KiCS2
The Kiel Curry System (Version 2)

User Manual

Version 0.4.0 of 2015-03-06

Michael Hanus\textsuperscript{1} [editor]

Additional Contributors:

Bernd Braßel\textsuperscript{2}
Björn Peemöller\textsuperscript{3}
Fabian Reck\textsuperscript{4}

\textsuperscript{(1)} University of Kiel, Germany, mh@informatik.uni-kiel.de
\textsuperscript{(2)} University of Kiel, Germany, bbr@informatik.uni-kiel.de
\textsuperscript{(3)} University of Kiel, Germany, bjp@informatik.uni-kiel.de
\textsuperscript{(4)} University of Kiel, Germany, fre@informatik.uni-kiel.de
## Contents

**Preface**  
5  

1 **Overview of KiCS2**  
1.1 Installation ............................................ 6  
1.2 General Use ............................................. 6  
1.3 Restrictions ............................................. 7  
1.4 Modules in KiCS2 ....................................... 8  

2 **Using the Interactive Environment of KiCS2**  
2.1 Invoking KiCS2 ........................................... 9  
2.2 Commands of KiCS2 ...................................... 9  
2.3 Options of KiCS2 ....................................... 11  
2.4 Source-File Options .................................... 15  
2.5 Using KiCS2 in Batch Mode ............................. 15  
2.6 Command Line Editing .................................. 16  
2.7 Customization .......................................... 16  
2.8 Emacs Interface ....................................... 16  

3 **Extensions** ............................................. 17  
3.1 Narrowing on Int Literals ............................... 17  
3.2 Recursive Variable Bindings ........................... 17  
3.3 Functional Patterns .................................... 18  
3.4 Order of Pattern Matching ............................ 19  
3.5 Datatypes with Field Labels .......................... 19  
3.5.1 Declaration of Constructors with Labeled Fields  20  
3.5.2 Field Selection ..................................... 20  
3.5.3 Construction Using Field Labels .................... 21  
3.5.4 Updates Using Field Labels ........................ 22  
3.5.5 Pattern Matching Using Field Labels ............... 22  
3.5.6 Field Labels and Modules .......................... 23  

4 **Recognized Syntax of Curry**  
4.1 Notational Conventions .................................. 24  
4.2 Lexicon .................................................. 24  
4.2.1 Case Mode .......................................... 24  
4.2.2 Identifiers and Keywords ............................ 24  
4.2.3 Comments ............................................ 25  
4.2.4 Numeric and Character Literals .................... 25  
4.3 Layout ................................................... 26  
4.4 Context Free Grammar .................................. 27  

5 **CurryDoc: A Documentation Generator for Curry Programs**  
30
6 CurryBrowser: A Tool for Analyzing and Browsing Curry Programs
7 CurryTest: A Tool for Testing Curry Programs
8 ERD2Curry: A Tool to Generate Programs from ER Specifications
9 Spicey: An ER-based Web Framework
10 Technical Problems

Bibliography

A Libraries of the KiCS2 Distribution

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 AbstractCurry and FlatCurry: Meta-Programming in Curry</td>
<td>42</td>
</tr>
<tr>
<td>A.2 General Libraries</td>
<td></td>
</tr>
<tr>
<td>A.2.1 Library AllSolutions</td>
<td>43</td>
</tr>
<tr>
<td>A.2.2 Library Assertion</td>
<td>44</td>
</tr>
<tr>
<td>A.2.3 Library Char</td>
<td>45</td>
</tr>
<tr>
<td>A.2.4 Library Combinatorial</td>
<td>47</td>
</tr>
<tr>
<td>A.2.5 Library Constraint</td>
<td>47</td>
</tr>
<tr>
<td>A.2.6 Library CPNS</td>
<td>48</td>
</tr>
<tr>
<td>A.2.7 Library CSV</td>
<td>49</td>
</tr>
<tr>
<td>A.2.8 Library Debug</td>
<td>49</td>
</tr>
<tr>
<td>A.2.9 Library Directory</td>
<td>50</td>
</tr>
<tr>
<td>A.2.10 Library Distribution</td>
<td>51</td>
</tr>
<tr>
<td>A.2.11 Library Either</td>
<td>56</td>
</tr>
<tr>
<td>A.2.12 Library FileGoodies</td>
<td>57</td>
</tr>
<tr>
<td>A.2.13 Library FilePath</td>
<td>58</td>
</tr>
<tr>
<td>A.2.14 Library Float</td>
<td>62</td>
</tr>
<tr>
<td>A.2.15 Library Function</td>
<td>64</td>
</tr>
<tr>
<td>A.2.16 Library GetOpt</td>
<td>65</td>
</tr>
<tr>
<td>A.2.17 Library Global</td>
<td>66</td>
</tr>
<tr>
<td>A.2.18 Library GUI</td>
<td>67</td>
</tr>
<tr>
<td>A.2.19 Library Integer</td>
<td>80</td>
</tr>
<tr>
<td>A.2.20 Library IO</td>
<td>82</td>
</tr>
<tr>
<td>A.2.21 Library IOExts</td>
<td>84</td>
</tr>
<tr>
<td>A.2.22 Library JavaScript</td>
<td>86</td>
</tr>
<tr>
<td>A.2.23 Library KeyDatabaseSQLite</td>
<td>89</td>
</tr>
<tr>
<td>A.2.24 Library List</td>
<td>94</td>
</tr>
<tr>
<td>A.2.25 Library Maybe</td>
<td>97</td>
</tr>
<tr>
<td>A.2.26 Library NamedSocket</td>
<td>98</td>
</tr>
<tr>
<td>A.2.27 Library Parser</td>
<td>100</td>
</tr>
<tr>
<td>A.2.28 Library Pretty</td>
<td>101</td>
</tr>
<tr>
<td>A.2.29 Library Profile</td>
<td>110</td>
</tr>
<tr>
<td>A.2.30 Library Prolog</td>
<td>111</td>
</tr>
</tbody>
</table>
Preface

This document describes KiCS2 (Kiel Curry System Version 2), an implementation of the multi-paradigm language Curry [8, 17] 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 [9, 10, 11, 14]).

We assume familiarity with the ideas and features of Curry as described in the Curry language definition [18]. 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 0.8.3). The basic ideas of the implementation of KiCS2 can be found in [7, 6].

Acknowledgements

This work has been supported in part by the DFG grants Ha 2457/5-1 and Ha 2457/5-2.
1 Overview of KiCS2

1.1 Installation

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

Installation instructions for KiCS2 can be found in the file INSTALL.txt stored in the KiCS2 installation directory. Note that there are two possibilities to install KiCS2:

Global installation: KiCS2 is installed in some global system directory where users have no write permission. In this case, some options for experimenting with KiCS2 (like supply or ghc, see below) are not available (since they require the recompilation of parts of the installed system).

Local installation: KiCS2 is installed in some local user directory where the user has write permission and the option GLOBALINSTALL in the Makefile of the KiCS2 installation is set as follows:

GLOBALINSTALL=no

In this case, all options of KiCS2 are available.

Furthermore, KiCS2 can be installed with experimental support for profiling of executables. To use profiling, two requirements have to be met:

- The libraries that are shipped with the GHC that is used by KiCS2 have to be installed with profiling enabled. This is the default for the system libraries contained in the GHC release, but may not be the case for additional libraries.

- The Makefile of KiCS2 contains an option PROFILING which has to be set to yes to enable profiling support. You may either change the Makefile to

  PROFILING = yes

  or specify this setting while starting the installation process using

  make <optional target> PROFILING=yes

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

1.2 General Use

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

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

Since the translation of Curry programs with KiCS2 creates some auxiliary files (see Section C for details), you need write permission in the directory where you have stored your Curry programs. Moreover, the current implementation also recompiles system libraries according to the setting of
some options. Therefore, the KiCS2 system should be locally installed in your user account. The auxiliary files for all Curry programs in the current directory can be deleted by the command

\begin{verbatim}
cleancurry
\end{verbatim}

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

\begin{verbatim}
cleancurry -r
\end{verbatim}

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 Curry-Browser, 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 [16] 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 [4] 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

  \begin{verbatim}
getAllValues :: a -> IO [a]
\end{verbatim}

to compute all values of a given argument expression. There is also the library `SearchTree` (Section A.2.35) which supports user-programmable search strategies and contains some predefined strategies like depth-first, breadth-first, iterative deepening search.

  **Weak encapsulation:** This means that only the non-determinism defined inside an encapsulation operator is encapsulated. Conceptually, these operators are offered as `set functions` [2] which compute the set of all results but do not encapsulate non-determinism in the actual arguments. See the library `SetFunctions` (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
\[ x == \text{True} \text{ where } x \text{ free} \]

is non-deterministically evaluated to \text{False} and \text{True} by instantiating \( x \) to \text{False} and \text{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

\[ x*y=:=1 \text{ where } x,y \text{ free} \]

is non-deterministically evaluated to the two solutions

- \( \{ x = -1, y = -1 \} \) Success
- \( \{ x = 1, y = 1 \} \) Success

- 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 directories “\text{kics2home/lib}” and “\text{kics2home/lib/meta}”. This search path can be extended by setting the environment variable \text{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 \text{CURRYPATH}.
3. The directories enumerated in the “.kics2rc” variable “libraries”.
4. The directories “\text{kics2home/lib}” and “\text{kics2home/lib/meta}”.

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 \text{MAINDIR} and imports the module \text{Test.Func}, then the module stored in \text{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 (\text{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 \text{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” (this is a shell script 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.

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.
- **:add m1...mn** Add modules m1,...,mn 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, they must be introduced by a \texttt{where...free} declaration. For instance, one can write
\begin{verbatim}
not b where b free
\end{verbatim}
in order to obtain the following bindings and results:
\begin{verbatim}
\{b = False\} True
\{b = True\} False
\end{verbatim}
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
\begin{verbatim}
xs++ys=:=[z] where xs,ys,z free
\end{verbatim}
they are specialized to the type \texttt{"\(\varnothing\)"} (since the current implementation of KiCS2 does not support computations with polymorphic logic variables).

- \texttt{:eval expr} Same as \texttt{expr}. This command might be useful when putting commands as arguments when invoking \texttt{kics2}.
- \texttt{:quit} Exit the system.

There are also a number of further commands that are often useful:
- \texttt{:type expr} Show the type of the expression \texttt{expr}.
- \texttt{:programs} Show the list of all Curry programs that are available in the load path.
- \texttt{:cd dir} Change the current working directory to \texttt{dir}.
- \texttt{:edit} Load the source code of the current main module into a text editor. If the variable \texttt{editcommand} is set in the configuration file \texttt{".kics2rc"} (see Section 2.7), its value is used as an editor command, otherwise the environment variable \texttt{EDITOR} is used as the editor program.
- \texttt{:edit m} Load the source text of module \texttt{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 \texttt{":edit"}.
- \texttt{:show} Show the source text of the currently loaded Curry program. If the variable \texttt{showcommand} is set in the configuration file \texttt{".kics2rc"} (see Section 2.7), its value is used as a command to show the source text, otherwise the command \texttt{"cat"} is used.
- \texttt{:show m} Show the source text of module \texttt{m} which must be accessible via the current load path if no path specification is given.

\footnote{Currently, bindings are only printed if the initial expression is not an I/O action (i.e., not of type \texttt{"IO..."}) and there are not more than ten free variables in the initial expression.}
:source $f$ Show the source code of function $f$ (which must be visible in the currently loaded module) in a separate window.

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

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

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

:interface $m$ Similar to “:interface” but shows the interface of the module $m$ which must be in the load path of KiCS2.

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

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

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

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

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

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

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

2.3 Options of KiCS2

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

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

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

:set path aux:~/tests
Relative directory names are replaced by absolute ones so that the path is independent of later changes of the current working directory.

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

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

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

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

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

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

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

**choices n** Show the internal choice structure (according to the implementation described in [7]) resulting from the complete evaluation of the main expression in a tree-like structure. This mode is only useful for debugging or understanding the implementation of non-deterministic evaluations used in KiCS2. If the optional argument n is provided, the tree is shown up to depth n.

**supply i** (not available in global installations, see Section 1.1) Use implementation i as the identifier supply for choice structures (see [7] for a detailed explanation). Currently, the following values for i are supported:

- **integer**: Use unbounded integers as choice identifiers. This implementation is described in [7].
- **ghc**: Use a more sophisticated implementation of choice identifiers (based on the ideas described in [3]) 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 (default)**: 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 search methods.

**vn** Set the verbosity level to n. The following values are allowed for n:
$n = 0$: Do not show any messages (except for errors).

$n = 1$: Show only messages of the front-end, like loading of modules.

$n = 2$: Show also messages of the back end, like compilation messages from the Haskell compiler.

$n = 3$: Show also intermediate messages and commands of the compilation process.

$n = 4$: Show also all intermediate results of the compilation process.

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

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

`+/-interactive` Turn on/off the interactive mode. In the interactive mode, the next nondeterministic 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]
```

`+/-ghci` Turn on/off the ghci mode. In the ghci mode, the initial goal is send to the interactive version of the Glasgow Haskell Compiler. This might result in a slower execution but in a faster startup time since the linker to create the main executable is not used.
**safe** Turn on the safe execution mode. In the safe execution mode, the initial goal is not allowed to be of type `IO` and the program should not import the module `Unsafe`. Furthermore, the allowed commands are `eval`, `load`, `quit`, and `reload`. This mode is useful to use KiCS2 in uncontrolled environments, like a computation service in a web page, where KiCS2 could be invoked by

```
  kics2 :set safe
```

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

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

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

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

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

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

**ghc opts** Define additional options passed to the Glasgow Haskell Compiler (GHC) when the generated Haskell programs are compiled. Many options necessary to compile Curry programs are already set (you can see them by setting the verbosity level to 2 or greater). One has to be careful when providing additional options. For instance, in a global installation of KiCS2 (see Section 1.1), 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 local installations (see Section 1.1) 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 (\(\doteq\)) 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
Define additional run-time options passed to the executable generated by the Glasgow Haskell Compiler, i.e., the parameters “+RTS o -RTS” are passed to the executable. For instance, setting the option

```
:set rts -H512m
```

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

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.2.38) 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:
The “:quit” directive is necessary to avoid KiCS2 going into interactive mode after the execution 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:

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

### 2.6 Command Line Editing

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

### 2.7 Customization

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

### 2.8 Emacs Interface

Emacs is a powerful programmable editor suitable for program development. It is freely available for many platforms (see [http://www.emacs.org](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

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

Note that the free variables occurring 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:

\[
\text{make [kernel|install] RUNTIMEFLAGS=BinaryInt}
\]

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

\[
:set \text{ ghc -DBinaryInt}
\]

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

\[
Prelude> x == 3 \text{ where } x \text{ free} \\
\{x = (\text{Neg} \ _x2)\} \text{ False} \\
\{x = 0\} \text{ False} \\
\{x = 1\} \text{ False} \\
\{x = (\text{Pos} (0 \ _x3))\} \text{ False} \\
\{x = 3\} \text{ True} \\
\{x = (\text{Pos} (I (0 \ _x4)))\} \text{ False} \\
\{x = (\text{Pos} (I (I \ _x4)))\} \text{ False}
\]

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 0 and I denote a 0– and 1–bit where the least significant bit comes first. That is, \((\text{Pos} (I (0 \ _x4))) = +(I (0 \ _x4)) = +(2 * (0 \ _x4)) + 1 = +(4 * _x4) + 1\) which meets the output above.

3.2 Recursive Variable Bindings

Local variable declarations (introduced by let or where) can be (mutually) recursive in KiCS2. For instance, the declaration
ones5 = let ones = 1 : ones
    in take 5 ones

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

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

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

3.3 Functional Patterns

Functional patterns [1] are a useful extension to code 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:

\[
\text{last } xs \mid _++[y] =:= xs = y \quad \text{where } y \text{ free}
\]

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:

\[
\text{last } (_++[y]) = y
\]

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

\[
\begin{align*}
\text{last } [y] &= y \\
\text{last } [,y] &= y \\
\text{last } [,,,y] &= y \\
\end{align*}
\]

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

In general, a pattern of the form \((f \ t_1\ldots t_n) \ (n > 0)\) 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.

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
\( f (_ \; ++ \; x \otimes [(42,\_)] \; ++ \; \_) = x \)

is considered as syntactic sugar for the expanded definition

\[
\begin{align*}
  f (_ \; ++ \; x \; ++ \; \_) &= f' \; x \\
  \text{where} \\
  f' \; [(42,\_)] &= x
\end{align*}
\]

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

### 3.4 Order of Pattern Matching

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

\[ f (_\;++[x]++++[x]++\_) = x \]

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

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

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

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

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

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

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

### 3.5 Datatypes with Field Labels

A datatype declaration may optionally define data constructors with field labels.\(^2\) These field labels can be used to construct, select from, and update fields in a manner that is independent of the overall structure of the datatype.

\(^2\)Field labels are quite similar to Haskell \([\?]\) so that we adopt most of the description of Haskell here.
3.5.1 Declaration of Constructors with Labeled Fields

A data constructor of arity \( n \) creates an object with \( n \) components. These components are normally accessed positionally as arguments to the constructor in expressions or patterns. For large datatypes it is useful to assign field labels to the components of a data object. This allows a specific field to be referenced independently of its location within the constructor. A constructor definition in a data declaration may assign labels to the fields of the constructor, using the record syntax \( c \{ \ldots \} \). Constructors using field labels may be freely mixed with constructors without them. A constructor with associated field labels may still be used as an ordinary constructor. The various use of labels (see below) are simply a shorthand for operations using an underlying positional constructor. The arguments to the positional constructor occur in the same order as the labeled fields.

Translation:

\[
\begin{align*}
[C \{ lts \}] &= C \llbracket lts \rrbracket \\
[l, lts] &= \llbracket l \rrbracket \llbracket lts \rrbracket \\
[l, ls :: t] &= t \llbracket ls :: t \rrbracket \\
[l :: t] &= t
\end{align*}
\]

For example, the definition using field labels

\[
data Person = Person \{ firstName, lastName :: String, age :: Int \} \\
| Agent \{ firstName, lastName :: String, trueIdentity :: Person \}
\]

is translated to

\[
data Person = Person String String Int \\
| Agent String String Person
\]

A data declaration may use the same field label in multiple constructors as long as the typing of the field is the same in all cases after type synonym expansion. A label cannot be shared by more than one type in scope. Field names share the top-level name space with ordinary definition of functions and must not conflict with other top-level names in scope.

Consider the following example:

\[
data S = S1 \{ x :: Int \} \mid S2 \{ x :: Int \} -- OK \\
data T = T1 \{ y :: Int \} \mid T2 \{ y :: Bool \} -- BAD
\]

Here S is legal but T is not, because \( y \) is given inconsistent typings in the latter.

3.5.2 Field Selection

Field labels are used as selector functions, i.e., each field label serves as a function that extracts the field from an object. Selectors are top-level bindings and so they may be shadowed by local variables but cannot conflict with other top-level bindings of the same name. This shadowing only affects selector functions; in record construction (Section 3.5.3) and update (Section 3.5.4), field labels cannot be confused with ordinary variables.
A field label \( lab \) introduces a selector function defined as:

\[
lab (C_1 p_{11} \ldots p_{1k_1}) = x \\
\vdots \\
lab (C_n p_{n1} \ldots p_{nk_n}) = x
\]

where \( C_1 \ldots C_n \) are all the constructors of the datatype containing a field labeled with \( lab \), \( p_{ij} \) is \( x \) when \( lab \) labels the \( j \)th component of \( C_i \) or \( _{-} \) otherwise.

For example the definition of Person above introduces the selector functions

\[
\begin{align*}
\text{firstName} & \colon \text{Person} \to \text{String} \\
\text{firstName} (\text{Person} \ x \ _{-}) &= x \\
\text{firstName} (\text{Agent} \ x \ _{-}) &= x \\
\text{lastName} & \colon \text{Person} \to \text{String} \\
\text{lastName} (\text{Person} \ _{x} \ _{-}) &= x \\
\text{lastName} (\text{Agent} \ _{x} \ _{-}) &= x \\
\text{age} & \colon \text{Person} \to \text{Int} \\
\text{age} (\text{Person} \ _{-} \ x) &= x \\
\text{trueIdentity} & \colon \text{Person} \to \text{Person} \\
\text{trueIdentity} (\text{Agent} \ _{-} \ x) &= x
\end{align*}
\]

### 3.5.3 Construction Using Field Labels

A constructor with labeled fields may be used to construct a value in which the components are specified by name rather than by position. In this case, the components are enclosed by braces. Construction using field labels is subject to the following constraints:

- Only field labels declared with the specified constructor may be mentioned.
- A field label may not be mentioned more than once.
- Fields not mentioned are initialized to different free variables.

The expression \( C\{\} \), where \( C \) is a data constructor, is legal \textit{whether or not} \( C \) was declared with record syntax; it denotes \( C \text{ Prelude.unknown}_1 \ldots \text{ Prelude.unknown}_n \) where \( n \) is the arity of \( C \). Note that this will introduce the constructor \( C \) with \textit{n different} free variables as arguments.

\[
\textbf{Translation:} \text{ In the binding } f = v, \text{ the field } f \text{ labels } v. \\
C \{ \ bs \} = C \ (\text{pick}^C_1 \ bs \text{ Prelude.unknown}) \ldots \ (\text{pick}^C_k \ bs \text{ Prelude.unknown})
\]

where \( k \) is the arity of \( C \).

The auxiliary function \( \text{pick}^C_i \ bs \) is defined as follows:

If the \( i \)th component of a constructor \( C \) has the field label \( f \) and \( f = v \) appears in the binding list \( bs \), then \( \text{pick}^C_i \ bs \) is \( v \). Otherwise, \( \text{pick}^C_i \ bs \) is the default value \( d \).

For example, a Person can be constructed by

\[
\text{smith} = \text{Agent} \{ \text{lastName} = \text{"Smith"}, \text{firstName} = \text{"Agent"} \}
\]
which is equivalent to the following agent, whose true identity might be any person:

\[
\text{smith} = \text{Agent "Agent" "Smith" _ }
\]

3.5.4 Updates Using Field Labels

Values belonging to a datatype with field labels may be non-destructively updated. This creates a new value in which the specified field values replace those in the existing value. Updates are restricted in the following ways:

- All labels must be taken from the same datatype.
- No label may be mentioned more than once.
- The computation fails when the value being updated does not contain all of the specified labels.

**Translation:** Using the prior definition of pick,

\[
e \{ \text{bs} \} \rightarrow C_1 (\text{pick}_{k_1}^{C_1} \text{bs} v_1) \ldots (\text{pick}_{k_i}^{C_1} \text{bs} v_{k_1}) \\
\ldots \\
C_j v_1 \ldots v_{k_j} \rightarrow C_j (\text{pick}_{k_j}^{C_j} \text{bs} v_1) \ldots (\text{pick}_{k_j}^{C_j} \text{bs} v_{k_j})
\]

where \(\{C_1, \ldots, C_j\}\) is the set of constructors containing all labels in bs, \(k_i\) is the arity of \(C_i\).

For example, after watching a few more movies, we might want to update our information about smith. We can do so by writing

\[
\text{smith} \{ \text{trueIdentity} = \text{complement neo} \}
\]

which is equivalent to

\[
\text{fcase smith of} \\
\quad \text{Agent fn ln _} \rightarrow \text{Agent fn ln (complement neo)}
\]

3.5.5 Pattern Matching Using Field Labels

A constructor with labeled fields may be used to specify a pattern in which the components are identified by name rather than by position. Matching against a constructor using labeled fields is the same as matching ordinary constructor patterns except that the fields are matched in the order they are named in the field list. All listed fields must be declared by the constructor; fields may not be named more than once. Fields not named by the pattern are ignored (matched against _).

**Translation:** Using the prior definition of pick,

\[
C \{ \text{bs} \} = (C (\text{pick}_{k}^{C} \text{bs} _)) \ldots (\text{pick}_{k}^{C} \text{bs} _))
\]

where \(k\) is the arity of \(C\).

For example, we could define a Smith-tester by writing:
isSmith Agent { lastName = "Smith" } = success

which is equivalent to

isSmith (Agent _ "Smith" _) = success

3.5.6 Field Labels and Modules

As described in the Curry report, there are two forms of exporting a data type $T$: The simple name $T$ exports only the types name without constructors, whereas $T(...)$ also exports all constructors. Analogously, the form $T$ does not export any field labels, whereas $T(...)$ exports all constructors and all field labels.
4 Recognized Syntax of Curry

The KiCS2 Curry compiler accepts a slightly extended version of the grammar specified in the Curry Report [18]. 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:

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

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 Case Mode

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.

4.2.2 Identifiers and Keywords

\[
\begin{align*}
Letter ::= & \text{ any ASCII letter} \\
Dashes ::= & \text{ - } \\
Ident ::= & \text{Letter } \{ \text{Letter | Digit | } \_ \text{ | } \_ \} \\
Symbol ::= & \^ \text{ | } \& \text{ | } * \text{ | } + \text{ | } - \text{ | } = \text{ | } < \text{ | } > \text{ | } ? \text{ | } . \text{ | } / \text{ | } \_ \text{ | } \_ \text{ | } \_ \text{ | } : \\
ModuleID ::= & \{ \text{Ident . } \} \text{ Ident} \\
TypeConstrID ::= & \text{Ident} \\
DataConstrID ::= & \text{Ident} \\
TypeVarID ::= & \text{Ident } \_ \\
InfixOpID ::= & (\text{Symbol } \{ \text{Symbol} \})_{\langle \text{Dashes} \rangle} \\
FunctionID ::= & \text{Ident} \\
VariableID ::= & \text{Ident} \\
LabelID ::= & \text{Ident}
\end{align*}
\]
QTypeConstrID ::= [ModuleID .] TypeConstrID
QDataConstrID ::= [ModuleID .] DataConstrID
QInfixOpID ::= [ModuleID .] InfixOpID
QFunctionID ::= [ModuleID .] FunctionID
QVariableID ::= [ModuleID .] VariableID
QLabelID ::= [ModuleID .] LabelID

The following identifiers are recognized as keywords and cannot be used as an identifier:
case data do else external fcase foreign
free if import in infix infixl infixr
let module newtype of then type where

Note that the symbols as, hiding and qualified are not keywords. They have only a special
meaning in module headers and can be used as ordinary identifiers.
The following symbols also have a special meaning and cannot be used as an infix operator
identifier:
   ..  :  ::  =  \  |  <-  ->  @  ~

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

4.2.4 Numeric and Character Literals
Contrasting to the Curry Report, KiCS2 adopts Haskell’s notation of literals, for both numeric
literals as well as Char and String literals. The precise syntax for both kinds is given below.

Int ::= Decimal
      | 0b Binary | 0B Binary
      | 0o Octal | 00 Octal
      | 0x Hexadecimal | 0X Hexadecimal

Float ::= Decimal . Decimal [Exponent]
         | Decimal Exponent
Exponent ::= (e | E) [+ | -] Decimal

Decimal ::= Digit [Decimal]
Binary ::= Binit [Binary]
Octal ::= Octit [Octal]
Hexadecimal ::= Hexit [Hexadecimal]

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Binit ::= 0 | 1
Octit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Hexit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    | A | B | C | D | E | F | a | b | c | d | e | f

Char ::= ' (Graphic | \ | Space | Escape \&)'
String ::= " (Graphic | \ | Space | Escape | Gap)"
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. The indentation of a line is the indentation of its first symbol.\(^3\)

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 brackets (\{ \}) and the single entities are separated by semicolons (;). Instead of using the curly brackets and semicolons of the context-free syntax, a Curry programmer must 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 start 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 its 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

\[
  f \ x = h \ x \ where \ \{ \ g \ y = y + 1 ; h \ z = (g \ z) \ast 2 \ \} 
\]

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

\[
  f \ x = h \ x \ \where \ g \ y = y + 1 \\
  h \ z = (g \ z) \ast 2 
\]

or also as

\[
  f \ x = h \ x \ where \\
  g \ y = y + 1 \\
  h \ z = (g \ z) \ast 2 
\]

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

\(^3\)In order to determine the exact column number, we assume a fixed-width font with tab stops at each 8th column.
4.4 Context Free Grammar

\[
\begin{align*}
\text{Module} &::= \text{module ModuleID } [\text{Exports}] \text{ where Block} \\
&| \hspace{1em} \text{Block} \\
\text{ModuleID} &::= \text{see lexicon} \\
\text{Exports} &::= (\text{Export}_1, \ldots, \text{Export}_n) \hspace{1em} (n \geq 0) \\
\text{Export} &::= \text{QFunctionName} \\
&| \hspace{1em} \text{QTypeConstrID} [(\text{ConsLabelName}_1, \ldots, \text{ConsLabelName}_n)] \hspace{1em} (n \geq 0) \\
&| \hspace{1em} \text{QTypeConstrID} (\ldots) \\
&| \hspace{1em} \text{module ModuleID} \\
\text{ConsLabelName} &::= \text{LabelID} | \text{DataConstr} \\
\text{Block} &::= \{ [\text{ImportDecl}_1; \ldots; \text{ImportDecl}_k; \} \hspace{1em} \text{(no fixity declarations here)} \\
&| \hspace{1em} \text{BlockDeclaration}_1; \ldots; \text{BlockDeclaration}_n \hspace{1em} (k, n \geq 0) \\
\text{ImportDecl} &::= \text{import} [\text{qualified}] \text{ ModuleID } [\text{as ModuleID}] [\text{ImportRestr}] \\
\text{ImportRestr} &::= (\text{Import}_1, \ldots, \text{Import}_n) \hspace{1em} (n \geq 0) \\
&| \hspace{1em} \text{hiding} (\text{Import}_1, \ldots, \text{Import}_n) \hspace{1em} (n \geq 0) \\
\text{Import} &::= \text{FunctionName} \\
&| \hspace{1em} \text{TypeConstrID} [(\text{ConsLabelName}_1, \ldots, \text{ConsLabelName}_n)] \hspace{1em} (n \geq 0) \\
&| \hspace{1em} \text{TypeConstrID} (\ldots) \\
\text{BlockDeclaration} &::= \text{TypeSynonymDecl} \\
&| \hspace{1em} \text{DataDeclaration} \\
&| \hspace{1em} \text{FixityDeclaration} \\
&| \hspace{1em} \text{FunctionDeclaration} \\
\text{TypeSynonymDecl} &::= \text{type SimpleType } = \text{TypeExpr} \\
\text{SimpleType} &::= \text{TypeConstrID TypeVarID}_1 \ldots \text{TypeVarID}_n \hspace{1em} (n \geq 0) \\
\text{TypeConstrID} &::= \text{see lexicon} \\
\text{DataDeclaration} &::= \text{data SimpleType} \hspace{1em} \text{(external data type)} \\
&| \hspace{1em} \text{data SimpleType } = \text{ConstrDecl}_1 | \ldots | \text{ConstrDecl}_n \hspace{1em} (n > 0) \\
\text{ConstrDecl} &::= \text{DataConstr SimpleTypeExpr}_1 \ldots \text{SimpleTypeExpr}_n \hspace{1em} (n \geq 0) \\
&| \hspace{1em} \text{SimpleTypeExpr ConsOp TypeConsExpr} \hspace{1em} \text{(infix data constructor)} \\
&| \hspace{1em} \text{DataConstr} \{ \text{FieldDeclaration}_1, \ldots, \text{FieldDeclaration}_n \} \hspace{1em} (n \geq 0) \\
\text{FieldDeclaration} &::= \text{LabelID}_1, \ldots, \text{LabelID}_n :: \text{TypeExpr} \hspace{1em} (n > 0) \\
\text{LabelID} &::= \text{see lexicon} \\
\text{TypeExpr} &::= \text{TypeConsExpr } [\rightarrow \text{TypeExpr}] \\
\text{TypeConsExpr} &::= \text{QTypeConstrID SimpleTypeExpr}_1 \ldots \text{SimpleTypeExpr}_n \hspace{1em} (n > 0) \\
&| \hspace{1em} \text{SimpleTypeExpr} \\
\text{SimpleTypeExpr} &::= \text{TypeVarID} \\
&| \hspace{1em} \text{QTypeConstrID} \\
&| \hspace{1em} () \hspace{1em} \text{(unit type)} \\
&| \hspace{1em} (\text{TypeExpr}_1, \ldots, \text{TypeExpr}_n) \hspace{1em} \text{(tuple type, } n > 1) \\
&| \hspace{1em} [\text{TypeExpr}] \hspace{1em} \text{(list type)} \\
&| \hspace{1em} (\text{TypeExpr}) \hspace{1em} \text{(parenthesized type)} \\
\text{TypeVarID} &::= \text{see lexicon} \\
\text{FixityDeclaration} &::= \text{FixityKeyword Digit InfixOpID}_1, \ldots, \text{InfixOpID}_n \hspace{1em} (n > 0) \\
\text{FixityKeyword} &::= \text{infixl} | \text{infixr} | \text{infix} \\
\text{InfixOpID} &::= \text{see lexicon}
\end{align*}
\]
FunctionDeclaration ::= Signature | External | Equat

External ::= FunctionNames external (externally defined functions)

Signature ::= FunctionNames ::= TypeExpr
FunctionNames ::= FunctionName1 , ... , FunctionName_n (n > 0)

Equat ::= FunLHS = TypedExpr [where LocalDefs]
        | FunLHS CondExprs [where LocalDefs]

FunLHS ::= FunctionName SimplePat1 ... SimplePat_n (n ≥ 0)
        | SimplePat InfixOpID SimplePat

CondExprs ::= | InfixExpr = TypedExpr [CondExprs]

Pattern ::= ConsPattern [QConsOp Pattern] (infix constructor pattern)
ConsPattern ::= GDataConstr SimplePat1 ... SimplePat_n (constructor pattern)
        | SimplePat

SimplePat ::= Variable (wildcard)
        | - QDataConstr
        | Literal
        | - Int (negative pattern)
        | -. Float (negative float pattern)
        | () (empty tuple pattern)
        | ( Pattern1 , ... , Pattern_n ) (n > 1)
        | ( Pattern ) (parenthesized pattern)
        | [ Pattern1 , ... , Pattern_n ] (n ≥ 0)
        | Variable @ SimplePat (as-pattern)
        | ~ SimplePat (irrefutable pattern)
        | ( SimplePat QFunOp SimplePat ) (infix functional pattern)
        | ( QFunctionName SimplePat1 ... SimplePat_n ) (functional pattern, n > 0)
        | QDataConsr { FieldPat1 , ... , FieldPat_n } (labeled pattern, n ≥ 0)

FieldPat ::= QLabelID = Pattern
QLabelID ::= see lexicon

LocalDefs ::= { ValueDeclaration1 ; ... ; ValueDeclaration_n } (n > 0)
ValueDeclaration ::= FunctionDeclaration
        | PatternDeclaration
        | VariableID1 , ... , VariableID_n free (n > 0)
        | FixityDeclaration

PatternDeclaration ::= Pattern = TypedExpr [where LocalDefs]

TypedExpr ::= InfixExpr ::= TypeExpr (expression type signature)
        | InfixExpr
InfixExpr ::= Expr QOp InfixExpr (infix operator application)
        | - InfixExpr (unary int minus)
        | -. InfixExpr (unary float minus)
        | Expr

Expr ::= \ SimplePat1 ... SimplePat_n (lambda expression, n > 0)
        | let LocalDefs in TypedExpr (let expression)
        | if TypedExpr then TypedExpr else TypedExpr (conditional)
        | case TypedExpr of { Alt1 ; ... ; Alt_n } (case expression, n ≥ 0)
        | fcase TypedExpr of { Alt1 ; ... ; Alt_n } (fcase expression, n ≥ 0)
        | do { Stmt1 ; ... ; Stmt_n ; TypedExpr } (do expression, n ≥ 0)
FunctExpr ::=
  |  [FunctExpr] BasicExpr                  (function application)
  |  QVariableID                          (variable)
  |  _                                    (anonymous free variable)
  |  QFunctionName                        (qualified function)
  |  GDataConstr                          (general constructor)

BasicExpr ::=
  |  QVariableID                         (variable)
  |  _                                    (anonymous free variable)
  |  QFunctionName                       (qualified function)
  |  GDataConstr                          (general constructor)
  |  Literal                              (parenthesized expression)
  |  (TypedExpr)                         (tuple, \(n > 0\))
  |  [ TypedExpr1, ..., TypedExpr\(_n\) ] (finite list, \(n > 0\))
  |  [ TypedExpr | TypedExpr | ... | TypedExpr ] (list comprehension)
  |  [ TypedExpr | Qual\(_1\), ..., Qual\(_n\) ] (list comprehension, \(n \geq 1\))
  |  ( InfixExpr QOp )                   (left section)
  |  ( QOp⟨...,−,−⟩ InfixExpr )         (right section)
  |  QDataConstr \{ FBInd\(_1\), ..., FBInd\(_n\) \} (labeled construction, \(n \geq 0\))
  |  BasicExpr⟨QDataConstr⟩\{ FBInd\(_1\), ..., FBInd\(_n\) \} (labeled update, \(n > 0\))

Alt ::= Pattern -> TypedExpr [where LocalDefs]
  |  Pattern GdAlts [where LocalDefs]

GdAlts ::= | TypedExpr -> TypedExpr [GdAlts]

FBInd ::= QLabelID = TypedExpr

Qual ::= TypedExpr
  |  let LocalDefs
  |  Pattern <- TypedExpr

Stmt ::= TypedExpr
  |  let LocalDefs
  |  Pattern <- TypedExpr

Literal ::= Int | Char | String | Float

GDataConstr ::= ()
  |  []
  |  (,{},{})
  |  QDataConstr

FunctionName ::= FunctionID | (InfixOpID) (function)
QFunctionName ::= QFunctionID | (QInfixOpID) (qualified function)

Variable ::= VariableID | (InfixOpID) (variable)
DataConstr ::= DataConstrID | (InfixOpID) (constructor)
QDataConstr ::= QDataConstrID | (QConsOp) (qualified constructor)

QFunOp ::= QInfixOpID | 'QFunctionID' (qualified function operator)
ConsOp ::= InfixOpID | 'DataConstrID' (constructor operator)
QOp ::= QFunOp | QConsOp (qualified operator)
QConsOp ::= GConSym | 'QDataConstrID' (qualified constructor operator)
GConSym ::= : | QInfixOpID (general constructor symbol)
5 CurryDoc: 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 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.\(^4\) 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., “&lt;”). 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

---

\(^4\)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.
inside the generated HTML documentation. Such references have to be enclosed in single quotes. For instance, the text 'conc' refers to the Curry operation conc inside the current module whereas the text 'Prelude.reverse' refers to the operation reverse of the module Prelude. If one wants to write single quotes without this specific meaning, one can escape them with a backslash:

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

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

--- This isn’t a reference.

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

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

module Example where

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

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

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

To generate the documentation, execute the command

currydoc Example

(currydoc is a command usually stored in $kics2home/bin$ where $kics2home$ is the installation directory of KiCS2; see Section 1.2). 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

currydoc 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 currydoc with the following options:

`currydoc --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).

`currydoc --onlyindexhtml docdir Mod1 Mod2 ...Modn`: This command generates only the index pages (i.e., a main index page and index pages for all functions and constructors defined in the modules Mod1, Mod2, ..., Modn and their imported modules) in the directory docdir.
CurryBrowser: 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.\(^5\) 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 [12, 13].

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

- In the command shell via the command: `kics2home/bin/currybrowser 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 [19] or pattern matching is translated into case expressions [8, 21]. 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

\(^5\) 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.
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 CurryTest: A Tool for Testing Curry Programs

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:

- `trueValue :: String -> Bool -> Assertion ()`
- `assertEqual :: String -> a -> a -> Assertion a`
- `assertValues :: String -> a -> [a] -> Assertion a`
- `assertSolutions :: String -> (a -> Success) -> [a] -> Assertion a`
- `assertIO :: String -> IO a -> a -> Assertion a`
- `assertEqualIO :: String -> IO a -> IO a -> Assertion a`

The expression “`trueValue s b`” is an assertion (named `s`) that the expression `b` has the value `True`. Similarly, the expression “`assertEqual s e1 e2`” asserts that the expressions `e1` and `e2` must be equal (i.e., `e1==e2` 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 a1 a2`” asserts that the I/O actions `a1` and `a2` 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:

```haskell
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

```
currytest TestList
```

(`currytest` is a program stored in `kics2home/bin` where `kics2home` is the installation directory of KiCS2; see Section 1.2). 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

```
currytest --window TestList
```

or

```
currytest -w TestList
```

A snapshot of the interface is shown in Figure 2.
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 [5]. 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

```
erd2curry -x myerd.xmi
```

(erd2curry is a program stored in kics2home/bin where kics2home is the installation directory of KiCS2; see Section 1.2). 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 [5]. 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 create a textual description of the ERD as a Curry term of type ERD (w.r.t. the type definition given in module kics2home/currytools/erd2curry/ERD.curry) and store it in some file, e.g., “myerd.term”. This description can be compiled into a Curry program by the command

```
erd2curry -t myerd.term
```

The directory kics2home/currytools/erd2curry/ contains two examples for such ERD term files:

Blog.erdterm: This is a simple ERD model for a blog with entries, comments, and tags.

Uni.erdterm: This is an ERD model for university lectures as presented in the paper [5].

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

```
erd2curry -v myerd.term
```
9  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 [15]. 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 as a Curry term of type ERD (w.r.t. the type definitions given in module kics2home/currytools/erd2curry/ERD.curry) and store it in some file, e.g., “mymodel.erdterm”. The directory kics2home/currytools/spicey/ contains two examples for such ERD term files:

Blog.erdterm: This is a simple ER model for a blog with entries, comments, and tags, as presented in the paper [15].

Uni.erdterm: This is an ER model for university lectures as presented in the paper [5].

Then change to the directory in which you want to create the project sources. Execute the command

`spiceup .../mymodel.erdterm`

with the path to the ERD term file as a parameter (spiceup is a program stored in kics2home/bin where kics2home is the installation directory of KiCS2; see Section 1.2). You can also provide a path name, i.e., the name of a directory, where the database files should be stored, e.g.,

`spiceup --dbpath DBDIR .../mymodel.erdterm`

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.
10 Technical Problems

One can implement distributed systems with KiCS2 by the use of the library NamedSocket (Section A.2.26) 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.4.2) or WUI (Section A.4.8). 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/cpns/start and kics2home/cpns/stop. If the demon is already running, the command kics2home/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

mh@informatik.uni-kiel.de
References


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 (Section A.5.8) and AbstractCurry (Section A.5.1), stored in the directory “kics2home/lib/meta”, which define datatypes for the representation of Curry programs. AbstractCurry is a more direct representation of a Curry program, whereas FlatCurry is a simplified representation where local function definitions are replaced by global definitions (i.e., lambda lifting has been performed) and pattern matching is translated into explicit case/or expressions. Thus, FlatCurry can be used for more back-end oriented program manipulations (or, for writing new back ends for Curry), whereas AbstractCurry is intended for manipulations of programs that are more oriented towards the source program.

Both modules contain predefined I/O actions to read programs in the AbstractCurry (readCurry) or FlatCurry (readFlatCurry) format. These actions parse the corresponding source program and return a data term representing this program (according to the definitions in the modules AbstractCurry and FlatCurry).

Since all datatypes are explained in detail in these modules, we refer to the online documentation6 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.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],
```

6http://www-ps.informatik.uni-kiel.de/kics2/lib/CDOC/FlatCurry.html and http://www-ps.informatik.uni-kiel.de/kics2/lib/CDOC/AbstractCurry.html
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:

\[ \text{getAllValues} :: a \rightarrow \text{IO} [a] \]

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.

\[ \text{getOneValue} :: a \rightarrow \text{IO} \text{ (Maybe} a) \]

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

\[ \text{getAllSolutions} :: (a \rightarrow \text{Success}) \rightarrow \text{IO} [a] \]

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.

\[ \text{getOneSolution} :: (a \rightarrow \text{Success}) \rightarrow \text{IO} \text{ (Maybe} a) \]

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

\[ \text{getAllFailures} :: a \rightarrow (a \rightarrow \text{Success}) \rightarrow \text{IO} [a] \]

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 → Success) → [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
(isEqualIO 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 a 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 contain no duplicate elements and the order of their elements cannot be observed. In practice, these conditions are not enforced.

Exported functions:

permute :: [a] → [a]
Compute any permutation of a list. For example, [1,2,3,4] may give [1,3,4,2].

subset :: [a] → [a]
Compute any sublist of a list. The sublist contains some of the elements of the list in the same order. For example, [1,2,3,4] may give [1,3], and [1,2,3] gives [1,2], [1,3], [1], [2,3], [2], [3], or [].

splitSet :: [a] → ([a],[a])
Split a list into any two sublists. For example, [1,2,3,4] may give ([1,3,4],[2]).

sizedSubset :: Int → [a] → [a]
Compute any sublist of fixed length of a list. Similar to subset, but the length of the result is fixed.

partition :: [a] → [[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. For example, [1,2,3,4] may give [[4],[2,3],[1]], and [1,2,3] gives [[1,2,3],[[2,3],[1]], [[1,3],[2]], [3],[1,2]], or [[3],[2],[1]].

A.2.5 Library Constraint

Some useful operations for constraint programming.

Exported functions:

(<:) :: a → a → Success
Less-than on ground data terms as a constraint.

(>:) :: a → a → Success
Greater-than on ground data terms as a constraint.

(<=:) :: a → a → Success
Less-or-equal on ground data terms as a constraint.
\((\geq\:):\) :: \(a \rightarrow a \rightarrow \text{Success}\)

Greater-or-equal on ground data terms as a constraint.

\(\text{andC}::[\text{Success}] \rightarrow \text{Success}\)

Evaluates the conjunction of a list of constraints.

\(\text{orC}::[\text{Success}] \rightarrow \text{Success}\)

Evaluates the disjunction of a list of constraints.

\(\text{allC}::(a \rightarrow \text{Success}) \rightarrow [a] \rightarrow \text{Success}\)

Is a given constraint abstraction satisfied by all elements in a list?

\(\text{anyC}::(a \rightarrow \text{Success}) \rightarrow [a] \rightarrow \text{Success}\)

Is there an element in a list satisfying a given constraint?

### A.2.6 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:

\(\text{cpnsStart}::\text{IO} ()\)

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.

\(\text{cpnsShow}::\text{IO} ()\)

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

\(\text{cpnsStop}::\text{IO} ()\)

Terminates the local CPNS demon

\(\text{registerPort}::\text{String} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{IO} ()\)

Registers a symbolic port at the local host.

\(\text{getPortInfo}::\text{String} \rightarrow \text{String} \rightarrow \text{IO} (\text{Int,Int})\)

Gets the information about a symbolic port at some host.

\(\text{unregisterPort}::\text{String} \rightarrow \text{IO} ()\)

Unregisters a symbolic port at the local host.

\(\text{cpnsAlive}::\text{Int} \rightarrow \text{String} \rightarrow \text{IO} \text{Bool}\)

Tests whether the CPNS demon at a host is alive.

\(\text{main}::\text{IO} ()\)

Main function for CPNS demon. Check arguments and execute command.
A.2.7 Library CSV

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

Exported functions:

\[
\text{writeCSVFile} :: \text{String} \rightarrow [[\text{String}]] \rightarrow \text{IO ()}
\]

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

\[
\text{showCSV} :: [[\text{String}]] \rightarrow \text{String}
\]

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

\[
\text{readCSVFile} :: \text{String} \rightarrow \text{IO [[String]]}
\]

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

\[
\text{readCSVFileWithDelims} :: \text{String} \rightarrow \text{String} \rightarrow \text{IO [[String]]}
\]

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

\[
\text{readCSV} :: \text{String} \rightarrow [[\text{String}]]
\]

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

\[
\text{readCSVWithDelims} :: \text{String} \rightarrow \text{String} \rightarrow [[\text{String}]]
\]

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

A.2.8 Library Debug

Library for debugging operations.

Exported functions:

\[
\text{trace} :: \text{String} \rightarrow \text{a} \rightarrow \text{a}
\]

Prints the first argument as a side effect and behaves as identity on the second argument.

\[
\text{traceId} :: \text{String} \rightarrow \text{String}
\]

Prints the first argument as a side effect and returns it afterwards.

\[
\text{traceShow} :: \text{a} \rightarrow \text{b} \rightarrow \text{b}
\]

Prints the first argument using \text{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.2.9 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.

currentDirectory :: IO String

Returns the current working directory.

setCurrentDirectory :: String → IO ()

Sets the current working directory.

defDirectoryContents :: 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.2.10 Library Distribution

This module contains functions to obtain information concerning the current distribution of the Curry implementation. Most of the information is based on the external constants curryCompiler....

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

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 contain-
ing auxiliary files.
lookupFileInLoadPath :: String → IO (Maybe String)

Adds a directory name to a file by looking up the current load path. An error message
is delivered if there is no such file.
findFileInLoadPath :: String → IO String

Adds a directory name to a file by looking up the current load path. An error message
is delivered if there is no such file.
readFirstFileInLoadPath :: String → IO String

Returns the contents of the file found first in the current load path. An error message
is delivered if there is no such file.
getLoadPath :: IO [String]

Returns the current path (list of directory names) that is used for loading modules.
getLoadPathForFile :: String → IO [String]

Returns the current path (list of directory names) that is used for loading modules
w.r.t. a given main module file. The directory prefix of the module file (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 CURRYPATH and the entry "libraries" of
the system’s rc file.
getLoadPathForModule :: String → 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 CURRYPATH and the entry "libraries" of the system's
re file.

lookupModuleSourceInLoadPath :: String → 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.

defaultParams :: FrontendParams

The default parameters of the front end.

crParams :: IO FrontendParams

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

setQuiet :: Bool → FrontendParams → FrontendParams

Set quiet mode of the front end.

setExtended :: Bool → FrontendParams → FrontendParams

Set extended mode of the front end.

setOverlapWarn :: Bool → FrontendParams → FrontendParams

Set overlap warn mode of the front end.

setFullPath :: [String] → FrontendParams → 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 → FrontendParams → FrontendParams

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

setLogfile :: String → FrontendParams → 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.

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.2.11 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.

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.2.12 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 directory prefix of a given (Unix) file name.

\[
\text{splitDirectoryBaseName :: String} \rightarrow (\text{String, String})
\]

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.

\[
\text{stripSuffix :: String} \rightarrow \text{String}
\]

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

\[
\text{fileSuffix :: String} \rightarrow \text{String}
\]

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

\[
\text{splitBaseName :: String} \rightarrow (\text{String, String})
\]

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

\[
\text{splitPath :: String} \rightarrow \text{[String]}
\]

Splits a path string into list of directory names.

\[
\text{lookupFileInPath :: String} \rightarrow \text{[String]} \rightarrow \text{[String]} \rightarrow \text{IO (Maybe String)}
\]

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

\[
\text{getFilePath :: String} \rightarrow \text{[String]} \rightarrow \text{[String]} \rightarrow \text{IO String}
\]

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:**

\[
\text{type FilePath = String}
\]

**Exported functions:**

\[
\text{pathSeparator :: Char}
\]

\[
\text{pathSeparators :: String}
\]

\[
\text{isPathSeparator :: Char} \rightarrow \text{Bool}
\]
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 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.

\[
\text{recip :: Float} \rightarrow \text{Float}
\]
Reciprocal

\[
\text{sqrt :: Float} \rightarrow \text{Float}
\]
Square root.

\[
\text{log :: Float} \rightarrow \text{Float}
\]
Natural logarithm.

\[
\text{logBase :: Float} \rightarrow \text{Float} \rightarrow \text{Float}
\]
Logarithm to arbitrary Base.

\[
\text{exp :: Float} \rightarrow \text{Float}
\]
Natural exponent.

\[
\text{sin :: Float} \rightarrow \text{Float}
\]
Sine.

\[
\text{cos :: Float} \rightarrow \text{Float}
\]
Cosine.

\[
\text{tan :: Float} \rightarrow \text{Float}
\]
Tangent.

\[
\text{asin :: Float} \rightarrow \text{Float}
\]
Arc sine.

\[
\text{acos :: Float} \rightarrow \text{Float}
\]

\[
\text{atan :: Float} \rightarrow \text{Float}
\]
Arc tangent.

\[
\text{sinh :: Float} \rightarrow \text{Float}
\]
Hyperbolic sine.

\[
\text{cosh :: Float} \rightarrow \text{Float}
\]
tanh :: Float → Float
    Hyperbolic tangent.

asinh :: Float → Float
    Hyperbolic Arc sine.

acosh :: Float → Float

atanh :: Float → Float
    Hyperbolic Arc tangent.

A.2.15 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.
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
(c) Sven Panne 2002-2005 The Glasgow Haskell Compiler License
Copyright 2004, The University Court of the University of Glasgow. All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted
provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation and/or other materials
  provided with the distribution.

- Neither name of the University nor the names of its contributors may be
  used to endorse or promote products derived from this software without specific prior written
  permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY
OF GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW
OR THE CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPE-
CIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFT-
WARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Exported types:

data ArgOrder

Exported constructors:

- RequireOrder :: ArgOrder a
• Permute :: ArgOrder a

• ReturnInOrder :: (String → a) → ArgOrder a

data OptDescr

Exported constructors:

• Option :: String → [String] → (ArgDescr a) → String → OptDescr a

data ArgDescr

Exported constructors:

• NoArg :: a → ArgDescr a

• ReqArg :: (String → a) → String → ArgDescr a

• OptArg :: (Maybe String → a) → String → ArgDescr a

Exported functions:

usageInfo :: String → [OptDescr a] → String

getOpt :: ArgOrder a → [OptDescr a] → [String] → ([a],[String],[String])

getOpt' :: ArgOrder a → [OptDescr a] → [String] → ([a],[String],[String],[String])

A.2.17 Library 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:

```haskell
data Global

   The abstract type of a global entity.
```

*Exported constructors:*

```haskell
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 persistently 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.

- `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.2.18 Library GUI

Library for GUI programming in Curry (based on Tcl/Tk). This paper contains a description of the basic ideas behind this library.
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
• \textbf{StreamHandler} :: \textit{Handle} → (\textit{Handle} → \textit{GuiPort} → \textit{IO} [\textit{ReconfigureItem}]) → \textit{ReconfigureItem}

\texttt{StreamHandler hdl handler}

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

• \textbf{RemoveStreamHandler} :: \textit{Handle} → \textit{ReconfigureItem}

\texttt{RemoveStreamHandler hdl}

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

\texttt{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.

\textit{Exported constructors:}

• \textbf{DefaultEvent} :: \textit{Event}

\texttt{DefaultEvent}

– the default event of the widget

• \textbf{MouseButton1} :: \textit{Event}

\texttt{MouseButton1}

– left mouse button pressed

• \textbf{MouseButton2} :: \textit{Event}

\texttt{MouseButton2}

– middle mouse button pressed

• \textbf{MouseButton3} :: \textit{Event}

\texttt{MouseButton3}

– right mouse button pressed

• \textbf{KeyPress} :: \textit{Event}

\texttt{KeyPress}

– any key is pressed

• \textbf{Return} :: \textit{Event}

\texttt{Return}

– return key is pressed
The data type for possible configurations of widget collections (e.g., columns, rows).

Exported constructors:

- `CenterAlign :: ConfCollection`
  - centered alignment

- `LeftAlign :: ConfCollection`
  - left alignment

- `RightAlign :: ConfCollection`
  - right alignment

- `TopAlign :: ConfCollection`
  - top alignment

- `BottomAlign :: ConfCollection`
  - bottom alignment

The data type for specifying items in a menu.

Exported constructors:

- `MButton :: (GuiPort → IO [ReconfigureItem]) → String → MenuItem`
  - a button with an associated command and a label string

- `MSeparator :: MenuItem`
  - a separator between menu entries

- `MMenuButton :: String → [MenuItem] → MenuItem`
  - a menu button

73
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
- COval :: (Int,Int) → (Int,Int) → String → CanvasItem
- CText :: (Int,Int) → String → String → 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 → Style
  Fg
– foreground color, i.e., color of the text font

• Bg :: Color → 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.

\[
\text{appendValue} :: \text{WidgetRef} \rightarrow \text{String} \rightarrow \text{GuiPort} \rightarrow \text{IO ()}
\]

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

\[
\text{appendStyledValue} :: \text{WidgetRef} \rightarrow \text{String} \rightarrow \text{[Style]} \rightarrow \text{GuiPort} \rightarrow \text{IO ()}
\]

Append 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.

\[
\text{addRegionStyle} :: \text{WidgetRef} \rightarrow (\text{Int,Int}) \rightarrow (\text{Int,Int}) \rightarrow \text{Style} \rightarrow \text{GuiPort} \rightarrow \text{IO ()}
\]

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.

\[
\text{removeRegionStyle} :: \text{WidgetRef} \rightarrow (\text{Int,Int}) \rightarrow (\text{Int,Int}) \rightarrow \text{Style} \rightarrow \text{GuiPort} \rightarrow \text{IO ()}
\]

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.

\[
\text{getCursorPosition} :: \text{WidgetRef} \rightarrow \text{GuiPort} \rightarrow \text{IO (Int,Int)}
\]

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

\[
\text{seeText} :: \text{WidgetRef} \rightarrow (\text{Int,Int}) \rightarrow \text{GuiPort} \rightarrow \text{IO ()}
\]

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.

\[
\text{focusInput} :: \text{WidgetRef} \rightarrow \text{GuiPort} \rightarrow \text{IO ()}
\]

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.

\[
\text{addCanvas} :: \text{WidgetRef} \rightarrow \text{[CanvasItem]} \rightarrow \text{GuiPort} \rightarrow \text{IO ()}
\]

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

\[
\text{popupMessage} :: \text{String} \rightarrow \text{IO ()}
\]

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 configura-
tion 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.2.19 Library Integer

A collection of common operations on integer numbers. Most operations make no assumption on the precision of integers. Operation \( \text{bitNot} \) 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 \( \text{pow} a b \) is \( a \) raised to the power of \( b \). Fails if \( b < 0 \). Executes in \( O(\log b) \) steps.

ilog :: Int → Int

The value of \( \text{ilog} n \) is the floor of the logarithm in the base 10 of \( n \). Fails if \( n \leq 0 \). 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 \( \text{isqrt} n \) 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 \( \text{factorial} n \) is the factorial of \( n \). Fails if \( n < 0 \).
binomial :: Int → Int → Int

The value of binomial n m is \( \frac{n(n-1)...(n-m+1)}{m(m-1)...1} \) 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.2.20 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

data SeekMode

    The modes for positioning with hSeek in a file.

*Exported constructors:*

- AbsoluteSeek :: SeekMode
- RelativeSeek :: SeekMode
- SeekFromEnd :: SeekMode

**Exported functions:**

stdin :: Handle

    Standard input stream.

stdout :: Handle

    Standard output stream.

stderr :: Handle

    Standard error stream.

openFile :: String → IOMode → IO Handle

    Opens a file in specified mode and returns a handle to it.
hClose :: Handle → IO ()

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

hFlush :: Handle → IO ()

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

hIsEOF :: Handle → IO Bool

Is handle at end of file?

isEOF :: IO Bool

Is standard input at end of file?

hSeek :: Handle → SeekMode → Int → IO ()

Set the position of a handle to a seekable stream (e.g., a file). If the second argument is AbsoluteSeek, SeekFromEnd, or RelativeSeek, the position is set relative to the beginning of the file, to the end of the file, or to the current position, respectively.

hWaitForInput :: Handle → Int → IO Bool

Waits until input is available on the given handle. If no input is available within t milliseconds, it returns False, otherwise it returns True.

hWaitForInputs :: [Handle] → Int → IO Int

Waits until input is available on some of the given handles. If no input is available within 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).
\textit{hReady} :: \textit{Handle} \rightarrow \textit{IO} \textit{Bool}

Checks whether an input is available on a given handle.

\textit{hGetChar} :: \textit{Handle} \rightarrow \textit{IO} \textit{Char}

Reads a character from an input handle and returns it. Throws an error if the end of file has been reached.

\textit{hGetLine} :: \textit{Handle} \rightarrow \textit{IO} \textit{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 \textit{first} character. If the end of file is reached later in the line, it is treated as a line terminator and the (partial) line is returned.

\textit{hGetContents} :: \textit{Handle} \rightarrow \textit{IO} \textit{String}

Reads the complete contents from an input handle and closes the input handle before returning the contents.

\textit{getContents} :: \textit{IO} \textit{String}

Reads the complete contents from the standard input stream until EOF.

\textit{hPutChar} :: \textit{Handle} \rightarrow \textit{Char} \rightarrow \textit{IO} ()

Puts a character to an output handle.

\textit{hPutStr} :: \textit{Handle} \rightarrow \textit{String} \rightarrow \textit{IO} ()

Puts a string to an output handle.

\textit{hPutStrLn} :: \textit{Handle} \rightarrow \textit{String} \rightarrow \textit{IO} ()

Puts a string with a newline to an output handle.

\textit{hPrint} :: \textit{Handle} \rightarrow \textit{a} \rightarrow \textit{IO} ()

Converts a term into a string and puts it to an output handle.

\textit{hIsReadable} :: \textit{Handle} \rightarrow \textit{IO} \textit{Bool}

Is the handle readable?

\textit{hIsWritable} :: \textit{Handle} \rightarrow \textit{IO} \textit{Bool}

Is the handle writable?

\textit{hIsTerminalDevice} :: \textit{Handle} \rightarrow \textit{IO} \textit{Bool}

Is the handle connected to a terminal?

\textbf{A.2.21 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.

ewIORef :: a → IO (IORef a)

    Creates a new IORef with an initial values.

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.2.22 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 → JExp
  JSIArrayIdx
  – array access to index array variable

• JSOp :: String → JExp → JExp → JExp
  JSOp
  – infix operator expression

• JSFCall :: String → [JExp] → JExp
  JSFCall
  – function call

• JSApply :: JExp → JExp → JExp
  JSApply
  – function call where the function is an expression

• JSLambda :: [Int] → [JSStat] → JExp
  JSLambda
  – (anonymous) function with indexed variables as arguments

data JSStat
  Type of JavaScript statements.

Exported constructors:

• JSAssign :: JExp → JExp → JSStat
  JSAssign
  – assignment

• JSIf :: JExp → [JSStat] → [JSStat] → JSStat
  JSIf
  – conditional

• JSSwitch :: JExp → [JSBranch] → JSStat
  JSSwitch
  – switch statement

• JSPCall :: String → [JExp] → 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.2.23 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 `pathtosqlite3`.

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:

```haskell
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:*
```

89
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 transactions in SQLite is available online.

More information on 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.

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 trans-
action 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 trans-
actions 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 given
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.2.24 Library List

Library with some useful operations on lists.

Exported functions:

\[ \text{elemIndex} :: a \rightarrow [a] \rightarrow \text{Maybe Int} \]

Returns the index \( i \) of the first occurrence of an element in a list as \((\text{Just } i)\), otherwise \text{Nothing} is returned.

\[ \text{elemIndices} :: a \rightarrow [a] \rightarrow [\text{Int}] \]

Returns the list of indices of occurrences of an element in a list.

\[ \text{find} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Maybe } a \]

Returns the first element \( e \) of a list satisfying a predicate as \((\text{Just } e)\), otherwise \text{Nothing} is returned.

\[ \text{findIndex} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Maybe Int} \]

Returns the index \( i \) of the first occurrences of a list element satisfying a predicate as \((\text{Just } i)\), otherwise \text{Nothing} is returned.

\[ \text{findIndices} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [\text{Int}] \]

Returns the list of indices of list elements satisfying a predicate.

\[ \text{nub} :: [a] \rightarrow [a] \]

Removes all duplicates in the argument list.

\[ \text{nubBy} :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \]

Removes all duplicates in the argument list according to an equivalence relation.

\[ \text{delete} :: a \rightarrow [a] \rightarrow [a] \]

Deletes the first occurrence of an element in a list.

\[ \text{deleteBy} :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow a \rightarrow [a] \rightarrow [a] \]

Deletes the first occurrence of an element in a list according to an equivalence relation.

\[ (\setminus) :: [a] \rightarrow [a] \rightarrow [a] \]

Computes the difference of two lists.

\[ \text{union} :: [a] \rightarrow [a] \rightarrow [a] \]

Computes the union of two lists.

\[ \text{intersect} :: [a] \rightarrow [a] \rightarrow [a] \]
Computes the intersection of two lists.

\[ \text{intersperse} :: a \rightarrow [a] \rightarrow [a] \]

Puts a separator element between all elements in a list.

Example: \((\text{intersperse} \ 9 \ [1,2,3,4]) = [1,9,2,9,3,9,4]\)

\[ \text{intercalate} :: [a] \rightarrow [[a]] \rightarrow [a] \]

\[ \text{intercalate} \ xs \ xss \] is equivalent to \((\text{concat} \ (\text{intersperse} \ xs \ xss))\). It inserts the list \(xs\) in between the lists in \(xss\) and concatenates the result.

\[ \text{transpose} :: [[a]] \rightarrow [[a]] \]

Transposes the rows and columns of the argument.

Example: \((\text{transpose} \ [[1,2,3],[4,5,6]]) = [[1,4],[2,5],[3,6]]\)

\[ \text{permutations} :: [a] \rightarrow [[a]] \]

Returns the list of all permutations of the argument.

\[ \text{partition} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a],[a]) \]

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: \((\text{partition} \ (<4)></4)>\)

\[ \text{group} :: [a] \rightarrow [[a]] \]

Splits the list argument into a list of lists of equal adjacent elements.

Example: \((\text{group} \ [1,2,2,3,3,3,4]) = [[1],[2,2],[3,3,3],[4]]\)

\[ \text{groupBy} :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [[a]] \]

Splits the list argument into a list of lists of related adjacent elements.

\[ \text{splitOn} :: [a] \rightarrow [a] \rightarrow [[a]] \]

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.

\[ \text{split} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [[a]] \]

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.

\[ \text{split} \ (==a) \ "aabbaca" \ = \ ["","","bb","c",""] \quad \text{split} \ (==a) \ "" \ = \ ["] \]

\[ \text{inits} :: [a] \rightarrow [[a]] \]

95
Returns all initial segments of a list, starting with the shortest. Example: \texttt{inits [1,2,3] == [[]],[1],[1,2],[1,2,3]}\]

\texttt{tails :: [a] \rightarrow [[a]]}

Returns all final segments of a list, starting with the longest. Example: \texttt{tails [1,2,3] == [[1,2,3],[2,3],[3],[[]]}}

\texttt{replace :: a \rightarrow Int \rightarrow [a] \rightarrow [a]}

Replaces an element in a list.

\texttt{isPrefixOf :: [a] \rightarrow [a] \rightarrow Bool}

Checks whether a list is a prefix of another.

\texttt{isSuffixOf :: [a] \rightarrow [a] \rightarrow Bool}

Checks whether a list is a suffix of another.

\texttt{isInfixOf :: [a] \rightarrow [a] \rightarrow Bool}

Checks whether a list is contained in another.

\texttt{sortBy :: (a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow [a]}

Sorts a list w.r.t. an ordering relation by the insertion method.

\texttt{insertBy :: (a \rightarrow a \rightarrow Bool) \rightarrow a \rightarrow [a] \rightarrow [a]}

Inserts an object into a list according to an ordering relation.

\texttt{last :: [a] \rightarrow a}

Returns the last element of a non-empty list.

\texttt{init :: [a] \rightarrow [a]}

Returns the input list with the last element removed.

\texttt{sum :: [Int] \rightarrow Int}

Returns the sum of a list of integers.

\texttt{product :: [Int] \rightarrow Int}

Returns the product of a list of integers.

\texttt{maximum :: [a] \rightarrow a}

Returns the maximum of a non-empty list.

\texttt{minimum :: [a] \rightarrow a}

Returns the minimum of a non-empty list.
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.2.25 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

Monad 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]

Monad sequence for Maybe.

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

Monad map for Maybe.

mplus :: Maybe a → Maybe a → Maybe a

Combine two Maybes, returning the first Just value, if any.

A.2.26 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:

```haskell
data Socket

Abstract type for named sockets.
```

Exported constructors:

Exported functions:

```haskell
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.2.27  Library Parser

Library with functional logic parser combinators.

Exported types:

```haskell
type Parser a = [a] → [a]

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

Exported functions:

```haskell
(<|>) :: ([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.2.28 Library Pretty

This library provides pretty printing combinators. The interface is that of Daan Leijen’s library (fill, fillBreak and indent are missing) with a linear-time, bounded implementation by Olaf Chitil.

Exported types:

\[
data Doc
\]

The abstract data type Doc represents pretty documents.

Exported constructors:

Exported functions:

\[
empty :: Doc
\]

The empty document is, indeed, empty. Although empty has no content, it does have a height of 1 and behaves exactly like (text "") (and is therefore not a unit of \(<\$\>)).

\[
isEmpty :: Doc \rightarrow Bool
\]

Is the document empty?

\[
text :: String \rightarrow Doc
\]

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.

\[
linesep :: String \rightarrow Doc
\]

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.

\[
line :: Doc
\]

The line document 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 linebreak document advances to the next line and indents to the current nesting level. Document linebreak behaves like empty 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 empty if the resulting output fits the page, otherwise it behaves like line.

softbreak = group linebreak

group :: Doc → Doc

The group combinator 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 hang combinator 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
```

combine :: Doc → Doc → Doc → Doc

The document (combine x l r) encloses document x between documents l and r using (<>).

\[ combine x l r = l <> x <> r \]

(<> :: 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.

(<>$) :: Doc → Doc → Doc

The document (x <$> y) concatenates document x and y with a line in between.

(<>$+) :: Doc → Doc → Doc

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

(<>/) :: Doc → Doc → Doc

The document (x <>/ y) concatenates document x and y with a softline in between. 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.
The document (x \langle//\rangle y) concatenates document x and y with a softbreak in between. This effectively puts x and y either right next to each other or 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 (\langle+\rangle), (\langle$\rangle) and so on.

hsep :: [Doc] → Doc

The document (hsep xs) concatenates all documents xs horizontally with (\langle+\rangle).

vsep :: [Doc] → Doc

The document (vsep xs) concatenates all documents xs vertically with (\langle$\rangle). 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

fillSep :: [Doc] → Doc

The document (fillSep xs) concatenates documents xs horizontally with (\langle+\rangle) as long as its fits the page, than inserts a line and continues doing that for all documents in xs.

fillSep xs = foldr (\langle/>\rangle) empty xs
sep :: [Doc] → 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] → Doc

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

vcat :: [Doc] → 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] → Doc

The document (fillCat xs) concatenates documents xs horizontally with (<>) as long as it fits the page, then inserts a linebreak and continues doing that for all documents in xs.

fillCat xs = foldr (</>) empty xs

cat :: [Doc] → 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 → [Doc] → [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`.)

\[\text{encloseSep :: Doc} \rightarrow \text{Doc} \rightarrow \text{Doc} \rightarrow [\text{Doc}] \rightarrow \text{Doc}\]

The document \((\text{encloseSep} \; l \; r \; \text{sep} \; \text{xs})\) concatenates the documents \(\text{xs}\) separated by \(\text{sep}\) 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 separators are put in front of the elements.

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

\[
\text{list} \; \text{xs} = \text{encloseSep} \; \text{lbracket} \; \text{rbracket} \; \text{comma} \; \text{xs}
\]

\[
\text{test} = \text{text} \; "\text{list}" \; \langle \langle \text{list} \; (\text{map} \; \text{int} \; [10,200,3000]) \rangle\rangle
\]

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

\[
\text{list} \; [10,200,3000]
\]

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

\[
\text{list} \; [10
\quad ,200
\quad ,3000]
\]

\[\text{hEncloseSep :: Doc} \rightarrow \text{Doc} \rightarrow \text{Doc} \rightarrow [\text{Doc}] \rightarrow \text{Doc}\]

The document \((\text{hEncloseSep} \; l \; r \; \text{sep} \; \text{xs})\) concatenates the documents \(\text{xs}\) separated by \(\text{sep}\) and encloses the resulting document by \(l\) and \(r\).

The documents are rendered horizontally.

\[\text{fillEncloseSep :: Doc} \rightarrow \text{Doc} \rightarrow \text{Doc} \rightarrow [\text{Doc}] \rightarrow \text{Doc}\]

The document \((\text{fillEncloseSep} \; l \; r \; \text{sep} \; \text{xs})\) concatenates the documents \(\text{xs}\) separated by \(\text{sep}\) 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 separators are put in front of the elements.

\[\text{fillEncloseSepSpaced :: Doc} \rightarrow \text{Doc} \rightarrow \text{Doc} \rightarrow [\text{Doc}] \rightarrow \text{Doc}\]

The document \((\text{fillEncloseSepSpaced} \; l \; r \; \text{sep} \; \text{xs})\) concatenates the documents \(\text{xs}\) separated by \(\text{sep}\) and encloses the resulting document by \(l\) and \(r\). In addition, after each occurrence of \(\text{sep}\), 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 separators are put in front of the elements.

\[
\text{list :: [Doc]} \rightarrow \text{Doc}
\]
The document (list xs) comma separates 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 separators are put in front of the elements.

\[
\text{listSpaced} :: [\text{Doc}] \rightarrow \text{Doc}
\]

Spaced version of \text{list}

\[
\text{tupled} :: [\text{Doc}] \rightarrow \text{Doc}
\]

The document (tupled xs) comma separates 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 separators are put in front of the elements.

\[
\text{tupledSpaced} :: [\text{Doc}] \rightarrow \text{Doc}
\]

Spaced version of \text{tupled}

\[
\text{semiBraces} :: [\text{Doc}] \rightarrow \text{Doc}
\]

The document (semiBraces xs) separates 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.

\[
\text{semiBracesSpaced} :: [\text{Doc}] \rightarrow \text{Doc}
\]

Spaced version of \text{semiBraces}

\[
\text{enclose} :: \text{Doc} \rightarrow \text{Doc} \rightarrow \text{Doc} \rightarrow \text{Doc}
\]

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

\[
\text{enclose} \ l \ r \ x = \ l \ <> \ x \ <> \ r
\]

\[
\text{squotes} :: \text{Doc} \rightarrow \text{Doc}
\]

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

\[
\text{dquotes} :: \text{Doc} \rightarrow \text{Doc}
\]

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

\[
\text{bquotes} :: \text{Doc} \rightarrow \text{Doc}
\]

Document (bquotes x) encloses document x with ' quotes.

\[
\text{parens} :: \text{Doc} \rightarrow \text{Doc}
\]

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

\[
\text{parensIf} :: \text{Bool} \rightarrow \text{Doc} \rightarrow \text{Doc}
\]

Document (parens 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 shouldn’t 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, "}".

arrow :: Doc
    The document equals contains an arrow sign, "->".

doubleArrow :: Doc
    The document equals contains an double arrow sign, "=>".

doubleColon :: Doc
    The document equals contains a double colon sign, "::".

bar :: Doc
    The document equals contains a vertical bar sign, "|".

pretty :: Int → Doc → String
    (pretty w d) pretty prints document d with a page width of w characters
A.2.29 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 very implementation dependent so that one should interpret
them with care!

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)] → 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 → IO a

Print the time needed to execute a given IO action.

profileTimeNF :: a → IO ()

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

profileSpace :: IO a → IO a

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

profileSpaceNF :: a → 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.2.30 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 \text{[PlTerm]} \rightarrow \text{[PlGoal]} \rightarrow \text{PlClause}
• PlDirective :: \text{[PlGoal]} \rightarrow \text{PlClause}
• PlQuery :: \text{[PlGoal]} \rightarrow \text{PlClause}

data PlGoal

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

Exported constructors:

• PlLit :: String \rightarrow \text{[PlTerm]} \rightarrow \text{PlGoal}
• PlNeg :: \text{[PlGoal]} \rightarrow \text{PlGoal}
• PlCond :: \text{[PlGoal]} \rightarrow \text{[PlGoal]} \rightarrow \text{[PlGoal]} \rightarrow \text{PlGoal}

data PlTerm

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

Exported constructors:

• PlVar :: String \rightarrow \text{PlTerm}
• PlAtom :: String \rightarrow \text{PlTerm}
• PlInt :: \text{Int} \rightarrow \text{PlTerm}
• PlFloat :: \text{Float} \rightarrow \text{PlTerm}
• PlStruct :: String \rightarrow \text{[PlTerm]} \rightarrow \text{PlTerm}

Exported functions:

plList :: \text{[PlTerm]} \rightarrow \text{PlTerm}

A Prolog list of Prolog terms.

showPlProg :: \text{[PlClause]} \rightarrow \text{String}

Shows a Prolog program in standard Prolog syntax.

showPlClause :: \text{PlClause} \rightarrow \text{String}
showPlGoals :: [PlGoal] → String

showPlGoal :: PlGoal → String

showPlTerm :: PlTerm → String

A.2.31 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.2.32 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 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 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 integer is read up to the first non-hexdecimal digit.
A.2.33 Library ReadNumeric

Library with some functions for reading and converting numeric tokens.

Exported functions:

\[\text{readInt} :: \text{String} \rightarrow \text{Maybe (Int, String)}\]

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, \text{Nothing} is returned, otherwise the result is \text{Just (v, s)}, where \(v\) is the value of the integer and \(s\) is the remaining string without the integer token.

\[\text{readNat} :: \text{String} \rightarrow \text{Maybe (Int, String)}\]

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, \text{Nothing} is returned, otherwise the result is \text{Just (v, s)} where \(v\) is the value of the number and \(s\) is the remaining string without the number token.

\[\text{readHex} :: \text{String} \rightarrow \text{Maybe (Int, String)}\]

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, \text{Nothing} is returned, otherwise the result is \text{Just (v, s)} where \(v\) is the value of the number and \(s\) is the remaining string without the number token.

\[\text{readOct} :: \text{String} \rightarrow \text{Maybe (Int, String)}\]

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, \text{Nothing} is returned, otherwise the result is \text{Just (v, s)} where \(v\) is the value of the number and \(s\) is the remaining string without the number token.

A.2.34 Library ReadShowTerm

Library for converting ground terms to strings and vice versa.

Exported functions:

\[\text{showTerm} :: \text{a} \rightarrow \text{String}\]

Transforms a ground(!) term into a string representation in standard prefix notation. Thus, \text{showTerm} suspends until its argument is ground. This function is similar to the prelude function \text{show} but can read the string back with \text{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)]

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!). 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)

readTerm :: 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.2.35 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 this paper.

Exported types:

type Strategy a = SearchTree a → ValueSequence a

A search strategy maps a search tree into some sequence of values. Using the abstract 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

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)

    Return the size (number of Value/Fail/Or nodes) of the search tree

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.

g 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.2.36 Library SetFunctions

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

Intuition: If \( f \) is an \( n \)-ary function, then \((\text{setn } f)\) 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, \((\text{setn } f \ a1 \ldots \ an)\) returns the set of all values of \((f \ b1 \ldots \ bn)\) where \( b1,\ldots, bn \) are values of the arguments \( a1,\ldots, an \) (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 \((\text{setn} \ldots)\). Similarly, logical variables occurring in \( a1,\ldots, an \) are not bound inside this capsule.

The set of values returned by a set function is represented by an abstract type \textbf{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


\textbf{Exported types:}

data \text{Values}

Abstract type representing multisets of values.

\textit{Exported constructors:}

\textbf{Exported functions:}

\texttt{set0 :: a \to \text{Values} a}

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

\texttt{set0With :: (SearchTree a \to \text{ValueSequence} a) \to a \to \text{Values} a}

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

\texttt{set1 :: (a \to b) \to a \to \text{Values} b}

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

\texttt{set1With :: (SearchTree a \to \text{ValueSequence} a) \to (b \to a) \to b \to \text{Values} a}

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

\texttt{set2 :: (a \to b \to c) \to a \to b \to \text{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.
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 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.

\textbf{getArgs :: IO \{String\}}

Returns the list of the program’s command line arguments. The program name is not included.

\textbf{getEnviron :: String → IO String}

Returns the value of an environment variable. The empty string is returned for undefined environment variables.

\textbf{setEnviron :: String → String → IO ()}

Set an environment variable to a value. The new value will be passed to subsequent shell commands (see \texttt{system}) and visible to subsequent calls to \texttt{getEnviron} (but it is not visible in the environment of the process that started the program execution).

\textbf{unsetEnviron :: String → IO ()}

Removes an environment variable that has been set by \texttt{setEnviron}.

\textbf{getHostname :: IO String}

Returns the hostname of the machine running this process.

\textbf{getPID :: IO Int}

Returns the process identifier of the current Curry process.

\textbf{getProgName :: IO String}

Returns the name of the current program, i.e., the name of the main module currently executed.

\textbf{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.

\textbf{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.

\textbf{sleep :: Int → IO ()}

The evaluation of the action (sleep n) puts the Curry process asleep for n seconds.

\textbf{isPosix :: Bool}

Is the underlying operating system a POSIX system (unix, MacOS)?

\textbf{isWindows :: Bool}

Is the underlying operating system a Windows system?
A.2.39 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?

compareTo :: CalendarTime → CalendarTime → Ordering

Compares two dates (don’t use it, just for backward compatibility!).

compareToCalendarTime :: CalendarTime → CalendarTime → Ordering

Compares two calendar times.

compareToClockTime :: ClockTime → ClockTime → Ordering

Compares two clock times.

A.2.40 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.2.41 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 theSearchTree Module, free variables that are not bound in the encapsulated 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 mean-
   ingful 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.

\[
\text{someValueWith} :: (\text{SearchTree a} \rightarrow \text{ValueSequence a}) \rightarrow a \rightarrow a
\]

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., \text{dfsStrategy} or \text{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.2.42 Library ValueSequence

This library defines a data structure for sequence of values. It is used in search trees (module \text{SearchTree}) as well as in set functions (module \text{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


**Exported types:**

\[
\text{data ValueSequence}
\]

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:**

\[
\text{emptyVS} :: \text{ValueSequence} a
\]

An empty sequence of values.

\[
\text{addVS} :: a \rightarrow \text{ValueSequence} a \rightarrow \text{ValueSequence} a
\]

Adds a value to a sequence of values.

\[
\text{failVS} :: \text{Int} \rightarrow \text{ValueSequence} a
\]

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 Data Structures and Algorithms

A.3.1 Library Array

Implementation of Arrays with Braun Trees. Conceptually, Braun trees are always infinite. Consequently, 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 → a
```

The first element of the queue.

```
deqTail :: Queue a → Queue a
```

Removes an element at the front of the queue.
deqLast :: Queue a → a

The last element of the queue.

deqInit :: Queue a → Queue a

Removes an element at the end of the queue.

deqReverse :: Queue a → Queue a

Reverses a double ended queue.

rotate :: Queue a → Queue a

Moves the first element to the end of the queue.

matchHead :: Queue a → 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 → 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] → Queue a

Transforms a list to a double ended queue.

deqToList :: Queue a → [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 constructors:
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

Builds a finite map from given list of tuples (key,element). For multiple occurrences 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 function.

A context is a node together with 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, unlabeled and quasi unlabeled (labeled with ()).

- 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 decomposition 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 decomposition 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 decomposition.

According to the main idea, matchAny . (λ&;) 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 LNodes and LEdges.

buildGr :: [((a,Int),Int,b,[(a,Int)])] → Graph b a

Build a Graph from a list of Contexts.

mkUGraph :: [Int] → [(Int,Int)] → Graph () ()

Build a quasi-unlabeled Graph from the list of Nodes and Edges.

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 LNodes into the Graph.

insEdges :: [(Int,Int,a)] → Graph b a → Graph b a

Insert multiple LEdges into the Graph.

delNodes :: [Int] → Graph a b → Graph a b

Remove multiple Nodes from the Graph.
delEdges :: [(Int,Int)] → Graph a b → Graph a b

Remove multiple Edges 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 Nodes 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 Nodes that have a link from the given Node.

pre :: Graph a b → Int → [Int]

Find all Nodes 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 Nodes that link to the given Node and the label of each link.

out :: Graph a b → Int → [(Int,Int,b)]

Find all outward-bound LEdges for the given Node.

inn :: Graph a b → Int → [(Int,Int,b)]
Find all inward-bound LEEdges for the given Node.

\texttt{outdeg :: Graph a b \rightarrow Int \rightarrow Int}

The outward-bound degree of the Node.

\texttt{indeg :: Graph a b \rightarrow Int \rightarrow Int}

The inward-bound degree of the Node.

\texttt{deg :: Graph a b \rightarrow Int \rightarrow Int}

The degree of the Node.

\texttt{gelem :: Int \rightarrow Graph a b \rightarrow Bool}

True if the Node is present in the Graph.

\texttt{equal :: Graph a b \rightarrow Graph a b \rightarrow Bool}

graph equality

\texttt{node' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow Int

The Node in a Context.

\texttt{lab' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow b

The label in a Context.

\texttt{labNode' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow (Int,b)

The LNode from a Context.

\texttt{neighbors' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow [Int]

All Nodes linked to or from in a Context.

\texttt{suc' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow [Int]

All Nodes linked to in a Context.

\texttt{pre' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow [Int]

All Nodes linked from in a Context.

\texttt{lpre' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow [(Int,a)]

All Nodes linked from in a Context, and the label of the links.

\texttt{lsuc' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow [(Int,a)]

All Nodes linked from in a Context, and the label of the links.

\texttt{out' :: \[(a,Int)\],Int,b,[(a,Int)\]} \rightarrow [(Int,Int,a)]
All outward-directed LEdges in a Context.

\[
\text{inn'} :: ([(a,\text{Int})],\text{Int},\text{b},[(a,\text{Int})]) \rightarrow [(\text{Int},\text{Int},a)]
\]

All inward-directed LEdges in a Context.

\[
\text{outdeg'} :: ([(a,\text{Int})],\text{Int},\text{b},[(a,\text{Int})]) \rightarrow \text{Int}
\]

The outward degree of a Context.

\[
\text{indeg'} :: ([(a,\text{Int})],\text{Int},\text{b},[(a,\text{Int})]) \rightarrow \text{Int}
\]

The inward degree of a Context.

\[
\text{deg'} :: ([(a,\text{Int})],\text{Int},\text{b},[(a,\text{Int})]) \rightarrow \text{Int}
\]

The degree of a Context.

\[
\text{labNodes} :: \text{Graph a b} \rightarrow [(\text{Int},a)]
\]

A list of all LNodes in the Graph.

\[
\text{labEdges} :: \text{Graph a b} \rightarrow [(\text{Int},\text{Int},\text{b})]
\]

A list of all LEdges in the Graph.

\[
\text{nodes} :: \text{Graph a b} \rightarrow [\text{Int}]
\]

List all Nodes in the Graph.

\[
\text{edges} :: \text{Graph a b} \rightarrow [(\text{Int},\text{Int})]
\]

List all Edges in the Graph.

\[
\text{newNodes} :: \text{Int} \rightarrow \text{Graph a b} \rightarrow [\text{Int}]
\]

List N available Nodes, ie Nodes that are not used in the Graph.

\[
\text{ufold} :: (([(a,\text{Int})],\text{Int},\text{b},[(a,\text{Int})]) \rightarrow \text{c} \rightarrow \text{c}) \rightarrow \text{c} \rightarrow \text{Graph b a} \rightarrow \text{c}
\]

Fold a function over the graph.

\[
\text{gmap} :: (([(a,\text{Int})],\text{Int},\text{b},[(a,\text{Int})]) \rightarrow [(\text{c},\text{Int})],\text{Int},\text{d},[(\text{c},\text{Int})])) \rightarrow \text{Graph b a} \rightarrow \text{Graph d c}
\]

Map a function over the graph.

\[
\text{nmap} :: (a \rightarrow b) \rightarrow \text{Graph a c} \rightarrow \text{Graph b c}
\]

Map a function over the Node labels in a graph.

\[
\text{emap} :: (a \rightarrow b) \rightarrow \text{Graph c a} \rightarrow \text{Graph c b}
\]

Map a function over the Edge labels in a graph.

\[
\text{labUEdges} :: [(a,b)] \rightarrow [(a,b,())]
\]

add label () to list of edges (node,node)

\[
\text{labUNodes} :: [a] \rightarrow [(a,())]
\]

add label () to list of nodes

\[
\text{showGraph} :: \text{Graph a b} \rightarrow \text{String}
\]

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.
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.

\[
m_w = <\text{choose-initializer}>; \quad /* \text{must not be zero} */
m_z = <\text{choose-initializer}>; \quad /* \text{must not be zero} */
\]

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.

g 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

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  
e.g. for a set `eqInsert` is `==`, for a multiset it is `(-> False)` for a lookUp-table it is `((==) . fst)` 2) equality for looking up values e.g. 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.

`sort :: (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 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:

\[
\text{type SetRBT } a = \text{RedBlackTree } a
\]

Exported functions:

\[
\text{emptySetRBT :: } (a \rightarrow a \rightarrow \text{Bool}) \rightarrow \text{RedBlackTree } a
\]

Returns an empty set, i.e., an empty red-black tree augmented with an order predicate.

\[
\text{isEmptySetRBT :: RedBlackTree } a \rightarrow \text{Bool}
\]

Test for an empty set.

\[
\text{elemRBT :: } a \rightarrow \text{RedBlackTree } a \rightarrow \text{Bool}
\]

Returns true if an element is contained in a (red-black tree) set.

\[
\text{insertRBT :: } a \rightarrow \text{RedBlackTree } a \rightarrow \text{RedBlackTree } a
\]

Inserts an element into a set if it is not already there.

\[
\text{insertMultiRBT :: } a \rightarrow \text{RedBlackTree } a \rightarrow \text{RedBlackTree } a
\]

Inserts an element into a multiset. Thus, the same element can have several occurrences in the multiset.

\[
\text{deleteRBT :: } a \rightarrow \text{RedBlackTree } a \rightarrow \text{RedBlackTree } a
\]

delete an element from a set. Deletes only a single element from a multi set

\[
\text{setRBT2list :: RedBlackTree } a \rightarrow [a]
\]

Transforms a (red-black tree) set into an ordered list of its elements.

\[
\text{unionRBT :: RedBlackTree } a \rightarrow \text{RedBlackTree } a \rightarrow \text{RedBlackTree } a
\]

Computes the union of two (red-black tree) sets. This is done by inserting all elements of the first set into the second set.

\[
\text{intersectRBT :: RedBlackTree } a \rightarrow \text{RedBlackTree } a \rightarrow \text{RedBlackTree } a
\]

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.

\[
\text{sortRBT :: } (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

Generic sort based on insertion into red-black trees. The first argument is the order for the elements.
A.3.8 Library Sort

A collection of useful functions for sorting and comparing characters, strings, and lists.

Exported functions:

```haskell
quickSort :: (a -> a -> Bool) -> [a] -> [a]
  Quicksort.
mergeSort :: (a -> a -> Bool) -> [a] -> [a]
  Bottom-up mergesort.
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></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></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 distin-
guished and Umlauts are sorted as vocals.
```

A.3.9 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.3.10 Library Traversal

Library to support lightweight generic traversals through tree-structured data. See here\(^8\) 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.

\(^8\)http://www-ps.informatik.uni-kiel.de/~sebf/projects/traversal.html
Exported functions:

\[
\text{noChildren :: } a \rightarrow ([b],[b] \rightarrow a)
\]

Traversal function for constructors without children.

\[
\text{children :: } (a \rightarrow ([b],[b] \rightarrow a)) \rightarrow a \rightarrow [b]
\]

Yields the children of a value.

\[
\text{replaceChildren :: } (a \rightarrow ([b],[b] \rightarrow a)) \rightarrow a \rightarrow [b] \rightarrow a
\]

Replaces the children of a value.

\[
\text{mapChildren :: } (a \rightarrow ([b],[b] \rightarrow a)) \rightarrow (b \rightarrow b) \rightarrow a \rightarrow a
\]

Applies the given function to each child of a value.

\[
\text{family :: } (a \rightarrow ([a],[a] \rightarrow a)) \rightarrow a \rightarrow [a]
\]

Computes a list of the given value, its children, those children, etc.

\[
\text{childFamilies :: } (a \rightarrow ([b],[b] \rightarrow a)) \rightarrow (b \rightarrow ([b],[b] \rightarrow b)) \rightarrow a \rightarrow [b]
\]

Computes a list of family members of the children of a value. The value and its children can have different types.

\[
\text{mapFamily :: } (a \rightarrow ([a],[a] \rightarrow a)) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a
\]

Applies the given function to each member of the family of a value. Proceeds bottom-up.

\[
\text{mapChildFamilies :: } (a \rightarrow ([b],[b] \rightarrow a)) \rightarrow (b \rightarrow ([b],[b] \rightarrow b)) \rightarrow (b \rightarrow b) \rightarrow a \rightarrow a
\]

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.

\[
\text{evalFamily :: } (a \rightarrow ([a],[a] \rightarrow a)) \rightarrow (a \rightarrow \text{Maybe a}) \rightarrow a \rightarrow a
\]

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 \text{Nothing}. Proceeds bottom-up.

\[
\text{evalChildFamilies :: } (a \rightarrow ([b],[b] \rightarrow a)) \rightarrow (b \rightarrow ([b],[b] \rightarrow b)) \rightarrow (b \rightarrow \text{Maybe b}) \rightarrow a \rightarrow a
\]

Applies the given function to each member of the families of the children of a value as long as possible. Similar to \text{evalFamily}.

\[
\text{fold :: } (a \rightarrow ([a],[a] \rightarrow a)) \rightarrow (a \rightarrow [b] \rightarrow b) \rightarrow a \rightarrow b
\]

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.4 Libraries for Web Applications

#### A.4.1 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 initial keys.

The categories are named as all initial characters of the keys of the items.

stringList2ItemList :: [String] → [(String, [HtmlExp])]

Convert a string list into a key-item list. The strings are used as keys and for the simple text layout.
A.4.2 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

\texttt{makecurrycgi -m initialForm -o /home/joe/public\_html/prog.cgi prog}

where \texttt{prog} is the name of the Curry program with the cgi script, \texttt{/home/joe/public\_html/prog.cgi} is the desired location of the compiled cgi script, and \texttt{initialForm} is the Curry expression (of type \texttt{IO HtmlForm}) computing the HTML form (where \texttt{makecurrycgi} is a shell script stored in \texttt{pakcs/home/bin}).

Exported types:

\texttt{type CgiEnv = CgiRef \rightarrow String}

The type for representing cgi environments (i.e., mappings from cgi references to the corresponding values of the input elements).

\texttt{type HtmlHandler = (CgiRef \rightarrow String) \rightarrow IO HtmlForm}

The type of event handlers in HTML forms.

\texttt{data CgiRef}

The (abstract) data type for representing references to input elements in HTML forms.

\texttt{Exported constructors:}

\texttt{data HtmlExp}

The data type for representing HTML expressions.

\texttt{Exported constructors:}

- \texttt{HtmlText :: String \rightarrow HtmlExp}
  \texttt{HtmlText s}
  \quad \texttt{– a text string without any further structure}

- \texttt{HtmlStruct :: String \rightarrow [(String, String)] \rightarrow [HtmlExp] \rightarrow HtmlExp}
  \texttt{HtmlStruct t as hs}
  \quad \texttt{– a structure with a tag, attributes, and HTML expressions inside the structure}

- \texttt{HtmlCRef :: HtmlExp \rightarrow CgiRef \rightarrow HtmlExp}
  \texttt{HtmlCRef h ref}
  \quad \texttt{– 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 (typically, 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.

didOfCgiRef :: 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

emphasize :: [HtmlExp] → HtmlExp

Emphasize

strong :: [HtmlExp] → HtmlExp

Strong (more emphasized) text.

bold :: [HtmlExp] → HtmlExp

Boldface
italic :: [HtmlExp] → HtmlExp

Italic

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 \rightarrow ((CgiRef \rightarrow String) \rightarrow IO HtmlForm) \rightarrow HtmlExp

Submit button with a label string and an event handler

resetbutton :: String \rightarrow HtmlExp

Reset button with a label string

imageButton :: String \rightarrow ((CgiRef \rightarrow String) \rightarrow IO HtmlForm) \rightarrow HtmlExp

Submit button in form of an imag.

textfield :: CgiRef \rightarrow String \rightarrow HtmlExp

Input text field with a reference and an initial contents

password :: CgiRef \rightarrow HtmlExp

Input text field (where the entered text is obscured) with a reference

textarea :: CgiRef \rightarrow (Int,Int) \rightarrow String \rightarrow HtmlExp

Input text area with a reference, height/width, and initial contents

checkbox :: CgiRef \rightarrow String \rightarrow HtmlExp

A checkbox with a reference and a value. The value is returned if checkbox is on, otherwise "" is returned.

checkedbox :: CgiRef \rightarrow String \rightarrow 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 \rightarrow String \rightarrow 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 \rightarrow String \rightarrow 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 \rightarrow String \rightarrow 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.

\[\text{selection} :: \text{CgiRef} \rightarrow \left[(\text{String},\text{String})\right] \rightarrow \text{HtmlExp}\]

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.

\[\text{selectionInitial} :: \text{CgiRef} \rightarrow \left[(\text{String},\text{String})\right] \rightarrow \text{Int} \rightarrow \text{HtmlExp}\]

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.

\[\text{multipleSelection} :: \text{CgiRef} \rightarrow \left[(\text{String},\text{String},\text{Bool})\right] \rightarrow \text{HtmlExp}\]

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 corresponding 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 (\n) characters.

\[\text{hiddenfield} :: \text{String} \rightarrow \text{String} \rightarrow \text{HtmlExp}\]

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.

\[\text{htmlQuote} :: \text{String} \rightarrow \text{String}\]

Quotes special characters (<,>,&," , umlauts) in a string as HTML special characters.

\[\text{htmlIsoUmlauts} :: \text{String} \rightarrow \text{String}\]

Translates umlauts in iso-8859-1 encoding into HTML special characters.

\[\text{addAttr} :: \text{HtmlExp} \rightarrow (\text{String},\text{String}) \rightarrow \text{HtmlExp}\]

Adds an attribute (name/value pair) to an HTML element.

\[\text{addAttrs} :: \text{HtmlExp} \rightarrow \left[(\text{String},\text{String})\right] \rightarrow \text{HtmlExp}\]

Adds a list of attributes (name/value pair) to an HTML element.

\[\text{addClass} :: \text{HtmlExp} \rightarrow \text{String} \rightarrow \text{HtmlExp}\]

Adds a class attribute to an HTML element.

\[\text{showHtmlExps} :: \left[\text{HtmlExp}\right] \rightarrow \text{String}\]

Transforms a list of HTML expressions into string representation.

\[\text{showHtmlExp} :: \text{HtmlExp} \rightarrow \text{String}\]
Transforms a single HTML expression into string representation.

\[ \text{showHtmlPage :: HtmlPage} \rightarrow \text{String} \]

Transforms HTML page into string representation.

\[ \text{getUriParameter :: IO String} \]

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.

\[ \text{urlencoded2string :: String} \rightarrow \text{String} \]

Translates urlencoded string into equivalent ASCII string.

\[ \text{string2urlencoded :: String} \rightarrow \text{String} \]

Translates arbitrary strings into equivalent urlencoded string.

\[ \text{getCookies :: IO [(String,String)\]} \]

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.

\[ \text{coordinates :: (CgiRef} \rightarrow \text{String}) \rightarrow \text{Maybe (Int,Int)} \]

For image buttons: retrieve the coordinates where the user clicked within the image.

\[ \text{runFormServerWithKey :: String} \rightarrow \text{String} \rightarrow \text{IO HtmlForm} \rightarrow \text{IO ()} \]

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.

\[ \text{runFormServerWithKeyAndFormParams :: String} \rightarrow \text{String} \rightarrow \text{[FormParam]} \rightarrow \text{IO HtmlForm} \rightarrow \text{IO ()} \]

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.

\[ \text{showLatexExps :: [HtmlExp]} \rightarrow \text{String} \]

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.4.3 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:

```haskell
data CgiServerMsg
```

The messages to communicate 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:

\texttt{readCgiServerMsg :: Handle \rightarrow IO (Maybe CgiServerMsg)}

Reads a line from a handle and check whether it is a syntactically correct cgi server message.

\texttt{submitForm :: IO ()}

\texttt{runCgiServerCmd :: String \rightarrow CgiServerMsg \rightarrow IO ()}

Executes a specific command for a cgi server.

\texttt{noHandlerPage :: String \rightarrow String \rightarrow String}

\texttt{cgiServerRegistry :: String}

The name of the file to register all cgi servers.

\texttt{registerCgiServer :: String \rightarrow String \rightarrow IO ()}

\texttt{unregisterCgiServer :: String \rightarrow IO ()}

\textbf{A.4.4 Library HtmlParser}

This module contains a very simple parser for HTML documents.

Exported functions:

\texttt{readHtmlFile :: String \rightarrow IO [HtmlExp]}

Reads a file with HTML text and returns the corresponding HTML expressions.

\texttt{parseHtmlString :: String \rightarrow [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.

\textbf{A.4.5 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.4.6 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**
  
  \[\text{Text } s\]
  
  - a simple text in a markdown document

- **Emph :: String → MarkdownElem**
  
  \[\text{Emph } s\]
  
  - an emphasized text in a markdown document

- **Strong :: String → MarkdownElem**
  
  \[\text{Strong } s\]
  
  - a strongly emphasized text in a markdown document

- **Code :: String → MarkdownElem**
  
  \[\text{Code } s\]
  
  - a code string in a markdown document

- **HRef :: String → String → MarkdownElem**
  
  \[\text{HRef } s u\]
  
  - a reference to URL \(u\) with text \(s\) in a markdown document

- **Par :: [MarkdownElem] → MarkdownElem**
  
  \[\text{Par } md\]
  
  - a paragraph in a markdown document

- **CodeBlock :: String → MarkdownElem**
  
  \[\text{CodeBlock } s\]
  
  - a code block in a markdown document

- **UList :: [[MarkdownElem]] → MarkdownElem**
  
  \[\text{UList } mds\]
  
  - an unordered list in a markdown document

- **OList :: [[MarkdownElem]] → MarkdownElem**
  
  \[\text{OList } mds\]
  
  - an ordered list in a markdown document

- **Quote :: [[MarkdownElem]] → MarkdownElem**
  
  \[\text{Quote } md\]
- a quoted paragraph in a markdown document

- HRule :: MarkdownElem
  HRule
  - a horizontal 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.4.7 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.4.8 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:

\[\text{wuiHandler2button} :: \text{String} \rightarrow \text{WuiHandler} \rightarrow \text{HtmlExp}\]

Transform a WUI handler into a submit button with a given label string.

\[\text{withRendering} :: \text{WuiSpec } a \rightarrow ([\text{HtmlExp}] \rightarrow \text{HtmlExp}) \rightarrow \text{WuiSpec } a\]

Puts a new rendering function into a WUI specification.

\[\text{withError} :: \text{WuiSpec } a \rightarrow \text{String} \rightarrow \text{WuiSpec } a\]

Puts a new error message into a WUI specification.

\[\text{withCondition} :: \text{WuiSpec } a \rightarrow (a ightarrow \text{Bool}) \rightarrow \text{WuiSpec } a\]

Puts a new condition into a WUI specification.

\[\text{transformWSpec} :: (a \rightarrow b, b \rightarrow a) \rightarrow \text{WuiSpec } a \rightarrow \text{WuiSpec } b\]

Transforms a WUI specification from one type to another.

\[\text{adaptWSpec} :: (a \rightarrow b) \rightarrow \text{WuiSpec } a \rightarrow \text{WuiSpec } b\]

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 \text{transformWSpec}!

\[\text{wHidden} :: \text{WuiSpec } a\]

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.

\[\text{wConstant} :: (a \rightarrow \text{HtmlExp}) \rightarrow \text{WuiSpec } a\]

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.

\[\text{wInt} :: \text{WuiSpec } \text{Int}\]

A widget for editing integer values.

\[\text{wString} :: \text{WuiSpec } \text{String}\]

A widget for editing string values.

\[\text{wStringLength} :: \text{Int} \rightarrow \text{WuiSpec } \text{String}\]

A widget for editing string values with a size attribute.

\[\text{wRequiredString} :: \text{WuiSpec } \text{String}\]

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.
Cons2 :: (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.

Triple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec (a,b,c)

WUI combinator for triples.

Cons3 :: (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.

4Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec (a,b,c,d)

WUI combinator for tuples of arity 4.

Cons4 :: (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.

5Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec (a,b,c,d,e)

WUI combinator for tuples of arity 5.

Cons5 :: (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 constructor. The further arguments are the WUI specifications for the argument types.

6Tuple :: 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.

Cons6 :: (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 constructor. The further arguments are the WUI specifications for the argument types.

7Tuple :: 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.

\[
w_{12\text{Tuple}} :: WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i \rightarrow WuiSpec j \rightarrow WuiSpec k \rightarrow WuiSpec l \rightarrow WuiSpec (a,b,c,d,e,f,g,h,i,j,k,l)
\]

WUI combinator for tuples of arity 12.

\[
w_{\text{Cons12}} :: (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow k \rightarrow l \rightarrow m) \rightarrow WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i \rightarrow WuiSpec j \rightarrow WuiSpec k \rightarrow WuiSpec l \rightarrow WuiSpec m
\]

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.

\[
w_{\text{JoinTuple}} :: WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec (a,b)
\]

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 \textit{renderTuple}. This combinator is useful to define combinators for large tuples.

\[
w_{\text{List}} :: WuiSpec a \rightarrow WuiSpec [a]
\]

WUI combinator for list structures where the list elements are vertically aligned in a table.

\[
w_{\text{ListWithHeadings}} :: [\text{String}] \rightarrow WuiSpec a \rightarrow WuiSpec [a]
\]

Add headings to a standard WUI for list structures:

\[
w_{\text{HList}} :: WuiSpec a \rightarrow WuiSpec [a]
\]

WUI combinator for list structures where the list elements are horizontally aligned in a table.

\[
w_{\text{Matrix}} :: WuiSpec a \rightarrow WuiSpec [[a]]
\]

WUI for matrices, i.e., list of list of elements visualized as a matrix.

\[
w_{\text{Maybe}} :: WuiSpec \text{Bool} \rightarrow WuiSpec a \rightarrow a \rightarrow WuiSpec (\text{Maybe } a)
\]

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.

\[
w_{\text{CheckMaybe}} :: WuiSpec a \rightarrow \text{HtmlExp} \rightarrow a \rightarrow WuiSpec (\text{Maybe } a)
\]

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.4.9  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:

\[\text{type \ Rendering} = \text{[HtmlExp]} \rightarrow \text{HtmlExp}\]

A rendering is a function that combines the visualization of components of a data
structure into some HTML expression.

\[\text{data \ WuiHandler}\]

A handler for a WUI is an event handler for HTML forms possibly with some specific
JavaScript code attached.

Exported constructors:

\[\text{data \ WuiSpec}\]

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:

\[\text{data \ WTree}\]

A simple tree structure to demonstrate the construction of WUIs for tree types.

Exported constructors:

\[\begin{align*}
\text{• } & \text{WLeaf :: a} \rightarrow \text{WTree a} \\
\text{• } & \text{WNode :: [WTree a]} \rightarrow \text{WTree a}
\end{align*}\]
Exported functions:

\[
\text{wuiHandler2button} :: \text{String} \to \text{WuiHandler} \to \text{HtmlExp}
\]

Transform a WUI handler into a submit button with a given label string.

\[
\text{withRendering} :: \text{WuiSpec a} \to ([\text{HtmlExp}] \to \text{HtmlExp}) \to \text{WuiSpec a}
\]

Puts a new rendering function into a WUI specification.

\[
\text{withError} :: \text{WuiSpec a} \to \text{String} \to \text{WuiSpec a}
\]

Puts a new error message into a WUI specification.

\[
\text{withCondition} :: \text{WuiSpec a} \to (a \to \text{Bool}) \to \text{WuiSpec a}
\]

Puts a new condition into a WUI specification.

\[
\text{withConditionJS} :: \text{WuiSpec a} \to (a \to \text{Bool}) \to \text{WuiSpec a}
\]

Puts a new JavaScript implementation of the condition into a WUI specification.

\[
\text{withConditionJSName} :: \text{WuiSpec a} \to (a \to \text{Bool}, \text{String}) \to \text{WuiSpec a}
\]

Puts a new JavaScript implementation of the condition into a WUI specification.

\[
\text{transformWSpec} :: (a \to b, b \to a) \to \text{WuiSpec a} \to \text{WuiSpec b}
\]

Transforms a WUI specification from one type to another.

\[
\text{adaptWSpec} :: (a \to b) \to \text{WuiSpec a} \to \text{WuiSpec b}
\]

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!

\[
\text{wHidden} :: \text{WuiSpec a}
\]

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.

\[
\text{wConstant} :: (a \to \text{HtmlExp}) \to \text{WuiSpec a}
\]

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.

\[
\text{wInt} :: \text{WuiSpec Int}
\]

A widget for editing integer values.

\[
\text{wString} :: \text{WuiSpec String}
\]

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.

\begin{align*}
\text{wJoinTuple} :: \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} (a,b)
\end{align*}

WUI combinator to combine two tuples into a joint tuple. It is similar to \text{wPair} but
renders both components as a single tuple provided that the components are already
rendered as tuples, i.e., by the rendering function \text{renderTuple}. This combinator is
useful to define combinators for large tuples.

\begin{align*}
\text{wPair} :: \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} (a,b)
\end{align*}

WUI combinator for pairs.

\begin{align*}
\text{wCons2} :: (a \rightarrow b \rightarrow c) \rightarrow \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} c
\end{align*}

WUI combinator for constructors of arity 2. The first argument is the binary con-
structor. The second and third arguments are the WUI specifications for the argument
types.

\begin{align*}
\text{wCons2JS} :: \text{Maybe ([JSExp] \rightarrow JSExp)} \rightarrow (a \rightarrow b \rightarrow c) \rightarrow \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} c
\end{align*}

\begin{align*}
\text{wTriple} :: \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} c \rightarrow \text{WuiSpec} (a,b,c)
\end{align*}

WUI combinator for triples.

\begin{align*}
\text{wCons3} :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} c \rightarrow \text{WuiSpec} d
\end{align*}

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.

\begin{align*}
\text{wCons3JS} :: \text{Maybe ([JSExp] \rightarrow JSExp)} \rightarrow (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} c \rightarrow \text{WuiSpec} d
\end{align*}

\begin{align*}
\text{w4Tuple} :: \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} c \rightarrow \text{WuiSpec} d \rightarrow \text{WuiSpec} (a,b,c,d)
\end{align*}

WUI combinator for tuples of arity 4.

\begin{align*}
\text{wCons4} :: (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e) \rightarrow \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} c \rightarrow \text{WuiSpec} d \rightarrow \text{WuiSpec} e
\end{align*}

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.

\begin{align*}
\text{wCons4JS} :: \text{Maybe ([JSExp] \rightarrow JSExp)} \rightarrow (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e) \rightarrow \text{WuiSpec} a \rightarrow \text{WuiSpec} b \rightarrow \text{WuiSpec} c \rightarrow \text{WuiSpec} d \rightarrow \text{WuiSpec} e
\end{align*}
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 constructor. 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 constructor. 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 constructor. 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.

\[ \text{wCons8 :: (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i) \rightarrow WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i} \]

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.

\[ \text{wCons8JS :: Maybe ([JSExp] \rightarrow JSExp) \rightarrow (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i) \rightarrow WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i} \]

w9Tuple :: WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i \rightarrow WuiSpec (a,b,c,d,e,f,g,h,i)

WUI combinator for tuples of arity 9.

\[ \text{wCons9 :: (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j) \rightarrow WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i \rightarrow WuiSpec j} \]

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.

\[ \text{wCons9JS :: Maybe ([JSExp] \rightarrow JSExp) \rightarrow (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j) \rightarrow WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i \rightarrow WuiSpec j} \]

w10Tuple :: WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i \rightarrow WuiSpec j \rightarrow WuiSpec (a,b,c,d,e,f,g,h,i,j)

WUI combinator for tuples of arity 10.

\[ \text{wCons10 :: (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow k) \rightarrow WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i \rightarrow WuiSpec j \rightarrow WuiSpec k} \]

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.

\[ \text{wCons10JS :: Maybe ([JSExp] \rightarrow JSExp) \rightarrow (a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow k) \rightarrow WuiSpec a \rightarrow WuiSpec b \rightarrow WuiSpec c \rightarrow WuiSpec d \rightarrow WuiSpec e \rightarrow WuiSpec f \rightarrow WuiSpec g \rightarrow WuiSpec h \rightarrow WuiSpec i \rightarrow WuiSpec j \rightarrow WuiSpec k} \]
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.

\[
\text{wMatrix} : \text{WuiSpec } a \rightarrow \text{WuiSpec } [[a]]
\]

WUI for matrices, i.e., list of list of elements visualized as a matrix.

\[
\text{wMaybe} : \text{WuiSpec } \text{Bool} \rightarrow \text{WuiSpec } a \rightarrow a \rightarrow \text{WuiSpec } (\text{Maybe } a)
\]

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.

\[
\text{wCheckMaybe} : \text{WuiSpec } a \rightarrow \text{[HtmlExp]} \rightarrow a \rightarrow \text{WuiSpec } (\text{Maybe } a)
\]

A WUI for Maybe values where a check box is used to select Just. The value WUI is shown after the check box.

\[
\text{wRadioMaybe} : \text{WuiSpec } a \rightarrow \text{[HtmlExp]} \rightarrow \text{[HtmlExp]} \rightarrow a \rightarrow \text{WuiSpec } (\text{Maybe } a)
\]

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.

\[
\text{wEither} : \text{WuiSpec } a \rightarrow \text{WuiSpec } b \rightarrow \text{WuiSpec } (\text{Either } a \ b)
\]

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.

\[
\text{wTree} : \text{WuiSpec } a \rightarrow \text{WuiSpec } (\text{WTree } a)
\]

WUI for tree types. The rendering specifies the rendering of inner nodes. Leaves are shown with their default rendering.

\[
\text{renderTuple} : \text{[HtmlExp]} \rightarrow \text{HtmlExp}
\]

Standard rendering of tuples as a table with a single row. Thus, the elements are horizontally aligned.

\[
\text{renderTaggedTuple} : \text{[String]} \rightarrow \text{[HtmlExp]} \rightarrow \text{HtmlExp}
\]

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.

\[
\text{renderList} : \text{[HtmlExp]} \rightarrow \text{HtmlExp}
\]

Standard rendering of lists as a table with a row for each item: Thus, the elements are vertically aligned.

\[
\text{mainWUI} : \text{WuiSpec } a \rightarrow a \rightarrow (a \rightarrow \text{IO } \text{HtmlForm}) \rightarrow \text{IO } \text{HtmlForm}
\]

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.4.10 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
      
    • XElem :: String → [(String, String)] → [XmlExp] → XmlExp
      

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.4.11 Library XmlConv

Provides type-based combinators to construct XML converters. Arbitrary XML data can be represented as algebraic datatypes and vice versa. See here\(^9\) 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

\(^9\)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
  representation 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 repe-
  titions 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 repeti-
  tions 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 repeti-
    tions 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.

\[ a\text{Bool} :: \text{String} \to \text{String} \to \text{String} \to \text{XmlConv NotRepeatable NoElem} \text{ Bool} \]

Creates an XML converter for boolean attributes. Boolean attributes must not be used in repetitions and do not represent XML elements.

\[ e\text{Int} :: \text{String} \to \text{XmlConv Repeatable Elem} \text{ Int} \]

Creates an XML converter for integer elements. Integer elements may be used in repetitions.

\[ e\text{Float} :: \text{String} \to \text{XmlConv Repeatable Elem} \text{ Float} \]

Creates an XML converter for float elements. Float elements may be used in repetitions.

\[ e\text{Char} :: \text{String} \to \text{XmlConv Repeatable Elem} \text{ Char} \]

Creates an XML converter for character elements. Character elements may be used in repetitions.

\[ e\text{String} :: \text{String} \to \text{XmlConv Repeatable Elem} \text{ String} \]

Creates an XML converter for string elements. String elements may be used in repetitions.

\[ e\text{Bool} :: \text{String} \to \text{String} \to \text{XmlConv Repeatable Elem} \text{ Bool} \]

Creates an XML converter for boolean elements. Boolean elements may be used in repetitions.

\[ e\text{Empty} :: \text{String} \to \text{a} \to \text{XmlConv Repeatable Elem} \text{ a} \]

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.

\[ e\text{Opt} :: \text{String} \to \text{XmlConv} \text{ a b c} \to \text{XmlConv Repeatable Elem} \text{ (Maybe c)} \]

Creates an XML converter that represents an element containing optional XML data. The created element may be used in repetitions.

\[ e\text{Rep} :: \text{String} \to \text{XmlConv Repeatable} \text{ a b} \to \text{XmlConv Repeatable Elem} \text{ [b]} \]

Creates an XML converter that represents an element containing repeated XML data. The created element may be used in repetitions.

\[ \text{seq1} :: (\text{a} \to \text{b}) \to \text{XmlConv c d a} \to \text{XmlConv c NoElem b} \]

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.

\[ \text{repSeq1} :: (\text{a} \to \text{b}) \to \text{XmlConv Repeatable c a} \to \text{XmlConv NotRepeatable NoElem [b]} \]
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.

\[\text{eSeq1 :: String } \rightarrow (a \rightarrow b) \rightarrow \text{XmlConv } c \text{ } d \text{ } a \rightarrow \text{XmlConv } \text{Repeatable } \text{Elem } b\]

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.

\[\text{eRepSeq1 :: String } \rightarrow (a \rightarrow b) \rightarrow \text{XmlConv } \text{Repeatable } c \text{ } a \rightarrow \text{XmlConv } \text{Repeatable } \text{Elem } [b]\]

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

\[\text{seq2 :: (a } \rightarrow b \rightarrow c) \rightarrow \text{XmlConv } d \text{ } e \text{ } a \rightarrow \text{XmlConv } f \text{ } g \text{ } b \rightarrow \text{XmlConv } \text{NotRepeatable } \text{NoElem } c\]

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.

\[\text{repSeq2 :: (a } \rightarrow b \rightarrow c) \rightarrow \text{XmlConv } \text{Repeatable } d \text{ } a \rightarrow \text{XmlConv } \text{Repeatable } e \text{ } b \rightarrow \text{XmlConv } \text{NotRepeatable } \text{NoElem } [c]\]

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.

\[\text{eSeq2 :: String } \rightarrow (a \rightarrow b \rightarrow c) \rightarrow \text{XmlConv } d \text{ } e \text{ } a \rightarrow \text{XmlConv } f \text{ } g \text{ } b \rightarrow \text{XmlConv } \text{Repeatable } \text{Elem } c\]

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.

\[\text{eRepSeq2 :: String } \rightarrow (a \rightarrow b \rightarrow c) \rightarrow \text{XmlConv } \text{Repeatable } d \text{ } a \rightarrow \text{XmlConv } \text{Repeatable } e \text{ } b \rightarrow \text{XmlConv } \text{Repeatable } \text{Elem } [c]\]

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

\[\text{seq3 :: (a } \rightarrow b \rightarrow c \rightarrow d) \rightarrow \text{XmlConv } e \text{ } f \text{ } a \rightarrow \text{XmlConv } g \text{ } h \text{ } b \rightarrow \text{XmlConv } i \text{ } j \text{ } c \rightarrow \text{XmlConv } \text{NotRepeatable } \text{NoElem } d\]

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 Repeatable 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 repetitions.

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 NoElem [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.

\[ \text{eRepSeq6} :: \text{String} \to (a \to b \to c \to d \to e \to f \to g) \to \text{XmlConv Repeatable} \ h \ a \to \text{XmlConv Repeatable} \ i \ b \to \text{XmlConv Repeatable} \ j \ c \to \text{XmlConv Repeatable} \ k \ d \to \text{XmlConv Repeatable} \ l \ e \to \text{XmlConv Repeatable} \ m \ f \to \text{XmlConv Repeatable} \ \text{Elem \ [g]} \]

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

A.5 Libraries for Meta-Programming

A.5.1 Library AbstractCurry

Library to support meta-programming in Curry. This library contains a definition for representing Curry programs in Curry (type "CurryProg") and an I/O action to read Curry programs and transform them into this abstract representation (function "readCurry"). 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:

\begin{itemize}
\item type MName = String
  
  A module name.
\item 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.
\item type CVarIName = (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).
\item type CField a = ((String,String),a)
  
  Labeled record fields.
\item type Arity = Int
  
  Function arity.
\item type CVarIName = (Int,String)
\end{itemize}
Data types for representing object variables. Object variables occurring in expressions are represented by \((\text{Var} \; i)\) where \(i\) is a variable index.

\textbf{data \texttt{CVisibility}}

Data type to specify the visibility of various entities.

\textit{Exported constructors:}

- \texttt{Public} :: \texttt{CVisibility}
- \texttt{Private} :: \texttt{CVisibility}

\textbf{data \texttt{CurryProg}}

Data type for representing a Curry module in the intermediate form. A value of this data type has the form

\[(\text{CProg} \; \text{modname} \; \text{imports} \; \text{typedecls} \; \text{functions} \; \text{opdecls})\]

where \text{modname}: name of this module, \text{imports}: list of modules names that are imported, \text{typedecls}: Type declarations \text{functions}: Function declarations \text{opdecls}: Operator precedence declarations

\textit{Exported constructors:}

- \texttt{CurryProg} :: \texttt{String} \to \texttt{[String]} \to \texttt{[CTypeDecl]} \to \texttt{[CFuncDecl]} \to \texttt{[COpDecl]} \to \texttt{CurryProg}

\textbf{data \texttt{CTypeDecl}}

Data type for representing definitions of algebraic data types and type synonyms.

A data type definition of the form

\[\text{data} \; \text{t} \; \text{x1...xn} = \ldots | \; \text{c} \; \text{t1....tkc} | \ldots\]

is represented by the Curry term

\[(\text{CType} \; \text{t} \; \text{v} \; \text{[i1,...,in]} \; \ldots (\text{CCons} \; \text{c} \; \text{kc} \; \text{v} \; \text{[t1,...,tkc]}) \ldots)\]

where each \(ij\) is the index of the type variable \(x_j\).

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.

\textit{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", "Success", "(())" (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:*

194
• 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:

- \text{CSimpleRhs} :: \text{CExpr} \rightarrow \text{[CLocalDecl]} \rightarrow \text{CRhs}

- \text{CGuardedRhs} :: \text{[(CExpr,CExpr)]} \rightarrow \text{[CLocalDecl]} \rightarrow \text{CRhs}

data \text{CLocalDecl}

Data type for representing local (let/where) declarations

Exported constructors:

- \text{CLocalFunc} :: \text{CFuncDecl} \rightarrow \text{CLocalDecl}

- \text{CLocalPat} :: \text{CPattern} \rightarrow \text{CRhs} \rightarrow \text{CLocalDecl}

- \text{CLocalVars} :: \text{[(Int,String)]} \rightarrow \text{CLocalDecl}

data \text{CPattern}

Data type for representing pattern expressions.

Exported constructors:

- \text{CPVar} :: \text{(Int,String)} \rightarrow \text{CPattern}

- \text{CPLit} :: \text{CLiteral} \rightarrow \text{CPattern}

- \text{CPComb} :: \text{(String,String)} \rightarrow \text{[CPattern]} \rightarrow \text{CPattern}

- \text{CPAs} :: \text{(Int,String)} \rightarrow \text{CPattern} \rightarrow \text{CPattern}

- \text{CPFuncComb} :: \text{(String,String)} \rightarrow \text{[CPattern]} \rightarrow \text{CPattern}

- \text{CPLazy} :: \text{CPattern} \rightarrow \text{CPattern}

- \text{CPRrecord} :: \text{(String,String)} \rightarrow \text{[((String,String),CPattern)]} \rightarrow \text{CPattern}

data \text{CExpr}

Data type for representing Curry expressions.

Exported constructors:

- \text{CVar} :: \text{(Int,String)} \rightarrow \text{CExpr}

- \text{CLit} :: \text{CLiteral} \rightarrow \text{CExpr}

- \text{CSymbol} :: \text{(String,String)} \rightarrow \text{CExpr}

- \text{CApply} :: \text{CExpr} \rightarrow \text{CExpr} \rightarrow \text{CExpr}

- \text{CLambda} :: \text{[CPattern]} \rightarrow \text{CExpr} \rightarrow \text{CExpr}
- 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

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.

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

\texttt{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").

\textbf{A.5.2 Library AbstractCurryGoodies}

This module provides some useful functions to write programs that generate AbstractCurry programs more compact and readable.

\textbf{Exported functions:}

\texttt{\textbackslash{ (>)) :: CTypeExpr → CTypeExpr → CTypeExpr}

A function type.

\texttt{baseType :: (String,String) → CTypeExpr}

A base type.

\texttt{listType :: CTypeExpr → CTypeExpr}

Constructs a list type from an element type.

\texttt{tupleType :: [CTypeExpr] → CTypeExpr}

Constructs a tuple type from list of component types.

\texttt{ioType :: CTypeExpr → CTypeExpr}

Constructs an IO type from a type.

\texttt{maybeType :: CTypeExpr → CTypeExpr}

Constructs a Maybe type from element type.

\texttt{stringType :: CTypeExpr}

The type expression of the String type.

\texttt{intType :: CTypeExpr}

The type expression of the Int type.

\texttt{floatType :: CTypeExpr}

The type expression of the Float type.

\texttt{boolType :: CTypeExpr}

The type expression of the Bool type.
unitType :: CTypeExpr

The type expression of the unit type.

dateType :: CTypeExpr

The type expression of the Time.CalendarTime type.

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

funcName :: CFuncDecl → (String,String)

Returns the name of a given function declaration

funcVis :: CFuncDecl → CVisibility

Returns the visibility of a given function 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.

modsOfType :: CTypeExpr → [String]

Returns all modules used in the given 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.

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.

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.

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.

cChar :: Char → CExpr

Converts a character into an AbstractCurry expression.

string2ac :: String → CExpr

Converts a string into an AbstractCurry representation of this string.

pre :: String → (String,String)

Converts a string into a qualified name of the Prelude.

isPrelude :: String → Bool

Tests whether a module name is the prelude.

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.

cvar :: String → CTypeExpr

Converts a string into a type variable with index 1.
A.5.3  Library AbstractCurryPrinter

A pretty printer for AbstractCurry programs. This library defines a function "showProg" that shows an AbstractCurry program in standard Curry syntax.

Exported functions:

showProg :: CurryProg → String

Shows an AbstractCurry program in standard Curry syntax. The export list contains the public functions and the types with their data constructors (if all data constructors are public), otherwise only the type constructors. The potential comments in function declarations are formatted as documentation comments.

showTypeDecls :: [CTypeDecl] → String

Shows a list of AbstractCurry type declarations in standard Curry syntax.

showTypeDecl :: CTypeDecl → String

Shows an AbstractCurry type declaration in standard Curry syntax.

showTypeExpr :: Bool → CTypeExpr → String

Shows an AbstractCurry type expression in standard Curry syntax. If the first argument is True, the type expression is enclosed in brackets.

showFuncDecl :: CFuncDecl → String

Shows an AbstractCurry function declaration in standard Curry syntax.

showExpr :: CExpr → String

Shows an AbstractCurry expression in standard Curry syntax.

showPattern :: CPattern → String

A.5.4  Library AnnotatedFlatCurry

An annotatable version of FlatCurry. This module 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
- $\text{ATyped} :: a \to (\text{AExpr} \ a) \to \text{TypeExpr} \to \text{AExpr} \ a$

data $\text{ABranchExpr}$

Annotated case branch

*Exported constructors:*

- $\text{ABranch} :: (\text{APattern} \ a) \to (\text{AExpr} \ a) \to \text{ABranchExpr} \ a$

data $\text{APattern}$

Annotated pattern

*Exported constructors:*

- $\text{APattern} :: a \to ((\text{String},\text{String}),a) \to [(\text{Int},a)] \to \text{APattern} \ a$
- $\text{APattern} :: a \to \text{Literal} \to \text{APattern} \ a$

A.5.5 **Library AnnotatedFlatCurryGoodies**

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 $\text{Update} \ a \ b = (b \to b) \to a \to a$

*Exported functions:*

$\text{trProg} :: (\text{String} \to [\text{String}] \to [\text{TypeDecl}] \to [\text{AFuncDecl} \ a] \to [\text{OpDecl}] \to b) \to \text{AProg} \ a \to b$

transform program

$\text{progName} :: \text{AProg} \ a \to \text{String}$

get name from program

$\text{progImports} :: \text{AProg} \ a \to [\text{String}]$

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)

\[ trType :: ((\text{String}, \text{String}) \rightarrow \text{Visibility} \rightarrow \text{[Int]} \rightarrow \text{[ConsDecl]} \rightarrow \text{a}) \rightarrow ((\text{String}, \text{String}) \rightarrow \text{Visibility} \rightarrow \text{[Int]} \rightarrow \text{TypeExpr} \rightarrow \text{a}) \rightarrow \text{TypeDecl} \rightarrow \text{a} \]

transform type declaration

\[ typeName :: \text{TypeDecl} \rightarrow (\text{String}, \text{String}) \]

get name of type declaration

\[ typeVisibility :: \text{TypeDecl} \rightarrow \text{Visibility} \]

get visibility of type declaration

\[ typeParams :: \text{TypeDecl} \rightarrow \text{[Int]} \]

get type parameters of type declaration

\[ typeConsDecls :: \text{TypeDecl} \rightarrow \text{[ConsDecl]} \]

get constructor declarations from type declaration

\[ typeSyn :: \text{TypeDecl} \rightarrow \text{TypeExpr} \]

get synonym of type declaration

\[ isTypeSyn :: \text{TypeDecl} \rightarrow \text{Bool} \]

is type declaration a type synonym?

\[ updType :: ((\text{String}, \text{String}) \rightarrow (\text{String}, \text{String})) \rightarrow (\text{Visibility} \rightarrow \text{Visibility}) \rightarrow (\text{[Int]} \rightarrow \text{[Int]}) \rightarrow ((\text{ConsDecl}) \rightarrow \text{[ConsDecl]}) \rightarrow (\text{TypeExpr} \rightarrow \text{TypeExpr}) \rightarrow \text{TypeDecl} \rightarrow \text{TypeDecl} \]

update type declaration

\[ updTypeName :: ((\text{String}, \text{String}) \rightarrow (\text{String}, \text{String})) \rightarrow \text{TypeDecl} \rightarrow \text{TypeDecl} \]

update name of type declaration

\[ updTypeVisibility :: (\text{Visibility} \rightarrow \text{Visibility}) \rightarrow \text{TypeDecl} \rightarrow \text{TypeDecl} \]

update visibility of type declaration

\[ updTypeParams :: (\text{[Int]} \rightarrow \text{[Int]}) \rightarrow \text{TypeDecl} \rightarrow \text{TypeDecl} \]

update type parameters of type declaration

\[ updTypeConsDecls :: ((\text{ConsDecl}) \rightarrow \text{[ConsDecl]}) \rightarrow \text{TypeDecl} \rightarrow \text{TypeDecl} \]

update constructor declarations of type declaration

\[ updTypeSynonym :: (\text{TypeExpr} \rightarrow \text{TypeExpr}) \rightarrow \text{TypeDecl} \rightarrow \text{TypeDecl} \]
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

$$\text{domain :: TypeExpr } \rightarrow \text{TypeExpr}$$

get domain from functional type

$$\text{range :: TypeExpr } \rightarrow \text{TypeExpr}$$

get range from functional type

$$\text{tConsName :: TypeExpr } \rightarrow \text{(String,String)}$$

get name from constructed type

$$\text{tConsArgs :: TypeExpr } \rightarrow \text{[TypeExpr]}$$

get arguments from constructed type

$$\text{trTypeExpr :: (Int } \rightarrow \text{a)} \rightarrow \text{((String,String) } \rightarrow \text{[a]} \rightarrow \text{a)} \rightarrow \text{(a } \rightarrow \text{a } \rightarrow \text{a)} \rightarrow \text{TypeExpr } \rightarrow \text{a}$$

$$\text{isTVar :: TypeExpr } \rightarrow \text{Bool}$$

is type expression a type variable?

$$\text{isTCons :: TypeExpr } \rightarrow \text{Bool}$$

is type declaration a constructed type?

$$\text{isFuncType :: TypeExpr } \rightarrow \text{Bool}$$

is type declaration a functional type?

$$\text{updTVars :: (Int } \rightarrow \text{TypeExpr)} \rightarrow \text{TypeExpr } \rightarrow \text{TypeExpr}$$

update all type variables

$$\text{updTCons :: ((String,String) } \rightarrow \text{[TypeExpr]} \rightarrow \text{TypeExpr)} \rightarrow \text{TypeExpr } \rightarrow \text{TypeExpr}$$

update all type constructors

$$\text{updFuncTypes :: (TypeExpr } \rightarrow \text{TypeExpr } \rightarrow \text{TypeExpr)} \rightarrow \text{TypeExpr } \rightarrow \text{TypeExpr}$$

update all functional types

$$\text{argTypes :: TypeExpr } \rightarrow \text{[TypeExpr]}$$

get argument types from functional type

$$\text{resultType :: TypeExpr } \rightarrow \text{TypeExpr}$$

get result type from (nested) functional type

$$\text{rnmAllVarsInTypeExpr :: (Int } \rightarrow \text{Int)} \rightarrow \text{TypeExpr } \rightarrow \text{TypeExpr}$$
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]
rename all variables in function

update all qualified names in function

update arguments of function, if not externally defined

update body of function, if not externally defined

transform rule

get rules arguments if it’s not external

get rules body if it’s not external

get rules external declaration

is rule external?

update rule

update rules arguments

update rules body

update rules external declaration

get all variables in rule
get variable names in a functions rule

\[
rnmAllVarsInRule :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{ARule a} \rightarrow \text{ARule a}
\]
rename all variables in rule

\[
updQNamesInRule :: ((\text{String}, \text{String}) \rightarrow (\text{String}, \text{String})) \rightarrow \text{ARule a} \rightarrow \text{ARule a}
\]
update all qualified names in rule

\[
\text{trCombType} :: \text{a} \rightarrow (\text{Int} \rightarrow \text{a}) \rightarrow \text{a} \rightarrow (\text{Int} \rightarrow \text{a}) \rightarrow \text{CombType} \rightarrow \text{a}
\]
transform combination type

\[
isCombTypeFuncCall :: \text{CombType} \rightarrow \text{Bool}
\]
is type of combination FuncCall?

\[
isCombTypeFuncPartCall :: \text{CombType} \rightarrow \text{Bool}
\]
is type of combination FuncPartCall?

\[
isCombTypeConsCall :: \text{CombType} \rightarrow \text{Bool}
\]
is type of combination ConsCall?

\[
isCombTypeConsPartCall :: \text{CombType} \rightarrow \text{Bool}
\]
is type of combination ConsPartCall?

\[
\text{missingArgs} :: \text{CombType} \rightarrow \text{Int}
\]
missing arguments

\[
\text{varNr} :: \text{AExpr a} \rightarrow \text{Int}
\]
get internal number of variable

\[
\text{literal} :: \text{AExpr a} \rightarrow \text{Literal}
\]
get literal if expression is literal expression

\[
\text{combType} :: \text{AExpr a} \rightarrow \text{CombType}
\]
get combination type of a combined expression

\[
\text{combName} :: \text{AExpr a} \rightarrow (\text{String}, \text{String})
\]
get name of a combined expression

\[
\text{combArgs} :: \text{AExpr a} \rightarrow [\text{AExpr a}]
\]
get arguments of a combined expression

\[
\text{missingCombArgs} :: \text{AExpr a} \rightarrow \text{Int}
\]
missing combination arguments
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

\[ \text{patCons :: APattern } a \rightarrow (\text{String}, \text{String}) \]
get name from constructor pattern

\[ \text{patArgs :: APattern } a \rightarrow [(\text{Int}, a)] \]
get arguments from constructor pattern

\[ \text{patLiteral :: APattern } a \rightarrow \text{Literal} \]
get literal from literal pattern

\[ \text{isConsPattern :: APattern } a \rightarrow \text{Bool} \]
is pattern a constructor pattern?

\[ \text{updPattern :: } ((\text{String}, \text{String}), a) \rightarrow ((\text{String}, \text{String}), a) \rightarrow [(\text{Int}, a)] \rightarrow [(\text{Int}, a)]) \rightarrow (\text{Literal} \rightarrow \text{Literal}) \rightarrow \text{APattern } a \rightarrow \text{APattern } a \]
update pattern

\[ \text{updPatCons :: } ((\text{String}, \text{String}) \rightarrow (\text{String}, \text{String})) \rightarrow \text{APattern } a \rightarrow \text{APattern } a \]
update constructors name of pattern

\[ \text{updPatArgs :: } [(\text{Int}, a)] \rightarrow [(\text{Int}, a)]) \rightarrow \text{APattern } a \rightarrow \text{APattern } a \]
update arguments of constructor pattern

\[ \text{updPatLiteral :: } (\text{Literal} \rightarrow \text{Literal}) \rightarrow \text{APattern } a \rightarrow \text{APattern } a \]
update literal of pattern

\[ \text{patExpr :: APattern } a \rightarrow \text{AExpr } a \]
build expression from pattern

\[ \text{annRule :: ARule } a \rightarrow a \]

\[ \text{annExpr :: AExpr } a \rightarrow a \]
Extract the annotation of an annotated expression.

\[ \text{annPattern :: APattern } a \rightarrow a \]
Extract the annotation of an annotated pattern.

\[ \text{unAnnProg :: AProg } a \rightarrow \text{Prog} \]

\[ \text{unAnnFuncDecl :: AFuncDecl } a \rightarrow \text{FuncDecl} \]

unAnnRule :: ARule a → Rule

unAnnExpr :: AExpr a → Expr

unAnnPattern :: APattern a → Pattern

### A.5.6 Library CompactFlatCurry

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.5.7 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 com-
    ments).

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
A.5.8 Library FlatCurry

Library to support meta-programming in Curry. This library contains a definition for representing FlatCurry programs in Curry (type "Prog") and an I/O action to read Curry programs and transform them into this representation (function "readFlatCurry").

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.

```
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

\[ \text{(Type t [i1,...,in] [...(Cons c kc [t1,...,tkc])...])} \]

where each \( i_j \) is the index of the type variable \( x_j \).
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:

- \( \text{Type :: (String,String) \to Visibility \to [Int] \to [ConsDecl] \to TypeDecl} \)
- \( \text{TypeSyn :: (String,String) \to Visibility \to [Int] \to TypeExpr \to TypeDecl} \)

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:

- \( \text{Cons :: (String,String) \to Int \to Visibility \to [TypeExpr] \to ConsDecl} \)

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", "Success", "()" (unit type), "(,...)" (tuple types), "[]" (list type)

Exported constructors:

- \( \text{TVar :: Int \to TypeExpr} \)
- \( \text{FuncType :: TypeExpr \to TypeExpr \to TypeExpr} \)
- \( \text{TCons :: (String,String) \to [TypeExpr] \to TypeExpr} \)

data OpDecl

223
Data type for operator declarations. An operator declaration \texttt{fix p n} in Curry corresponds to the FlatCurry term \((\texttt{Op n fix p})\).

\textit{Exported constructors:}

- \texttt{Op :: (String,String) \rightarrow Fixity \rightarrow Int \rightarrow OpDecl}

\textbf{data Fixity}

Data types for the different choices for the fixity of an operator.

\textit{Exported constructors:}

- \texttt{InfixOp :: Fixity}
- \texttt{InfixlOp :: Fixity}
- \texttt{InfixrOp :: Fixity}

\textbf{data FuncDecl}

Data type for representing function declarations.

A function declaration in FlatCurry is a term of the form

\((\texttt{Func name k type (Rule [i1,...,ik] e)})\)

and represents the function \texttt{name} with definition

\texttt{name :: type}
\texttt{name x1...xk = e}

where each \(i_j\) is the index of the variable \(x_j\).

Note: the variable indices are unique inside each function declaration and are usually numbered from 0.

External functions are represented as

\((\texttt{Func name arity type (External s)})\)

where \(s\) is the external name associated to this function.

Thus, a function declaration consists of the name, arity, type, and rule.

\textit{Exported constructors:}

- \texttt{Func :: (String,String) \rightarrow Int \rightarrow Visibility \rightarrow TypeExpr \rightarrow Rule \rightarrow FuncDecl}

\textbf{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 function calls:

\[(\text{if } e_1 \text{ then } e_2 \text{ else } e_3)\]

is represented as

\[(\text{Comb FuncCall } ("Prelude","if\_then\_else") [e_1,e_2,e_3])\]

Higher-order applications are represented as calls to the (external) function \texttt{apply}. For instance, the rule

\[\text{app } f \; x = f \; x\]

is represented as

\[(\text{Rule } [0,1] (\text{Comb FuncCall } ("Prelude","apply") [\text{Var } 0, \text{Var } 1]))\]

A conditional rule is represented as a call to an external function \texttt{cond} where the first argument is the condition (a constraint). For instance, the rule

\[\text{equal2 } x \mid x =: = 2 = \text{success}\]

is represented as

\[(\text{Rule } [0] (\text{Comb FuncCall } ("Prelude","cond")
\hspace{1cm} [\text{Comb FuncCall } ("Prelude","=:=") [Var 0, \text{Lit (Intc 2)}],
\hspace{1cm} \text{Comb FuncCall } ("Prelude","success") []]))\]

\textit{Exported constructors:}

\begin{itemize}
  \item \textbf{Var} :: Int $\rightarrow$ Expr
    \begin{itemize}
      \item \texttt{Var}
    \end{itemize}
    \hspace{1cm} – variable (represented by unique index)
  \item \textbf{Lit} :: Literal $\rightarrow$ Expr
    \begin{itemize}
      \item \texttt{Lit}
    \end{itemize}
    \hspace{1cm} – literal (Int/Float/Char constant)
\end{itemize}
• **Comb**: \( \text{CombType} \rightarrow (\text{String, String}) \rightarrow [\text{Expr}] \rightarrow \text{Expr} \)

  Comb
  
  - application \((f \ e_1 \ldots \ e_n)\) of function/constructor \(f\) with \(n \leq \text{arity}(f)\)

• **Let**: \([\text{(Int, Expr)}] \rightarrow \text{Expr} \rightarrow \text{Expr} \)

  Let
  
  - introduction of local variables via (recursive) let declarations

• **Free**: \([\text{Int}] \rightarrow \text{Expr} \rightarrow \text{Expr} \)

  Free
  
  - introduction of free local variables

• **Or**: \(\text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr} \)

  Or
  
  - disjunction of two expressions (used to translate rules with overlapping left-hand sides)

• **Case**: \(\text{CaseType} \rightarrow \text{Expr} \rightarrow [\text{BranchExpr}] \rightarrow \text{Expr} \)

  Case
  
  - case distinction (rigid or flex)

• **Typed**: \(\text{Expr} \rightarrow \text{TypeExpr} \rightarrow \text{Expr} \)

  Typed
  
  - typed expression to represent an expression with a type declaration

```haskell
data BranchExpr
```

Data type for representing branches in a case expression.

Branches "(m.c x1...xn) -> e" in case expressions are represented as

\[(\text{Branch} \ (\text{Pattern} \ (m,c) \ [i1,...,in]) \ e)\]

where each \(i_j\) is the index of the pattern variable \(x_j\), or as

\[(\text{Branch} \ (\text{LPattern} \ (\text{Int} c \ i)) \ e)\]

for integers as branch patterns (similarly for other literals like float or character constants).

*Exported constructors:*

• **Branch**: \(\text{Pattern} \rightarrow \text{Expr} \rightarrow \text{BranchExpr} \)
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:*

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.

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").

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.5.9 Library FlatCurryGoodies

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

\[
\text{progFuncs} :: \text{Prog} \rightarrow \text{[FuncDecl]}
\]

get functions from program

\[
\text{progOps} :: \text{Prog} \rightarrow \text{[OpDecl]}
\]

get infix operators from program

\[
\text{updProg} :: (\text{String} \rightarrow \text{String}) \rightarrow \text{([String] \rightarrow \text{String})} \rightarrow \text{([TypeDecl] \rightarrow \text{[TypeDecl]})} \rightarrow \text{([FuncDecl] \rightarrow \text{[FuncDecl]})} \rightarrow \text{([OpDecl] \rightarrow \text{[OpDecl]})} \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

update program

\[
\text{updProgName} :: (\text{String} \rightarrow \text{String}) \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

update name of program

\[
\text{updProgImports} :: \text{([String] \rightarrow \text{[String]})} \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

update imports of program

\[
\text{updProgTypes} :: \text{([TypeDecl] \rightarrow \text{[TypeDecl]})} \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

update type declarations of program

\[
\text{updProgFuncs} :: \text{([FuncDecl] \rightarrow \text{[FuncDecl]})} \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

update functions of program

\[
\text{updProgOps} :: \text{([OpDecl] \rightarrow \text{[OpDecl]})} \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

update infix operators of program

\[
\text{allVarsInProg} :: \text{Prog} \rightarrow \text{[Int]}
\]

get all program variables (also from patterns)

\[
\text{updProgExps} :: (\text{Expr} \rightarrow \text{Expr}) \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

lift transformation on expressions to program

\[
\text{rnmAllVarsInProg} :: \text{(Int} \rightarrow \text{Int}) \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

rename programs variables

\[
\text{updQNamesInProg} :: \text{((String,\text{String})} \rightarrow \text{\text{(String,\text{String})})} \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

update all qualified names in program

\[
\text{rnmProg} :: \text{String} \rightarrow \text{Prog} \rightarrow \text{Prog}
\]

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

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?

\[\text{isGround} :: \text{Expr} \to \text{Bool}\]

is expression fully evaluated?

\[\text{allVars} :: \text{Expr} \to \text{[Int]}\]

got all variables (also pattern variables) in expression

\[\text{rnmAllVars} :: (\text{Int} \to \text{Int}) \to \text{Expr} \to \text{Expr}\]

rename all variables (also in patterns) in expression

\[\text{updQNames} :: ((\text{String},\text{String}) \to (\text{String},\text{String})) \to \text{Expr} \to \text{Expr}\]

update all qualified names in expression

\[\text{trBranch} :: (\text{Pattern} \to \text{Expr} \to \text{a}) \to \text{BranchExpr} \to \text{a}\]

transform branch expression

\[\text{branchPattern} :: \text{BranchExpr} \to \text{Pattern}\]

got pattern from branch expression

\[\text{branchExpr} :: \text{BranchExpr} \to \text{Expr}\]

got expression from branch expression

\[\text{updBranch} :: (\text{Pattern} \to \text{Pattern}) \to (\text{Expr} \to \text{Expr}) \to \text{BranchExpr} \to \text{BranchExpr}\]

update branch expression

\[\text{updBranchPattern} :: (\text{Pattern} \to \text{Pattern}) \to \text{BranchExpr} \to \text{BranchExpr}\]

update pattern of branch expression

\[\text{updBranchExpr} :: (\text{Expr} \to \text{Expr}) \to \text{BranchExpr} \to \text{BranchExpr}\]

update expression of branch expression

\[\text{trPattern} :: ((\text{String},\text{String}) \to \text{[Int]} \to \text{a}) \to (\text{Literal} \to \text{a}) \to \text{Pattern} \to \text{a}\]

transform pattern

\[\text{patCons} :: \text{Pattern} \to (\text{String},\text{String})\]

got name from constructor pattern

\[\text{patArgs} :: \text{Pattern} \to \text{[Int]}\]

got arguments from constructor pattern

\[\text{patLiteral} :: \text{Pattern} \to \text{Literal}\]
get literal from literal pattern

\textbf{isConsPattern} :: \textbf{Pattern} \rightarrow \textbf{Bool}

is pattern a constructor pattern?

\textbf{updPattern} :: ((\textbf{String},\textbf{String}) \rightarrow (\textbf{String},\textbf{String})) \rightarrow ([\textbf{Int}] \rightarrow [\textbf{Int}]) \rightarrow (\textbf{Literal} \rightarrow \textbf{Literal}) \rightarrow \textbf{Pattern} \rightarrow \textbf{Pattern}

update pattern

\textbf{updPatCons} :: ((\textbf{String},\textbf{String}) \rightarrow (\textbf{String},\textbf{String})) \rightarrow \textbf{Pattern} \rightarrow \textbf{Pattern}

update constructors name of pattern

\textbf{updPatArgs} :: ([\textbf{Int}] \rightarrow [\textbf{Int}]) \rightarrow \textbf{Pattern} \rightarrow \textbf{Pattern}

update arguments of constructor pattern

\textbf{updPatLiteral} :: (\textbf{Literal} \rightarrow \textbf{Literal}) \rightarrow \textbf{Pattern} \rightarrow \textbf{Pattern}

update literal of pattern

\textbf{patExpr} :: \textbf{Pattern} \rightarrow \textbf{Expr}

build expression from pattern

\textbf{A.5.10 Library FlatCurryRead}

This library defines operations to read a FlatCurry programs or interfaces together with all its imported modules in the current load path.

\textbf{Exported functions:}

\textbf{readFlatCurryWithImports} :: \textbf{String} \rightarrow \textbf{IO} [\textbf{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).

\textbf{readFlatCurryWithImportsInPath} :: [\textbf{String}] \rightarrow \textbf{String} \rightarrow \textbf{IO} [\textbf{Prog}]

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.

\textbf{readFlatCurryIntWithImports} :: \textbf{String} \rightarrow \textbf{IO} [\textbf{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.

\textbf{readFlatCurryIntWithImportsInPath} :: [\textbf{String}] \rightarrow \textbf{String} \rightarrow \textbf{IO} [\textbf{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.5.11 Library FlatCurryShow

Some tools to show FlatCurry programs. 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.5.12 Library FlatCurryXML

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 from PAKCS or to use the PAKCS back end by other systems.
Exported functions:

\[ \text{flatCurry2XmlFile} :: \text{Prog} \rightarrow \text{String} \rightarrow \text{IO}() \]

Transforms a FlatCurry program term into a corresponding XML file.

\[ \text{flatCurry2Xml} :: \text{Prog} \rightarrow \text{XmlExp} \]

Transforms a FlatCurry program term into a corresponding XML expression.

\[ \text{xmlFile2FlatCurry} :: \text{String} \rightarrow \text{IO Prog} \]

Reads an XML file with a FlatCurry program and returns the FlatCurry program.

\[ \text{xml2FlatCurry} :: \text{XmlExp} \rightarrow \text{Prog} \]

Transforms an XML term into a FlatCurry program.

### A.5.13 Library 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:

\[ \text{data FlexRigidResult} \]

Datatype for representing a flex/rigid status of an expression.

**Exported constructors:**

- UnknownFR :: FlexRigidResult
- ConflictFR :: FlexRigidResult
- KnownFlex :: FlexRigidResult
- KnownRigid :: FlexRigidResult

Exported functions:

\[ \text{getFlexRigid} :: \text{Expr} \rightarrow \text{FlexRigidResult} \]

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.5.14 Library PrettyAbstract

Library for pretty printing AbstractCurry programs. In contrast to the library AbstractCurryPrinter, this library implements a better human-readable pretty printing of AbstractCurry programs.
Exported functions:

preludePrecs :: [((String,String),(CFixity,Int))]

the precedences of the operators in the Prelude module

prettyCProg :: Int → CurryProg → String

(prettyCProg w prog) pretty prints the curry prog prog and fits it to a page width of w characters.

prettyCTypeExpr :: String → CTypeExpr → String

(prettyCTypeExpr mod typeExpr) pretty prints the type expression typeExpr of the module mod and fits it to a page width of 78 characters.

prettyCTypes :: String → [CTypeDecl] → String

(prettyCTypes mod typeDecls) pretty prints the type declarations typeDecls of the module mod and fits it to a page width of 78 characters.

prettyCOps :: [COpDecl] → String

(prettyCOps opDecls) pretty prints the operators opDecls and fits it to a page width of 78 characters.

showCProg :: CurryProg → String

(showCProg prog) pretty prints the curry prog prog and fits it to a page width of 78 characters.

cprogDoc :: CurryProg → Doc

(cprogDoc prog) creates a document of the Curry program prog and fits it to a page width of 78 characters.

cprogDocWithPrecedences :: [((String,String),(CFixity,Int))] → CurryProg → Doc

(cprogDocWithPrecedences precs prog) creates a document of the curry prog prog and fits it to a page width of 78 characters, the precedences precs ensure a correct bracketing of infix operators

precs :: [COpDecl] → [((String,String),(CFixity,Int))]

generates a list of precedences
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:

```
_emphasis_
*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
```
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:
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:

\[
f x y = \text{let } z = (x, y) \\
\hspace{4em} \text{in } (z, z)
\]

The indentation is removed in the output:

\[
f x y = \text{let } z = (x, y) \\
\hspace{4em} \text{in } (z, z)
\]

The visually structure 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  Auxiliary Files

During the translation and execution of a Curry program with KiCS2, various intermediate represen-
tations of the source program are created and stored in different files which are shortly explained in
this section. In general, 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. 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 \( \text{dir} \), the corresponding FlatCurry pro-
gram is stored in the directory “\( \text{dir}/.curry \)”. This is also the case for hierarchical module
names: if the module \( D1.D2.M \) is stored in the directory \( \text{dir} \) (i.e., the module is actu-
ally stored in \( \text{dir}/D1/D2/M.curry \)), then the corresponding FlatCurry program is stored in
“\( \text{dir}/.curry/D1/D2/M.fcy \)”.

**prog.fint**: This file contains the interface of the program in the so-called “FlatCurry” represen-
tation, 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 \( \text{dir} \), the corresponding Haskell program
is stored in the directory “\( \text{dir}/.curry/kics2 \)”. This is also the case for hierarchical module
names: if the module \( D1.D2.M \) is stored in the directory \( \text{dir} \) (i.e., the module is actu-
ally stored in \( \text{dir}/D1/D2/M.curry \)), then the corresponding Haskell program is stored in
“\( \text{dir}/.curry/kics2/D1/D2/Curry_prog.hs \)”.

**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 [7] for more details about the use of this information).
This file is stored in the same directory as **Curry_prog.hs**.
Curry_prog.info: This file contains some information about the top-level functions of module prog that are used by the interactive environment, like determinism behavior or IO status. This file is stored in the same directory as Curry_prog.hs.

prog: This file contains the executable after compiling and saving a program with KiCS2 (see command “:save” in Section 2.2).
D External Operations

Currently, KiCS2 has no general interface to external operations, i.e., operations whose semantics is not defined by program rules in a Curry program but by some code written in another programming language. Thus, if an external operation should be added to the system, this operation must be declared as `external` in the Curry source code and an implementation for this external operation must be provided for the run-time system. An external operation is defined as follows in the Curry source code:

1. Add a type declaration for the external operation somewhere in a module defining this operation (usually, the prelude or some system module).

2. For external operations it is not allowed to define any rule since their semantics is determined by an external implementation. Instead of the defining rules, you have to write

   \[ f \text{ external} \]

   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

\[ \text{sconc :: String } \rightarrow \text{ String } \rightarrow \text{ String} \]

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.\(^{10}\) Thus, we define `sconc` by

\[ \text{sconc s1 s2 = (prim_sconc $## s1) $## s2} \]

\[ \text{prim_sconc :: String } \rightarrow \text{ String } \rightarrow \text{ String} \]

\[ \text{prim_sconc external} \]

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`):

\(^{10}\)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`.

251
The type constructors \texttt{OP\_List} and \texttt{C\_Char} of the prelude \texttt{Curry\_Prelude}\textsuperscript{11} correspond to the Curry type constructors for lists and characters. The Haskell operation \texttt{external\_d\_C\_prim\_sconc} is the external operation to be implemented in Haskell by the programmer. The additional argument of type \texttt{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 \texttt{String\_curry} contains the code of the Curry function \texttt{sconc} described above, the Haskell code implementing the external operations occurring in the module \texttt{String} must be in the file \texttt{External\_String.hs} which is located in the same directory as the file \texttt{String\_curry}. The KiCS2 compiler appends the code contained in \texttt{External\_String.hs} to the generated code stored in the file \texttt{.curry/kics2/Curry\_String.hs}\textsuperscript{12}.

In order to complete our example, we have to write into the file \texttt{External\_String.hs} a definition of the Haskell function \texttt{external\_d\_C\_prim\_sconc}. Thus, we start with the following definitions:

\begin{verbatim}
import qualified Curry\_Prelude as CP

external\_d\_C\_prim\_sconc :: CP.OPL\_ist CP.C\_Char → CP.OPL\_ist CP.C\_Char → Const\_Store → CP.OPL\_ist CP.C\_Char

external\_d\_C\_prim\_sconc x1 x2 x3500 = external\_d\_C\_prim\_sconc x1 x2 x3500
\end{verbatim}

First, we import the standard prelude with the name \texttt{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 \texttt{C\_Char}) into the corresponding Haskell types (like \texttt{Char}). Note that the Curry-related types contain information about non-deterministic or constrained values (see \cite{7,6}) 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, \texttt{fromCurry} converts a Curry type into the corresponding Haskell type, and \texttt{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)

\begin{verbatim}
external\_d\_C\_prim\_sconc s1 s2 _ =
    toCurry ((fromCurry s1 ++ fromCurry s2) :: String)
\end{verbatim}

Here, we use Haskell's concatenation operation \texttt{"++"} to concatenate the string arguments. The type annotation \texttt{":: String"} is necessary because \texttt{"++"} 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 \texttt{fromCurry} and \texttt{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

\textsuperscript{11}Note that all translated Curry modules are imported in the Haskell code fully qualified in order to avoid name conflicts.

\textsuperscript{12}If the file \texttt{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:

```haskell
printString :: String → IO ()
printString s = prim_printString $## s
```

```haskell
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):

```haskell
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

```haskell
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

```haskell
(<&>) :: Int → Int → Int
(<&>) external
```

the generated Haskell module contains the code

```haskell
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**

<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;,</td>
<td>100</td>
</tr>
<tr>
<td>***,</td>
<td>64</td>
</tr>
<tr>
<td>*,</td>
<td>62</td>
</tr>
<tr>
<td>+,</td>
<td>62</td>
</tr>
<tr>
<td>---,</td>
<td>30</td>
</tr>
<tr>
<td>-,</td>
<td>62</td>
</tr>
<tr>
<td>.kics2rc,</td>
<td>16</td>
</tr>
<tr>
<td>/,</td>
<td>62</td>
</tr>
<tr>
<td>//,</td>
<td>131</td>
</tr>
<tr>
<td>:!,</td>
<td>11</td>
</tr>
<tr>
<td>&amp;:</td>
<td>138</td>
</tr>
<tr>
<td>:add</td>
<td>9</td>
</tr>
<tr>
<td>:browse,</td>
<td>11</td>
</tr>
<tr>
<td>:cd,</td>
<td>10</td>
</tr>
<tr>
<td>:edit,</td>
<td>10</td>
</tr>
<tr>
<td>:eval,</td>
<td>10</td>
</tr>
<tr>
<td>:fork,</td>
<td>11</td>
</tr>
<tr>
<td>:help,</td>
<td>9</td>
</tr>
<tr>
<td>:interface,</td>
<td>11</td>
</tr>
<tr>
<td>:load,</td>
<td>9</td>
</tr>
<tr>
<td>:programs,</td>
<td>10</td>
</tr>
<tr>
<td>:quit,</td>
<td>10</td>
</tr>
<tr>
<td>:reload,</td>
<td>9</td>
</tr>
<tr>
<td>:save,</td>
<td>11</td>
</tr>
<tr>
<td>:set,</td>
<td>11</td>
</tr>
<tr>
<td>:set path,</td>
<td>8</td>
</tr>
<tr>
<td>:show,</td>
<td>10</td>
</tr>
<tr>
<td>:source,</td>
<td>11</td>
</tr>
<tr>
<td>:type,</td>
<td>10</td>
</tr>
<tr>
<td>:usedimports,</td>
<td>11</td>
</tr>
<tr>
<td>@,</td>
<td>18</td>
</tr>
<tr>
<td>@author,</td>
<td>30</td>
</tr>
<tr>
<td>@cons,</td>
<td>30</td>
</tr>
<tr>
<td>@param,</td>
<td>30</td>
</tr>
<tr>
<td>@return,</td>
<td>30</td>
</tr>
<tr>
<td>@version,</td>
<td>30</td>
</tr>
<tr>
<td>&amp; &amp; &amp;</td>
<td>64</td>
</tr>
<tr>
<td>KiCS2,</td>
<td>9</td>
</tr>
<tr>
<td>{...},</td>
<td>20</td>
</tr>
<tr>
<td>&lt;**&gt;,</td>
<td>100</td>
</tr>
<tr>
<td>&lt;+&gt;,</td>
<td>103</td>
</tr>
<tr>
<td>&lt;.,</td>
<td>59</td>
</tr>
<tr>
<td>&lt;///&gt;,</td>
<td>104</td>
</tr>
<tr>
<td>&lt;/&gt;.</td>
<td>61, 103</td>
</tr>
<tr>
<td>&lt;:,</td>
<td>47</td>
</tr>
<tr>
<td>&lt;=:,</td>
<td>47</td>
</tr>
<tr>
<td>&lt;$+&gt;,</td>
<td>103</td>
</tr>
<tr>
<td>&lt;$&gt;,</td>
<td>103</td>
</tr>
<tr>
<td>&lt;&gt;,</td>
<td>103</td>
</tr>
<tr>
<td>&gt;:,</td>
<td>47</td>
</tr>
<tr>
<td>&gt;&gt;:,</td>
<td>48</td>
</tr>
<tr>
<td>&gt;&gt;&gt;-,</td>
<td>98</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;,</td>
<td>100</td>
</tr>
<tr>
<td>~&gt;,</td>
<td>199</td>
</tr>
<tr>
<td>,</td>
<td>94</td>
</tr>
<tr>
<td>~,</td>
<td>80</td>
</tr>
<tr>
<td>^</td>
<td>62</td>
</tr>
<tr>
<td>aBool,</td>
<td>188</td>
</tr>
<tr>
<td>ABranchExpr,</td>
<td>205</td>
</tr>
<tr>
<td>abs,</td>
<td>81</td>
</tr>
<tr>
<td>AbstractCurry,</td>
<td>42</td>
</tr>
<tr>
<td>abstractCurryFileName,</td>
<td>198</td>
</tr>
<tr>
<td>aChar,</td>
<td>187</td>
</tr>
<tr>
<td>acos,</td>
<td>63</td>
</tr>
<tr>
<td>acosh,</td>
<td>64</td>
</tr>
<tr>
<td>adapt,</td>
<td>187</td>
</tr>
<tr>
<td>adaptWSpec,</td>
<td>169, 176</td>
</tr>
<tr>
<td>addAttr,</td>
<td>160</td>
</tr>
<tr>
<td>addAttrs,</td>
<td>160</td>
</tr>
<tr>
<td>addCanvas,</td>
<td>78</td>
</tr>
<tr>
<td>addClass,</td>
<td>160</td>
</tr>
<tr>
<td>addCookies,</td>
<td>154</td>
</tr>
<tr>
<td>addCurrySubdir,</td>
<td>54</td>
</tr>
<tr>
<td>addDays,</td>
<td>126</td>
</tr>
<tr>
<td>addExtension,</td>
<td>59</td>
</tr>
<tr>
<td>addFormParam,</td>
<td>155</td>
</tr>
<tr>
<td>addHeadings,</td>
<td>158</td>
</tr>
<tr>
<td>addHours,</td>
<td>126</td>
</tr>
<tr>
<td>addListToFM,</td>
<td>134</td>
</tr>
<tr>
<td>addListToFM_C,</td>
<td>134</td>
</tr>
</tbody>
</table>
addMinutes, 126
addMonths, 126
addPageParam, 155
addRegionStyle, 78
address, 157
addSeconds, 126
addSound, 155
addToFM, 134
addToFM_C, 134
addTrailingPathSeparator, 61
addVS, 130
addYears, 127
AExpr, 204
aFloat, 187
AFuncDecl, 204
aInt, 187
align, 103
allC, 48
allDBInfos, 92
allDBKeyInfos, 92
allDBKeys, 92
AllSolutions, 7
allValuesBFS, 117, 129
allValuesDFS, 117, 129
allValuesDiag, 118
allValuesIDS, 117, 129
allValuesIDSwith, 117, 129
allValuesWith, 117
allVars, 216, 240
allVarsInFunc, 211, 235
allVarsInProg, 206, 230
allVarsInRule, 212, 236
alwaysRequired, 219
anchor, 157
andC, 48
angles, 108
annExpr, 217
annPattern, 217
annRule, 217
answerEncText, 154
answerText, 154
anyC, 48
APattern, 205
appendStyledValue, 78
appendValue, 78
applyAt, 131
applyE, 201
applyF, 201
applyJust, 201
applyMaybe, 201
applyV, 201
AProg, 204
ArgDescr, 66
ArgOrder, 65
args, 15
argTypes, 200, 209, 233
Arity, 192, 204
Array, 131
arrow, 109
ARule, 204
as-pattern, 18
asin, 63
asinh, 64
assert, 50
assertEqual, 44
assertEqualIO, 44
assertIO, 44, 50
Assertion, 44
assertSolutions, 44
assertTrue, 44
assertValues, 44
aString, 187
atan, 63
atanh, 64
attr, 187
backslash, 109
bar, 109
baseName, 57
baseType, 199
best, 7
bfs, 12
bfsStrategy, 117, 129
bindings, 13
binomial, 81
bitAnd, 81
bitNot, 81
bitOr, 81
bitTrunc, 81
bitXor, 81
blink, 157
block, 158
blockstyle, 158
bold, 156
boolType, 199
both, 64
bquotes, 107
braces, 108
brackets, 108
BranchExpr, 227
branchExpr, 216, 240
branchPattern, 216, 240
breakline, 158
buildGr, 139
Button, 79
button, 159

CalendarTime, 125
calendarTimeToString, 126
callFrontend, 56
callFrontendWithParams, 56
CanvasItem, 74
CanvasScroll, 79
caseBranches, 214, 238
caseExpr, 214, 238
caseType, 225
caseType, 214, 238
cat, 105
categorizeByItemKey, 149
catMaybes, 98
cBranch, 202
CCaseType, 197
cChar, 202
CConsDecl, 194
center, 157
CExpr, 196
CField, 192
CFieldDecl, 194
CFixity, 195
cfunc, 201
CFuncDecl, 195
cgiEnv, 150
CgiRef, 150
CgiServerMsg, 163
cgiServerRegistry, 164
char, 108, 186
checkAssertion, 45
checkbox, 159
checkedbox, 159
childFamilies, 148
children, 148
choices, 12
choose, 121
chooseColor, 80
chooseValue, 121
cleancurry, 7
cleanDb, 93
CLiteral, 197
CLocalDecl, 196
ClockTime, 125
clockTimeToInt, 126
closeDbHandles, 93
Cmd, 79
cmp, 14
cmpChar, 146
cmpList, 146
cmpString, 146
cmtfunc, 201
code, 157
col, 76
colon, 109
Color, 75
ColVal, 89
combArgs, 213, 237
combine, 61, 103, 132
combineSimilar, 132
combName, 213, 237
CombType, 225
combType, 213, 237
comma, 109
Command, 79
cmtfunc

text

documentation, 30
compareCalendarTime, 127
compareClockTime, 127
compareDate, 127
compose, 104
computeCompactFlatCurry, 219
concurrency, 7
ConfCollection, 73
ConfigButton, 79
ConfItem, 69
connectToCommand, 85
connectToSocket, 99, 123
connectToSocketRepeat, 99
connectToSocketWait, 99
cons, 132
consArgs, 208, 232
consArity, 208, 232
ConsDecl, 223
consName, 208, 232
constF, 201
consVis, 200
consVisibility, 208, 232
Context, 137
context, 140
Context', 138
cookieForm, 154
CookieParam, 152
coordinates, 161
COpDecl, 194
copyFile, 51
cos, 63
cosh, 63
CPattern, 196
cpnsAlive, 48
cpnsShow, 48
cpnsStart, 48
cpnsStop, 48
cprogDoc, 244
cprogDocWithPrecedences, 244
cpvar, 202
createDirectory, 50
createDirectoryIfMissing, 51
CRhs, 195
CRule, 195
CStatement, 197
ctDay, 125
ctHour, 125
ctMin, 125
ctMonth, 125
ctSec, 125
ctTZ, 125
ctvar, 202
CTVarIName, 192
ctYear, 125
CurryDoc, 30
currydoc, 31
CURRYPATH, 8
CurryProg, 193
curryRuntime, 53
curryRuntimeMajorVersion, 53
curryRuntimeMinorVersion, 53
currySubdir, 54
CurryTest, 35
currytest, 35
cvar, 202
CVarIName, 192
CVisibility, 193
cycle, 97
cyclic structure, 18
database programming, 37
dateType, 200
daysOfMonth, 127
debugTcl, 76
Decomp, 138
defaultEncoding, 154
defaultParams, 55
defaultRequired, 219
deg, 141
deg', 142
delEdge, 139
delEdges, 140
delete, 94, 144
deleteBy, 94
deleteDBEntries, 93
deleteDBEntry, 93
deleteRBT, 145, 147
delFromFM, 134
delListFromFM, 134
delnode, 139
delnodes, 139
deqHead, 132
deqInit, 133
deqLast, 133
deqLength, 132
deqReverse, 133
deqTail, 132
deqToList, 133
dfs, 12
dfsStrategy, 117, 129
diagStrategy, 117
digitToInt, 46
dirName, 57
dlist, 157
Doc, 101
documentation comment, 30
documentation generator, 30
doesDirectoryExist, 50
doesFileExist, 50
domain, 209, 233
doneT, 91
dot, 109
doubleArrow, 109
doubleColon, 109
dquote, 109
dquotes, 107
dropDrive, 60
dropExtension, 59
dropExtensions, 60
dropFileName, 60
dropTrailingPathSeparator, 61
Dynamic, 89
eBool, 188
eChar, 188
Edge, 137
edges, 142
eEmpty, 188
eFloat, 188
eInt, 188
element, 187
elemFM, 135
elemIndex, 94
elemIndices, 94
elemRBT, 145
elemsOf, 184
eIsTsFM, 136
Emacs, 16
eMap, 142
emphasize, 156
empty, 100, 101, 132, 139, 144, 187
emptyDefaultArray, 131
emptyErrorArray, 131
emptyFM, 134
emptySetRBT, 145
emptyTableRBT, 147
emptyVS, 130
encapsulated search, 7
close, 107
closeSep, 106
Encoding, 183
entity relationship diagrams, 37
EntryScroll, 79
eOpt, 188
eqFM, 135
equal, 141
equalFilePath, 61
equals, 109
ERD2Curry, 37
erd2curry, 37
eRep, 188
eRepSeq1, 189
eRepSeq2, 189
eRepSeq3, 190
eRepSeq4, 190
eRepSeq5, 191
eRepSeq6, 192
errorT, 91
eSeq1, 189
eSeq2, 189
eSeq3, 190
eSeq4, 190
eSeq5, 191
eSeq6, 191
garbageCollectorOn, 111
GDecomp, 138
gelem, 141
generateCompactFlatCurryFile, 219
germanLatexDoc, 162
getAbsolutePath, 51
getAllFailures, 43
getAllSolutions, 43
getAllValues, 43
getAllValuesWith, 118, 129
getArgs, 124
getAssoc, 86
clockTime, 125
getContents, 84
getContentsOfUrl, 168
getCookies, 161
getCPUTime, 123
getCurrentDirectory, 50
cursorPosition, 78
getDB, 91
getDBInfo, 93
getDBInfos, 93
directoryContents, 50
getElapsedTime, 123
giveup, 124
giveupForFile, 54
giveupForModule, 55
giveupLocalTime, 126
giveupModificationTime, 50
giveupOneSolution, 43
giveupOneValue, 43
giveupOpenFile, 79
giveupOpenFileWithTypes, 79
giveupOpt, 66
giveupOpt', 66
giveupPID, 124
giveupPortInfo, 48
giveupProcessInfos, 111
giveupProgName, 124
giveupRandomSeed, 143
giveupRcVar, 53
giveupRcVars, 53
giveupSaveFile, 80
giveupSaveFileWithTypes, 80
giveupSearchPath, 59
giveupSearchTree, 116, 128
giveupTemporaryDirectory, 51
giveupUrlParameter, 161
getvalue, 77
ghc, 12, 14
ghci, 13
Global, 67
global, 67
global installation, 6
GlobalSpec, 67
gmap, 142
Graph, 138
group, 95, 102
groupBy, 95
GuiPort, 68
h1, 156
h2, 156
h3, 156
h4, 156
h5, 156
Handle, 82
hang, 102
hasDrive, 60
hasExtension, 59
hasTrailingPathSeparator, 61
hcat, 105
hClose, 83
headedTable, 158
hempty, 156
hEncloseSep, 106
hFlush, 83
hGetChar, 84
hGetContents, 84
hGetLine, 84
hIsEOF, 83
hIsReadable, 84
LEdge, 137
lefts, 57
leqChar, 146
leqCharIgnoreCase, 146
leqLexGerman, 146
leqList, 146
leqString, 146
leqStringIgnoreCase, 146
let, 17
letBinds, 214, 237
letBody, 214, 238
line, 101
linebreak, 101
linesep, 101
list, 106
list2ac, 202
list2CategorizedHtml, 149
ListBoxScroll, 79
listenOn, 99, 123
listenOnFresh, 123
listPattern, 202
listSpaced, 107
listToDefaultArray, 131
listToDeq, 133
listToErrorArray, 131
listToFm, 134
listToMaybe, 98
listType, 199
litem, 157
Literal, 228
literal, 213, 237
LNode, 137
local installation, 6
log, 63
logBase, 63
logfile, 56
lookup, 144
lookupFileInLoadPath, 54
lookupFileInPath, 58
lookupFm, 135
lookupModuleSourceInLoadPath, 55
lookupRBT, 147
lookupWithDefaultFm, 135
lparen, 108
LPath, 138
lpre, 140
lpre', 141
lsuc, 140
lsuc', 141
MailOption, 165
main, 48
mainWUI, 174, 182
makeRelative, 61
makeValid, 62
mapAccumL, 97
mapAccumR, 97
mapChildFamilies, 148
mapChildFamiliesIO, 149
mapChildren, 148
mapChildrenIO, 149
mapFamily, 148
mapFamilyIO, 149
mapFm, 135
mapMaybe, 98
mapMMaybe, 98
mapT, 92
mapT_, 92
mapValues, 122
markdown, 30
MarkdownDoc, 165
MarkdownElem, 165
markdownEscapeChars, 167
markdownText2CompleteHTML, 167
markdownText2CompleteLaTeX, 167
markdownText2HTML, 167
markdownText2LaTeX, 167
markdownText2LaTeXWithFormat, 167
match, 140
matchAny, 139
matchHead, 133
matchLast, 133
matrix, 76
max3, 81
maxFm, 136
maximum, 96
maxList, 81
maxValue, 122
readCompleteFile, 85
readCSV, 49
readCSVFile, 49
readCSVFileWithDelims, 49
readCurry, 42, 198
readCurryWithParseOptions, 198
readFileWithXmlDocs, 185
readFirstFileInLoadPath, 54
readFlatCurry, 42, 228
readFlatCurryFile, 228
readFlatCurryInt, 229
readFlatCurryIntWithImports, 241
readFlatCurryIntWithImportsInPath, 241
readFlatCurryWithImports, 241
readFlatCurryWithImportsInPath, 241
readFlatCurryWithParseOptions, 228
readFM, 136
readGlobal, 67
readHex, 113, 114
readHtmlFile, 164
readInt, 113, 114
readIORef, 86
readNat, 113, 114
readOct, 114
readPropertyFile, 113
readQTerm, 115
readQTermFile, 115
readQTermListFile, 116
readScan, 221
readsQTerm, 115
readsTerm, 115
readsUnqualifiedTerm, 115
readTerm, 115
readUnqualifiedTerm, 115
readUnsafeXmlFile, 185
readUntypedCurry, 198
readUntypedCurryWithParseOptions, 198
readXmlFile, 185
recip, 63
ReconfigureItem, 71
record syntax, 20
RedBlackTree, 144
redirect, 155
registerCgiServer, 164
registerPort, 48
removeDirectory, 51
removeEscapes, 167
removeFile, 51
removeRegionStyle, 78
renameDirectory, 51
renameFile, 51
Rendering, 168, 175
renderList, 174, 182
renderTaggedTuple, 174, 182
renderTuple, 174, 182
rep, 187
replace, 96
replaceBaseName, 60
replaceChildren, 148
replaceChildrenIO, 149
replaceDirectory, 61
replaceExtension, 59
replaceFileName, 60
repSeq1, 188
repSeq2, 189
repSeq3, 190
repSeq4, 190
repSeq5, 191
repSeq6, 191
RequiredSpec, 219
requires, 219
resetbutton, 159
resultType, 201, 209, 233
returnT, 91
rights, 57
rnmAllVars, 216, 240
rnmAllVarsInFunc, 212, 235
rnmAllVarsInProg, 206, 230
rnmAllVarsInRule, 213, 236
rnmAllVarsInTypeExpr, 209, 233
rnmProg, 206, 230
rotate, 133
round, 62
row, 76
rparen, 108
rts, 15
Rule, 224
splitModuleFileName, 53
splitModuleIdentifiers, 53
splitOn, 95
splitPath, 58, 61
splitSearchPath, 59
splitSet, 47
sqrt, 63
sqquote, 109
squotes, 107
standardForm, 154
standardPage, 155
star, 100
stderr, 82
stdin, 82
stdout, 82
Strategy, 116, 128
string, 108, 186
string2ac, 202
string2urlencoded, 161
stringList2ItemList, 149
stringPattern, 202
stringType, 199
stripCurrySuffix, 54
stripSuffix, 58
strong, 156
Style, 74
style, 158
styleSheet, 158
submitButton, 164
subset, 47
suc, 140
suc', 141
suffixSeparatorChar, 57
sum, 96
supply, 12
system, 124
table, 158
TableRBT, 147
tableRBT2list, 147
tabulator stops, 7
tagOf, 184
tails, 96
takeBaseName, 60
takeDirectory, 61
takeDrive, 60
takeExtension, 59
takeExtensions, 60
takeFileName, 60
tan, 63
tanh, 64
tConsArgs, 209, 233
tConsName, 209, 233
teltype, 157
terminal, 100
TError, 90
TErrorKind, 90
testing programs, 35
testScan, 221
text, 101
textarea, 159
TextEditScroll, 79
textfield, 159
textOf, 184
textOfXml, 184
textstyle, 158
time, 13
toCalendarTime, 126
toClockTime, 126
toDayString, 126
Token, 220
Tokens, 220
toLower, 46
toTimeString, 126
toupper, 46
toUTCTime, 126
toVar, 202
trace, 13, 49, 127
traceId, 49
traceIO, 50
traceShow, 49
traceShowId, 50
Transaction, 89
transformQ, 90
transformWSpec, 169, 176
transpose, 95
Traversable, 147
trBranch, 216, 240
updOp, 210, 234
updOpFixity, 210, 234
updOpName, 210, 234
updOpPrecedence, 210, 234
updOrs, 215, 239
updPatArgs, 217, 241
updPatCons, 217, 241
updPatLiteral, 217, 241
updPattern, 217, 241
updProg, 206, 230
updProgExps, 206, 230
updProgFuncs, 206, 230
updProgImports, 206, 230
updProgName, 206, 230
updProgOps, 206, 230
updProgTypes, 206, 230
updQNames, 216, 240
updQNamesInConsDecl, 208, 232
updQNamesInFunc, 212, 236
updQNamesInProg, 206, 230
updQNamesInRule, 213, 237
updQNamesInType, 208, 232
updQNamesInTypeExpr, 210, 234
updRule, 212, 236
updRuleArgs, 212, 236
updRuleBody, 212, 236
updRuleExtDecl, 212, 236
updTCons, 209, 233
updTVars, 209, 233
updType, 207, 231
updTypeConsDecls, 207, 231
updTypeds, 215, 239
updTypeName, 207, 231
updTypeParams, 207, 231
updTypeSynonym, 207, 231
updTypeVisibility, 207, 231
updVars, 215, 239
urlencoded2string, 161
usageInfo, 66

v, 12
validDate, 127
valueOf, 121
Values, 119
values2list, 122
ValueSequence, 130
variables
    singleton, 7
varId, 128
VarIndex, 222
varNr, 213, 237
vcat, 105
verbatim, 157
verbosity, 12
version, 198
Visibility, 222
vsep, 104
vsToList, 131
w10Tuple, 172, 180
w11Tuple, 172, 181
w12Tuple, 173, 181
w4Tuple, 171, 178
w5Tuple, 171, 179
w6Tuple, 171, 179
w7Tuple, 171, 179
w8Tuple, 172, 179
w9Tuple, 172, 180
waitForSocketAccept, 99, 123
wCheckBool, 170, 177
wCheckMaybe, 173, 182
wCons10, 172, 180
wCons10JS, 180
wCons11, 172, 181
wCons11JS, 181
wCons12, 173, 181
wCons12JS, 181
wCons2, 171, 178
wCons2JS, 178
wCons3, 171, 178
wCons3JS, 178
wCons4, 171, 178
wCons4JS, 178
wCons5, 171, 179
wCons5JS, 179
wCons6, 171, 179
wCons6JS, 179
wCons7, 172, 179
wCons7JS, 179
wCons8, 172, 180
wCons8JS, 180
wCons9, 172, 180
wCons9JS, 180
wConstant, 169, 176
wEither, 174, 182
where, 17
wHidden, 169, 176
wHList, 173, 181
Widget, 68
WidgetRef, 74
wInt, 169, 176
withCondition, 169, 176
withConditionJS, 176
withConditionJSName, 176
withError, 169, 176
withRendering, 169, 176
wJoinTuple, 173, 178
wList, 173, 181
wListWithHeadings, 173, 181
wMatrix, 173, 182
wMaybe, 173, 182
wMultiCheckSelect, 170, 177
wPair, 170, 178
wRadioBool, 170, 177
wRadioMaybe, 174, 182
wRadioSelect, 170, 177
wRequiredString, 169, 177
wRequiredStringSize, 170, 177
writeAbstractCurryFile, 199
writeAssertResult, 45
writeCSVFile, 49
writeFCY, 229
writeGlobal, 67
writeIORef, 86
writeQTermFile, 116
writeQTermListFile, 116
writeXmlFile, 184
writeXmlFileWithParams, 184
wSelect, 170, 177
wSelectBool, 170, 177
wSelectInt, 170, 177
wString, 169, 176
wStringLength, 169, 177
wTextarea, 170, 177
wTree, 168, 175
wTree, 174, 182
wTriple, 171, 178
wui2html, 174, 183
WuiHandler, 168, 175
wuiHandler2button, 169, 176
wuiInForm, 174, 183
WuiSpec, 168, 175
wuiWithErrorForm, 174, 183
XAttrConv, 185
XElemConv, 185
xml, 184
xml2FlatCurry, 243
XmlDocParams, 183
XmlExp, 183
xmlFile2FlatCurry, 243
xmlRead, 186
XmlReads, 185
xmlReads, 185
xmlShow, 186
xmlShows, 185
xmlShows, 186
XOptConv, 186
XPrimConv, 185
XRepConv, 186
xtxt, 184