]`

Reads a FlatCurry interface together with all its imported module interfaces. The argument is the name of the main module, possibly with a directory prefix. If there is no interface file but a FlatCurry file (suffix ".fcy"), the FlatCurry file is read instead of the interface.

`readFlatCurryIntWithImportsInPath :: [String] → String → IO [Prog]`

Reads a FlatCurry interface together with all its imported module interfaces in a given load path. The arguments are a load path and the name of the main module. If there is no interface file but a FlatCurry file (suffix ".fcy"), the FlatCurry file is read instead of the interface.

A.6.11 Library FlatCurry.Show

This library contains operations to transform FlatCurry programs into string representations, either in a FlatCurry format or in a Curry-like syntax.
This library contains

- show functions for a string representation of FlatCurry programs (`showFlatProg`, `showFlatType`, `showFlatFunc`)

- functions for showing FlatCurry (type) expressions in (almost) Curry syntax (`showCurryType`, `showCurryExpr`,...).

Exported functions:

`showFlatProg :: Prog → String`

Shows a FlatCurry program term as a string (with some pretty printing).

`showFlatType :: TypeDecl → String`

`showFlatFunc :: FuncDecl → String`

`showCurryType :: ((String,String) → String) → Bool → TypeExpr → String`

Shows a FlatCurry type in Curry syntax.

`showCurryExpr :: ((String,String) → String) → Bool → Int → Expr → String`

Shows a FlatCurry expressions in (almost) Curry syntax.

`showCurryVar :: a → String`

`showCurryId :: String → String`

Shows an identifier in Curry form. Thus, operators are enclosed in brackets.

A.6.12 Library FlatCurry.XML

This library contains functions to convert FlatCurry programs into corresponding XML expressions and vice versa. This can be used to store Curry programs in a way independent of a Curry system or to use a Curry system, like PAKCS, as back end by other functional logic programming systems.

Exported functions:

flatCurry2XmlFile :: Prog → String → IO ()

> Transforms a FlatCurry program term into a corresponding XML file.

flatCurry2Xml :: Prog → XmlExp

> Transforms a FlatCurry program term into a corresponding XML expression.

xmlFile2FlatCurry :: String → IO Prog

> Reads an XML file with a FlatCurry program and returns the FlatCurry program.

xml2FlatCurry :: XmlExp → Prog

> Transforms an XML term into a FlatCurry program.

A.6.13 Library FlatCurry.FlexRigid

This library provides a function to compute the rigid/flex status of a FlatCurry expression (right-hand side of a function definition).

Exported types:

data FlexRigidResult

> Datatype for representing a flex/rigid status of an expression.

> *Exported constructors:*

> - UnknownFR :: FlexRigidResult

> - ConflictFR :: FlexRigidResult

> - KnownFlex :: FlexRigidResult

> - KnownRigid :: FlexRigidResult

Exported functions:

getFlexRigid :: Expr → FlexRigidResult

> Computes the rigid/flex status of a FlatCurry expression. This function checks all cases in this expression. If the expression has rigid as well as flex cases (which cannot be the case for source level programs but might occur after some program transformations), the result ConflictFR is returned.

A.6.14 Library FlatCurry.Compact

This module contains functions to reduce the size of FlatCurry programs by combining the main module and all imports into a single program that contains only the functions directly or indirectly called from a set of main functions.

Exported types:

`data Option`

> Options to guide the compactification process.

Exported constructors:

- `Verbose :: Option`

  `Verbose`

  - for more output

- `Main :: String → Option`

  `Main`

  - optimize for one main (unqualified!) function supplied here

- `Exports :: Option`

  `Exports`

  - optimize w.r.t. the exported functions of the module only

- `InitFuncs :: [(String,String)] → Option`

  `InitFuncs`

  - optimize w.r.t. given list of initially required functions

- `Required :: [RequiredSpec] → Option`

  `Required`

  - list of functions that are implicitly required and, thus, should not be deleted if the corresponding module is imported

- `Import :: String → Option`

  `Import`

  - module that should always be imported (useful in combination with option InitFuncs)

`data RequiredSpec`

> Data type to specify requirements of functions.

Exported constructors:

Exported functions:

requires :: (String,String) → (String,String) → RequiredSpec

>   (fun requires reqfun) specifies that the use of the function "fun" implies the application
>   of function "reqfun".

alwaysRequired :: (String,String) → RequiredSpec

>   (alwaysRequired fun) specifies that the function "fun" should be always present if the
>   corresponding module is loaded.

defaultRequired :: [RequiredSpec]

>   Functions that are implicitly required in a FlatCurry program (since they might be
>   generated by external functions like "==" or "=:=" on the fly).

generateCompactFlatCurryFile :: [Option] → String → String → IO ()

>   Computes a single FlatCurry program containing all functions potentially called from
>   a set of main functions and writes it into a FlatCurry file. This is done by merging all
>   imported FlatCurry modules and removing the imported functions that are definitely
>   not used.

computeCompactFlatCurry :: [Option] → String → IO Prog

>   Computes a single FlatCurry program containing all functions potentially called from a
>   set of main functions. This is done by merging all imported FlatCurry modules (these
>   are loaded demand-driven so that modules that contains no potentially called functions
>   are not loaded) and removing the imported functions that are definitely not used.

A.6.15   Library FlatCurry.Annotated.Types

This library contains a version of FlatCurry's abstract syntax tree which can be annotated with
arbitrary information due to a polymorphic type parameter. For instance, this could be used to
annotate function declarations and expressions with their corresponding type.
For more information about the abstract syntax tree of FlatCurry, see the documentation of the
respective module.

Exported types:

type Arity = Int

>   Arity of a function declaration

data AProg

>   Annotated FlatCurry program (corresponds to a module)

  Exported constructors:

- APROG :: String → [String] → [TypeDecl] → [AFuncDecl a] → [OpDecl] →
  AProg a

data AFuncDecl

   Annotated function declaration

Exported constructors:

- AFunc :: (String,String) → Int → Visibility → TypeExpr → (ARule a) →
  AFuncDecl a

data ARule

   Annotated function rule

Exported constructors:

- ARule :: a → [(Int,a)] → (AExpr a) → ARule a

- AExternal :: a → String → ARule a

data AExpr

   Annotated expression

Exported constructors:

- AVar :: a → Int → AExpr a

- ALit :: a → Literal → AExpr a

- AComb :: a → CombType → ((String,String),a) → [AExpr a] → AExpr a

- ALet :: a → [((Int,a),AExpr a)] → (AExpr a) → AExpr a

- AFree :: a → [(Int,a)] → (AExpr a) → AExpr a

- AOr :: a → (AExpr a) → (AExpr a) → AExpr a

- ACase :: a → CaseType → (AExpr a) → [ABranchExpr a] → AExpr a

- ATyped :: a → (AExpr a) → TypeExpr → AExpr a

data ABranchExpr

   Annotated case branch

Exported constructors:

- ABranch :: (APattern a) → (AExpr a) → ABranchExpr a

data APattern

   Annotated pattern

Exported constructors:

- APattern :: a → ((String,String),a) → [(Int,a)] → APattern a

- ALPattern :: a → Literal → APattern a

336

A.6.16   Library FlatCurry.Annotated.Pretty

This library provides pretty-printers for AnnotatedFlatCurry modules and all substructures (e.g., expressions). Note that annotations are ignored for pretty-printing.

Exported functions:

ppProg :: AProg a → Doc

>   pretty-print a FlatCurry module

ppHeader :: String → [TypeDecl] → [AFuncDecl a] → Doc

>   pretty-print the module header

ppExports :: [TypeDecl] → [AFuncDecl a] → Doc

>   pretty-print the export list

ppTypeExport :: TypeDecl → Doc

>   pretty-print a type export

ppConsExports :: [ConsDecl] → [Doc]

>   pretty-print the export list of constructors

ppFuncExports :: [AFuncDecl a] → [Doc]

>   pretty-print the export list of functions

ppImports :: [String] → Doc

>   pretty-print a list of import statements

ppImport :: String → Doc

>   pretty-print a single import statement

ppOpDecls :: [OpDecl] → Doc

>   pretty-print a list of operator fixity declarations

ppOpDecl :: OpDecl → Doc

>   pretty-print a single operator fixity declaration

ppFixity :: Fixity → Doc

>   pretty-print the associativity keyword

ppTypeDecls :: [TypeDecl] → Doc

>   pretty-print a list of type declarations

```
ppTypeDecl :: TypeDecl → Doc
```

  pretty-print a type declaration

```
ppConsDecls :: [ConsDecl] → Doc
```

  pretty-print the constructor declarations

```
ppConsDecl :: ConsDecl → Doc
```

  pretty print a single constructor

```
ppTypeExp :: TypeExpr → Doc
```

  pretty a top-level type expression

```
ppTypeExpr :: Int → TypeExpr → Doc
```

  pretty-print a type expression

```
ppTVarIndex :: Int → Doc
```

  pretty-print a type variable

```
ppFuncDecls :: [AFuncDecl a] → Doc
```

  pretty-print a list of function declarations

```
ppFuncDecl :: AFuncDecl a → Doc
```

  pretty-print a function declaration

```
ppRule :: ARule a → Doc
```

  pretty-print a function rule

```
ppExp :: AExpr a → Doc
```

  pretty-print a top-level expression

```
ppExpr :: Int → AExpr a → Doc
```

  pretty-print an expression

```
ppAVarIndex :: (Int,a) → Doc
```

  pretty-print an annotated variable

```
ppVarIndex :: Int → Doc
```

  pretty-print a variable

```
ppLiteral :: Literal → Doc
```

  pretty-print a literal

```
showEscape :: Char → String
```

Escape character literal

```
ppComb :: Int → ((String,String),a) → [AExpr b] → Doc
```

Pretty print a constructor or function call

```
ppDecls :: [((Int,a),AExpr b)] → Doc
```

pretty-print a list of declarations

```
ppDecl :: ((Int,a),AExpr b) → Doc
```

pretty-print a single declaration

```
ppCaseType :: CaseType → Doc
```

Pretty print the type of a case expression

```
ppBranch :: ABranchExpr a → Doc
```

Pretty print a case branch

```
ppPattern :: APattern a → Doc
```

Pretty print a pattern

```
ppPrefixOp :: (String,String) → Doc
```

pretty-print a prefix operator

```
ppInfixOp :: (String,String) → Doc
```

pretty-print an infix operator

```
ppQName :: (String,String) → Doc
```

Pretty-print a qualified name

```
isInfixOp :: (String,String) → Bool
```

Check whether an operator is an infix operator

```
isListId :: (String,String) → Bool
```

Check whether an identifier represents a list

```
isTupleId :: (String,String) → Bool
```

Check whether an identifier represents a tuple

```
indent :: Doc → Doc
```

Indentation

A.6.17 Library FlatCurry.Annotated.Goodies

This library provides selector functions, test and update operations as well as some useful auxiliary functions for FlatCurry data terms. Most of the provided functions are based on general transformation functions that replace constructors with user-defined functions. For recursive datatypes the transformations are defined inductively over the term structure. This is quite usual for transformations on FlatCurry terms, so the provided functions can be used to implement specific transformations without having to explicitly state the recursion. Essentially, the tedious part of such transformations - descend in fairly complex term structures - is abstracted away, which hopefully makes the code more clear and brief.

Exported types:

```
type Update a b = (b → b) → a → a
```

Exported functions:

```
trProg :: (String → [String] → [TypeDecl] → [AFuncDecl a] → [OpDecl] → b) →
AProg a → b
```

   transform program

```
progName :: AProg a → String
```

   get name from program

```
progImports :: AProg a → [String]
```

   get imports from program

```
progTypes :: AProg a → [TypeDecl]
```

   get type declarations from program

```
progFuncs :: AProg a → [AFuncDecl a]
```

   get functions from program

```
progOps :: AProg a → [OpDecl]
```

   get infix operators from program

```
updProg :: (String → String) → ([String] → [String]) → ([TypeDecl] →
[TypeDecl]) → ([AFuncDecl a] → [AFuncDecl a]) → ([OpDecl] → [OpDecl]) → AProg
a → AProg a
```

   update program

```
updProgName :: (String → String) → AProg a → AProg a
```

update name of program

```
updProgImports :: ([String] → [String]) → AProg a → AProg a
```

      update imports of program

```
updProgTypes :: ([TypeDecl] → [TypeDecl]) → AProg a → AProg a
```

      update type declarations of program

```
updProgFuncs :: ([AFuncDecl a] → [AFuncDecl a]) → AProg a → AProg a
```

      update functions of program

```
updProgOps :: ([OpDecl] → [OpDecl]) → AProg a → AProg a
```

      update infix operators of program

```
allVarsInProg :: AProg a → [Int]
```

      get all program variables (also from patterns)

```
updProgExps :: (AExpr a → AExpr a) → AProg a → AProg a
```

      lift transformation on expressions to program

```
rnmAllVarsInProg :: (Int → Int) → AProg a → AProg a
```

      rename programs variables

```
updQNamesInProg :: ((String,String) → (String,String)) → AProg a → AProg a
```

      update all qualified names in program

```
rnmProg :: String → AProg a → AProg a
```

      rename program (update name of and all qualified names in program)

```
trType :: ((String,String) → Visibility → [Int] → [ConsDecl] → a) →
((String,String) → Visibility → [Int] → TypeExpr → a) → TypeDecl → a
```

      transform type declaration

```
typeName :: TypeDecl → (String,String)
```

      get name of type declaration

```
typeVisibility :: TypeDecl → Visibility
```

      get visibility of type declaration

```
typeParams :: TypeDecl → [Int]
```

      get type parameters of type declaration

```
typeConsDecls :: TypeDecl → [ConsDecl]
```

get constructor declarations from type declaration

`typeSyn :: TypeDecl → TypeExpr`

get synonym of type declaration

`isTypeSyn :: TypeDecl → Bool`

is type declaration a type synonym?

`updType :: ((String,String) → (String,String)) → (Visibility → Visibility) → ([Int] → [Int]) → ([ConsDecl] → [ConsDecl]) → (TypeExpr → TypeExpr) → TypeDecl → TypeDecl`

update type declaration

`updTypeName :: ((String,String) → (String,String)) → TypeDecl → TypeDecl`

update name of type declaration

`updTypeVisibility :: (Visibility → Visibility) → TypeDecl → TypeDecl`

update visibility of type declaration

`updTypeParams :: ([Int] → [Int]) → TypeDecl → TypeDecl`

update type parameters of type declaration

`updTypeConsDecls :: ([ConsDecl] → [ConsDecl]) → TypeDecl → TypeDecl`

update constructor declarations of type declaration

`updTypeSynonym :: (TypeExpr → TypeExpr) → TypeDecl → TypeDecl`

update synonym of type declaration

`updQNamesInType :: ((String,String) → (String,String)) → TypeDecl → TypeDecl`

update all qualified names in type declaration

`trCons :: ((String,String) → Int → Visibility → [TypeExpr] → a) → ConsDecl → a`

transform constructor declaration

`consName :: ConsDecl → (String,String)`

get name of constructor declaration

`consArity :: ConsDecl → Int`

get arity of constructor declaration

`consVisibility :: ConsDecl → Visibility`

get visibility of constructor declaration

`consArgs :: ConsDecl → [TypeExpr]`

get arguments of constructor declaration

`updCons :: ((String,String) → (String,String)) → (Int → Int) → (Visibility → Visibility) → ([TypeExpr] → [TypeExpr]) → ConsDecl → ConsDecl`

update constructor declaration

`updConsName :: ((String,String) → (String,String)) → ConsDecl → ConsDecl`

update name of constructor declaration

`updConsArity :: (Int → Int) → ConsDecl → ConsDecl`

update arity of constructor declaration

`updConsVisibility :: (Visibility → Visibility) → ConsDecl → ConsDecl`

update visibility of constructor declaration

`updConsArgs :: ([TypeExpr] → [TypeExpr]) → ConsDecl → ConsDecl`

update arguments of constructor declaration

`updQNamesInConsDecl :: ((String,String) → (String,String)) → ConsDecl → ConsDecl`

update all qualified names in constructor declaration

`tVarIndex :: TypeExpr → Int`

get index from type variable

`domain :: TypeExpr → TypeExpr`

get domain from functional type

`range :: TypeExpr → TypeExpr`

get range from functional type

`tConsName :: TypeExpr → (String,String)`

get name from constructed type

`tConsArgs :: TypeExpr → [TypeExpr]`

get arguments from constructed type

`trTypeExpr :: (Int → a) → ((String,String) → [a] → a) → (a → a → a) → TypeExpr → a`

`isTVar :: TypeExpr → Bool`

    is type expression a type variable?

`isTCons :: TypeExpr → Bool`

    is type declaration a constructed type?

`isFuncType :: TypeExpr → Bool`

    is type declaration a functional type?

`updTVars :: (Int → TypeExpr) → TypeExpr → TypeExpr`

    update all type variables

`updTCons :: ((String,String) → [TypeExpr] → TypeExpr) → TypeExpr → TypeExpr`

    update all type constructors

`updFuncTypes :: (TypeExpr → TypeExpr → TypeExpr) → TypeExpr → TypeExpr`

    update all functional types

`argTypes :: TypeExpr → [TypeExpr]`

    get argument types from functional type

`resultType :: TypeExpr → TypeExpr`

    get result type from (nested) functional type

`rnmAllVarsInTypeExpr :: (Int → Int) → TypeExpr → TypeExpr`

    rename variables in type expression

`updQNamesInTypeExpr :: ((String,String) → (String,String)) → TypeExpr → TypeExpr`

    update all qualified names in type expression

`trOp :: ((String,String) → Fixity → Int → a) → OpDecl → a`

    transform operator declaration

`opName :: OpDecl → (String,String)`

    get name from operator declaration

`opFixity :: OpDecl → Fixity`

    get fixity of operator declaration

`opPrecedence :: OpDecl → Int`

get precedence of operator declaration

updOp :: ((String,String) → (String,String)) → (Fixity → Fixity) → (Int →
Int) → OpDecl → OpDecl

update operator declaration

updOpName :: ((String,String) → (String,String)) → OpDecl → OpDecl

update name of operator declaration

updOpFixity :: (Fixity → Fixity) → OpDecl → OpDecl

update fixity of operator declaration

updOpPrecedence :: (Int → Int) → OpDecl → OpDecl

update precedence of operator declaration

trFunc :: ((String,String) → Int → Visibility → TypeExpr → ARule a → b) →
AFuncDecl a → b

transform function

funcName :: AFuncDecl a → (String,String)

get name of function

funcArity :: AFuncDecl a → Int

get arity of function

funcVisibility :: AFuncDecl a → Visibility

get visibility of function

funcType :: AFuncDecl a → TypeExpr

get type of function

funcRule :: AFuncDecl a → ARule a

get rule of function

updFunc :: ((String,String) → (String,String)) → (Int → Int) → (Visibility →
Visibility) → (TypeExpr → TypeExpr) → (ARule a → ARule a) → AFuncDecl a →
AFuncDecl a

update function

updFuncName :: ((String,String) → (String,String)) → AFuncDecl a → AFuncDecl a

update name of function

updFuncArity :: (Int → Int) → AFuncDecl a → AFuncDecl a

update arity of function

`updFuncVisibility :: (Visibility → Visibility) → AFuncDecl a → AFuncDecl a`

update visibility of function

`updFuncType :: (TypeExpr → TypeExpr) → AFuncDecl a → AFuncDecl a`

update type of function

`updFuncRule :: (ARule a → ARule a) → AFuncDecl a → AFuncDecl a`

update rule of function

`isExternal :: AFuncDecl a → Bool`

is function externally defined?

`allVarsInFunc :: AFuncDecl a → [Int]`

get variable names in a function declaration

`funcArgs :: AFuncDecl a → [(Int,a)]`

get arguments of function, if not externally defined

`funcBody :: AFuncDecl a → AExpr a`

get body of function, if not externally defined

`funcRHS :: AFuncDecl a → [AExpr a]`

`rnmAllVarsInFunc :: (Int → Int) → AFuncDecl a → AFuncDecl a`

rename all variables in function

`updQNamesInFunc :: ((String,String) → (String,String)) → AFuncDecl a →`
`AFuncDecl a`

update all qualified names in function

`updFuncArgs :: ([(Int,a)] → [(Int,a)]) → AFuncDecl a → AFuncDecl a`

update arguments of function, if not externally defined

`updFuncBody :: (AExpr a → AExpr a) → AFuncDecl a → AFuncDecl a`

update body of function, if not externally defined

`trRule :: (a → [(Int,a)] → AExpr a → b) → (a → String → b) → ARule a → b`

transform rule

`ruleArgs :: ARule a → [(Int,a)]`

get rules arguments if it's not external

```
ruleBody :: ARule a → AExpr a
```

get rules body if it's not external

```
ruleExtDecl :: ARule a → String
```

get rules external declaration

```
isRuleExternal :: ARule a → Bool
```

is rule external?

```
updRule :: (a → a) → ([(Int,a)] → [(Int,a)]) → (AExpr a → AExpr a) → (String
→ String) → ARule a → ARule a
```

update rule

```
updRuleArgs :: ([(Int,a)] → [(Int,a)]) → ARule a → ARule a
```

update rules arguments

```
updRuleBody :: (AExpr a → AExpr a) → ARule a → ARule a
```

update rules body

```
updRuleExtDecl :: (String → String) → ARule a → ARule a
```

update rules external declaration

```
allVarsInRule :: ARule a → [Int]
```

get variable names in a functions rule

```
rnmAllVarsInRule :: (Int → Int) → ARule a → ARule a
```

rename all variables in rule

```
updQNamesInRule :: ((String,String) → (String,String)) → ARule a → ARule a
```

update all qualified names in rule

```
trCombType :: a → (Int → a) → a → (Int → a) → CombType → a
```

transform combination type

```
isCombTypeFuncCall :: CombType → Bool
```

is type of combination FuncCall?

```
isCombTypeFuncPartCall :: CombType → Bool
```

is type of combination FuncPartCall?

```
isCombTypeConsCall :: CombType → Bool
```

is type of combination ConsCall?

```
isCombTypeConsPartCall :: CombType → Bool
```

is type of combination ConsPartCall?

```
missingArgs :: CombType → Int
```

```
varNr :: AExpr a → Int
```

get internal number of variable

```
literal :: AExpr a → Literal
```

get literal if expression is literal expression

```
combType :: AExpr a → CombType
```

get combination type of a combined expression

```
combName :: AExpr a → (String,String)
```

get name of a combined expression

```
combArgs :: AExpr a → [AExpr a]
```

get arguments of a combined expression

```
missingCombArgs :: AExpr a → Int
```

get number of missing arguments if expression is combined

```
letBinds :: AExpr a → [((Int,a),AExpr a)]
```

get indices of variables in let declaration

```
letBody :: AExpr a → AExpr a
```

get body of let declaration

```
freeVars :: AExpr a → [Int]
```

get variable indices from declaration of free variables

```
freeExpr :: AExpr a → AExpr a
```

get expression from declaration of free variables

```
orExps :: AExpr a → [AExpr a]
```

get expressions from or-expression

```
caseType :: AExpr a → CaseType
```

get case-type of case expression

`caseExpr :: AExpr a → AExpr a`

get scrutinee of case expression

`caseBranches :: AExpr a → [ABranchExpr a]`

`isVar :: AExpr a → Bool`

is expression a variable?

`isLit :: AExpr a → Bool`

is expression a literal expression?

`isComb :: AExpr a → Bool`

is expression combined?

`isLet :: AExpr a → Bool`

is expression a let expression?

`isFree :: AExpr a → Bool`

is expression a declaration of free variables?

`isOr :: AExpr a → Bool`

is expression an or-expression?

`isCase :: AExpr a → Bool`

is expression a case expression?

`trExpr :: (a → Int → b) → (a → Literal → b) → (a → CombType →`
`((String,String),a) → [b] → b) → (a → [((Int,a),b)] → b → b) → (a →`
`[(Int,a)] → b → b) → (a → b → b → b) → (a → CaseType → b → [c] → b)`
`→ (APattern a → b → c) → (a → b → TypeExpr → b) → AExpr a → b`

transform expression

`updVars :: (a → Int → AExpr a) → AExpr a → AExpr a`

update all variables in given expression

`updLiterals :: (a → Literal → AExpr a) → AExpr a → AExpr a`

update all literals in given expression

`updCombs :: (a → CombType → ((String,String),a) → [AExpr a] → AExpr a) →`
`AExpr a → AExpr a`

update all combined expressions in given expression

`updLets :: (a → [((Int,a),AExpr a)] → AExpr a → AExpr a) → AExpr a → AExpr a`

update all let expressions in given expression

`updFrees :: (a → [(Int,a)] → AExpr a → AExpr a) → AExpr a → AExpr a`

update all free declarations in given expression

`updOrs :: (a → AExpr a → AExpr a → AExpr a) → AExpr a → AExpr a`

update all or expressions in given expression

`updCases :: (a → CaseType → AExpr a → [ABranchExpr a] → AExpr a) → AExpr a → AExpr a`

update all case expressions in given expression

`updBranches :: (APattern a → AExpr a → ABranchExpr a) → AExpr a → AExpr a`

update all case branches in given expression

`updTypeds :: (a → AExpr a → TypeExpr → AExpr a) → AExpr a → AExpr a`

update all typed expressions in given expression

`isFuncCall :: AExpr a → Bool`

is expression a call of a function where all arguments are provided?

`isFuncPartCall :: AExpr a → Bool`

is expression a partial function call?

`isConsCall :: AExpr a → Bool`

is expression a call of a constructor?

`isConsPartCall :: AExpr a → Bool`

is expression a partial constructor call?

`isGround :: AExpr a → Bool`

is expression fully evaluated?

`allVars :: AExpr a → [Int]`

get all variables (also pattern variables) in expression

`rnmAllVars :: (Int → Int) → AExpr a → AExpr a`

rename all variables (also in patterns) in expression

`updQNames :: ((String,String) → (String,String)) → AExpr a → AExpr a`

update all qualified names in expression

```
trBranch :: (APattern a → AExpr a → b) → ABranchExpr a → b
```

transform branch expression

```
branchPattern :: ABranchExpr a → APattern a
```

get pattern from branch expression

```
branchExpr :: ABranchExpr a → AExpr a
```

get expression from branch expression

```
updBranch :: (APattern a → APattern a) → (AExpr a → AExpr a) → ABranchExpr a
→ ABranchExpr a
```

update branch expression

```
updBranchPattern :: (APattern a → APattern a) → ABranchExpr a → ABranchExpr a
```

update pattern of branch expression

```
updBranchExpr :: (AExpr a → AExpr a) → ABranchExpr a → ABranchExpr a
```

update expression of branch expression

```
trPattern :: (a → ((String,String),a) → [(Int,a)] → b) → (a → Literal → b)
→ APattern a → b
```

transform pattern

```
patCons :: APattern a → (String,String)
```

get name from constructor pattern

```
patArgs :: APattern a → [(Int,a)]
```

get arguments from constructor pattern

```
patLiteral :: APattern a → Literal
```

get literal from literal pattern

```
isConsPattern :: APattern a → Bool
```

is pattern a constructor pattern?

```
updPattern :: (((String,String),a) → ((String,String),a)) → ([(Int,a)] →
[(Int,a)]) → (Literal → Literal) → APattern a → APattern a
```

update pattern

```
updPatCons :: ((String,String) → (String,String)) → APattern a → APattern a
```

update constructors name of pattern

`updPatArgs :: ([(Int,a)] → [(Int,a)]) → APattern a → APattern a`

　　　　update arguments of constructor pattern

`updPatLiteral :: (Literal → Literal) → APattern a → APattern a`

　　　　update literal of pattern

`patExpr :: APattern a → AExpr a`

　　　　build expression from pattern

`annRule :: ARule a → a`

`annExpr :: AExpr a → a`

　　　　Extract the annotation of an annotated expression.

`annPattern :: APattern a → a`

　　　　Extract the annotation of an annotated pattern.

`unAnnProg :: AProg a → Prog`

`unAnnFuncDecl :: AFuncDecl a → FuncDecl`

`unAnnRule :: ARule a → Rule`

`unAnnExpr :: AExpr a → Expr`

`unAnnPattern :: APattern a → Pattern`

A.6.18　Library FlatCurry.Annotated.TypeSubst

Type substitutions on type-annotated AnnotatedFlatCurry

Exported types:

`type AFCSubst = FM Int TypeExpr`

　　　　The (abstract) data type for substitutions on TypeExpr.

Exported functions:

`showAFCSubst :: FM Int TypeExpr → String`

`emptyAFCSubst :: FM Int TypeExpr`

> The empty substitution

`lookupAFCSubst :: FM Int TypeExpr → Int → Maybe TypeExpr`

> Searches the substitution for a mapping from the given variable index to a term.

`substFunc :: FM Int TypeExpr → AFuncDecl TypeExpr → AFuncDecl TypeExpr`

> Applies a substitution to a function.

`substRule :: FM Int TypeExpr → ARule TypeExpr → ARule TypeExpr`

> Applies a substitution to a type expression.

`substExpr :: FM Int TypeExpr → AExpr TypeExpr → AExpr TypeExpr`

> Applies a substitution to a type expression.

`substSnd :: FM Int TypeExpr → (a,TypeExpr) → (a,TypeExpr)`

`substBranch :: FM Int TypeExpr → ABranchExpr TypeExpr → ABranchExpr TypeExpr`

> Applies a substitution to a branch expression.

`substPattern :: FM Int TypeExpr → APattern TypeExpr → APattern TypeExpr`

> Applies a substitution to a pattern.

`subst :: FM Int TypeExpr → TypeExpr → TypeExpr`

> Looks up a type in a substitution and converts the resulting Term to a TypeExpr. Returns a given default value if the lookup fails.

A.6.19   Library FlatCurry.Annotated.TypeInference

Library to annotate the expressions of a FlatCurry program with type information.
It can be used by any other Curry program which processes or transforms FlatCurry programs. The main operation to use is

`inferProg :: Prog -> IO (Either String (AProg TypeExpr))`

which annotates a FlatCurry program with type information.
The type inference works in several steps:

1. For each known function and constructor, either imported or defined in the module itself, the respective type is inserted into a type environment (type assumption).

2. Every subexpression is annotated with a fresh type variable, whereas constructor and function names are annotated with a fresh variant of the type in the type assumption.

3. Based on FlatCurry's type inference rules, type equations are generated for a function's rule.

4. The resulting equations are solved using unification and the resulting substitution is applied to the function rule.

5. The inferred types are then normalized such that for every function rule the type variables start with 0.

In addition, the function `inferNewFunctions` allows to infer the types of a list of functions whose type is not known before. Consequently, this disallows polymorphic recursive functions. Those functions are separated into strongly connected components before their types are inferred to allow mutually recursive function definitions.
In case of any error, the type inference quits with an error message.

Exported types:

```
type TypeEnv = FM (String,String) TypeExpr
```

   A type environment.

Exported functions:

```
inferProg :: Prog → IO (Either String (AProg TypeExpr))
```

   Infers the type of a whole program.

```
inferProgFromProgEnv :: [(String,Prog)] → Prog → Either String (AProg TypeExpr)
```

   Infers the type of a whole program w.r.t. a list of imported modules.

```
inferFunction :: Prog → (String,String) → IO (Either String (AFuncDecl
TypeExpr))
```

   Infers the types of a single function specified by its qualified name.

```
inferNewFunctions :: Prog → [FuncDecl] → IO (Either String [AFuncDecl
TypeExpr])
```

   Infers the types of a group of (possibly mutually recursive) functions. Note that the functions are only monomorphically instantiated, i.e., polymorphic recursion is not supported. The given type may be too general, for instance a type variable only, and will be specialised to the inferred type.

```
inferExpr :: Prog → Expr → IO (Either String (AExpr TypeExpr))
```

Infer the type of a single expression.

`inferProgEnv :: FM (String,String) TypeExpr → Prog → Either String (AProg TypeExpr)`

> Infers the type of a whole program. Uses the given type environment instead of generating a new one.

`inferFunctionEnv :: FM (String,String) TypeExpr → Prog → (String,String) → Either String (AFuncDecl TypeExpr)`

> Infers the types of a single function specified by its qualified name. Uses the given type environment instead of generating a new one.

`inferNewFunctionsEnv :: FM (String,String) TypeExpr → String → [FuncDecl] → Either String [AFuncDecl TypeExpr]`

> Infers the types of a group of (possibly mutually recursive) functions. Note that the functions are only monomorphically instantiated, i.e., polymorphic recursion is not supported. The given type may be too general, for instance a type variable only, and will be specialised to the inferred type.

`inferExprEnv :: FM (String,String) TypeExpr → Expr → Either String (AExpr TypeExpr)`

> Infers the types of a single expression. Uses the given type environment instead of generating a new one.

`getTypeEnv :: Prog → IO (FM (String,String) TypeExpr)`

> Extract the type environment from the given Prog.

`getTypeEnvFromProgEnv :: [(String,Prog)] → Prog → Either String (FM (String,String) TypeExpr)`

> Extract the type environment from the given Prog by lookup in a module name -> Prog environment.

A.6.20 Library CurryStringClassifier

The Curry string classifier is a simple tool to process strings containing Curry source code. The source string is classified into the following categories:

- moduleHead - module interface, imports, operators

- code - the part where the actual program is defined

- big comment - parts enclosed in {- ... -}

- small comment - from "−" to the end of a line

- text - a string, i.e. text enclosed in "..."

- letter - the given string is the representation of a character

- meta - containing information for meta programming

For an example to use the state scanner cf. addtypes, the tool to add function types to a given program.

Exported types:

```
type Tokens = [Token]
```

```
data Token
```

The different categories to classify the source code.

Exported constructors:

- SmallComment :: String → Token

- BigComment :: String → Token

- Text :: String → Token

- Letter :: String → Token

- Code :: String → Token

- ModuleHead :: String → Token

- Meta :: String → Token

Exported functions:

```
isSmallComment :: Token → Bool
```

test for category "SmallComment"

```
isBigComment :: Token → Bool
```

test for category "BigComment"

```
isComment :: Token → Bool
```

test if given token is a comment (big or small)

```
isText :: Token → Bool
```

test for category "Text" (String)

```
isLetter :: Token → Bool
```

test for category "Letter" (Char)

`isCode :: Token → Bool`

test for category "Code"

`isModuleHead :: Token → Bool`

test for category "ModuleHead", ie imports and operator declarations

`isMeta :: Token → Bool`

test for category "Meta", ie between $\{+$ and $+\}$

`scan :: String → [Token]`

Divides the given string into the six categories. For applications it is important to know whether a given part of code is at the beginning of a line or in the middle. The state scanner organizes the code in such a way that every string categorized as "Code" **always** starts in the middle of a line.

`plainCode :: [Token] → String`

Yields the program code without comments (but with the line breaks for small comments).

`unscan :: [Token] → String`

Inverse function of scan, i.e., unscan (scan x) = x. unscan is used to yield a program after changing the list of tokens.

`readScan :: String → IO [Token]`

return tokens for given filename

`testScan :: String → IO ()`

test whether (unscan . scan) is identity

B Markdown Syntax

This document describes the syntax of texts containing markdown elements. The markdown syntax is intended to simplify the writing of texts whose source is readable and can be easily formatted, e.g., as part of a web document. It is a subset of the original markdown syntax (basically, only internal links and pictures are missing) supported by the Curry library Markdown.

B.1 Paragraphs and Basic Formatting

Paragraphs are separated by at least one line which is empty or does contain only blanks.

Inside a paragraph, one can *emphasize* text or also **strongly emphasize** text. This is done by wrapping it with one or two _ or * characters:

```
_emphasize_
*emphasize*
__strong__
**strong**
```

Furthermore, one can also mark `program code` text by backtick quotes (`):

```
The function `fib` computes Fibonacci numbers.
```

Web links can be put in angle brackets, like in the link http://www.google.com:

```
<http://www.google.com>
```

Currently, only links starting with 'http' are recognized (so that one can also use HTML markup). If one wants to put a link under a text, one can put the text in square brackets directly followed by the link in round brackets, as in Google:

```
[Google](http://www.google.com)
```

If one wants to put a character that has a specific meaning in the syntax of Markdown, like * or _, in the output document, it should be escaped with a backslash, i.e., a backslash followed by a special character in the source text is translated into the given character (this also holds for program code, see below). For instance, the input text

```
\_word\_
```

produces the output "_word_". The following backslash escapes are recognized:

```
\    backslash
`    backtick
*    asterisk
_    underscore
{}   curly braces
[]   square brackets
```

```
()   parentheses
#    hash symbol
+    plus symbol
-    minus symbol (dash)
.    dot
     blank
!    exclamation mark
```

B.2   Lists and Block Formatting

An **unordered list** (i.e., without numbering) is introduced by putting a star in front of the list elements (where the star can be preceded by blanks). The individual list elements must contain the same indentation, as in

```
* First list element
  with two lines

* Next list element.

  It contains two paragraphs.

* Final list element.
```

This is formatted as follows:

- First list element with two lines

- Next list element.

  It contains two paragraphs.

- Final list element.

Instead of a star, one can also put dashes or plus to mark unordered list items. Furthermore, one could nest lists. Thus, the input text

```
- Color:
  + Yellow
  + Read
  + Blue
- BW:
  + Black
  + White
```

is formatted as

- Color:

        – Yellow

        – Read

        – Blue

- BW:

        – Black

        – White

Similarly, **ordered lists** (i.e., with numbering each item) are introduced by a number followed by a dot and at least one blank. All following lines belonging to the same numbered item must have the same indent as the first line. The actual value of the number is not important. Thus, the input

```
1. First element

99. Second
    element
```

is formatted as

1. First element

2. Second element

A quotation block is marked by putting a right angle followed by a blank in front of each line:

```
> This is
> a quotation.
```

It will be formatted as a quote element:

> This is a quotation.

A block containing **program code** starts with a blank line and is marked by intending each input line by *at least four spaces* where all following lines must have at least the same indentation as the first non-blank character of the first line:

```
    f x y = let z = (x,y)
            in (z,z)
```

The indentation is removed in the output:

```
f x y = let z = (x,y)
        in (z,z)
```

To visualize the structure of a document, one can also put a line containing only blanks and at least three dashes (stars would also work) in the source text:

```
--------------------------------------------------
```

This is formatted as a horizontal line:

B.3 Headers

The are two forms to mark headers. In the first form, one can "underline" the main header in the source text by equal signs and the second-level header by dashes:

```
First-level header
==================

Second-level header
-------------------
```

Alternatively (and for more levels), one can prefix the header line by up to six hash characters, where the number of characters corresponds to the header level (where level 1 is the main header):

```
# Main header

## Level 2 header

### Level 3

#### Level 4

##### Level 5

###### Level 6
```

C 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. 12.2). The grammar satisfies the LL(1) property and is influenced by the SQLite dialect.[12]

```
--------------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 --------------------------------
```

```
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 ::= ('"'|''')
```

D 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.

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 meta-programming 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.

If the Curry module M is stored in the directory dir, the corresponding FlatCurry program is stored in the directory "dir/`.curry`". 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 FlatCurry program is stored in "dir/`.curry/D1/D2/M.fcy`".

`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).

E 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.[13] 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`):

---

[13]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`[14] 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`.[15]

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

---

[14]Note that all translated Curry modules are imported in the Haskell code fully qualified in order to avoid name conflicts.

[15]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

373

397