KiCS2

The Kiel Curry System (Version 2)

User Manual

Version 0.3.0 of 2013-07-25

Michael Hanus[1] [editor]

Additional Contributors:

Bernd Braßel[2]
Björn Peemöller[3]
Fabian Reck[4]

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

Contents

Preface 4

1 Overview of KiCS2 5
1.1 Installation . . . 5
1.2 General Use . . . 5
1.3 Restrictions . . . 5
1.4 Modules in KiCS2 . . . 7

2 Using the Interactive Environment of KiCS2 8
2.1 Invoking KiCS2 . . . 8
2.2 Command of KiCS2 . . . 8
2.3 Options of KiCS2 . . . 10
2.4 Source-File Options . . . 13
2.5 Command Line Editing . . . 14
2.6 Customization . . . 14
2.7 Emacs Interface . . . 14

3 Extensions 15
3.1 Recursive Variable Bindings . . . 15
3.2 Functional Patterns . . . 15
3.3 Order of Pattern Matching . . . 16
3.4 Records . . . 17
3.4.1 Record Type Declaration . . . 17
3.4.2 Record Construction . . . 18
3.4.3 Field Selection . . . 18
3.4.4 Field Update . . . 18
3.4.5 Records in Pattern Matching . . . 19
3.4.6 Export of Records . . . 19
3.4.7 Restrictions in the Usage of Records . . . 20

4 Recognized Syntax of Curry 21
4.1 Notational Conventions . . . 21
4.2 Lexicon . . . 21
4.2.1 Case Mode . . . 21
4.2.2 Identifiers and Keywords . . . 21
4.2.3 Comments . . . 22
4.2.4 Numeric and Character Literals . . . 22
4.3 Layout . . . 23
4.4 Context Free Grammar . . . 23

5 CurryDoc: A Documentation Generator for Curry Programs 27

6 CurryBrowser: A Tool for Analyzing and Browsing Curry Programs 30

7 CurryTest: A Tool for Testing Curry Programs **32**

8 ERD2Curry: A Tool to Generate Programs from ER Specifications **34**

9 Technical Problems **35**

Bibliography **36**

A Libraries of the KiCS2 Distribution **38**

A.1 AbstractCurry and FlatCurry: Meta-Programming in Curry . . . 38
A.2 General Libraries . . . 39
A.2.1 Library AllSolutions . . . 39
A.2.2 Library Assertion . . . 40
A.2.3 Library Char . . . 41
A.2.4 Library Combinatorial . . . 42
A.2.5 Library Constraint . . . 43
A.2.6 Library CSV . . . 44
A.2.7 Library Directory . . . 44
A.2.8 Library FileGoodies . . . 45
A.2.9 Library Float . . . 47
A.2.10 Library Global . . . 48
A.2.11 Library GUI . . . 49
A.2.12 Library Integer . . . 61
A.2.13 Library IO . . . 62
A.2.14 Library IOExts . . . 65
A.2.15 Library JavaScript . . . 67
A.2.16 Library KeyDatabaseSQLite . . . 70
A.2.17 Library List . . . 75
A.2.18 Library Maybe . . . 78
A.2.19 Library NamedSocket . . . 79
A.2.20 Library Parser . . . 80
A.2.21 Library Pretty . . . 81
A.2.22 Library Profile . . . 90
A.2.23 Library PropertyFile . . . 91
A.2.24 Library Read . . . 92
A.2.25 Library ReadNumeric . . . 92
A.2.26 Library ReadShowTerm . . . 93
A.2.27 Library SetFunctions . . . 95
A.2.28 Library SearchTree . . . 98
A.2.29 Library Socket . . . 100
A.2.30 Library System . . . 101
A.2.31 Library Time . . . 102
A.2.32 Library Unsafe . . . 105
A.3 Data Structures and Algorithms . . . 105
A.3.1 Library Array . . . 105

A.3.2 Library Dequeue . . . 106
A.3.3 Library FiniteMap . . . 108
A.3.4 Library GraphInductive . . . 111
A.3.5 Library Random . . . 117
A.3.6 Library RedBlackTree . . . 118
A.3.7 Library SetRBT . . . 119
A.3.8 Library Sort . . . 120
A.3.9 Library TableRBT . . . 121
A.3.10 Library Traversal . . . 122
A.4 Libraries for Web Applications . . . 123
A.4.1 Library CategorizedHtmlList . . . 123
A.4.2 Library HTML . . . 124
A.4.3 Library HtmlParser . . . 137
A.4.4 Library Mail . . . 137
A.4.5 Library Markdown . . . 138
A.4.6 Library WUI . . . 140
A.4.7 Library URL . . . 146
A.4.8 Library XML . . . 146
A.4.9 Library XmlConv . . . 148
A.5 Libraries for Meta-Programming . . . 155
A.5.1 Library AbstractCurry . . . 155
A.5.2 Library AbstractCurryPrinter . . . 161
A.5.3 Library CompactFlatCurry . . . 162
A.5.4 Library CurryStringClassifier . . . 164
A.5.5 Library FlatCurry . . . 166
A.5.6 Library FlatCurryGoodies . . . 173
A.5.7 Library FlatCurryRead . . . 185
A.5.8 Library FlatCurryShow . . . 186
A.5.9 Library FlatCurryXML . . . 186
A.5.10 Library FlexRigid . . . 187
A.5.11 Library PrettyAbstract . . . 187

B Markdown Syntax **189**
B.1 Paragraphs and Basic Formatting . . . 189
B.2 Lists and Block Formatting . . . 190
B.3 Headers . . . 192

C Auxiliary Files **193**

D External Operations **194**

Index **197**

Preface

This document describes KiCS2 (**Ki**el **C**urry **S**ystem Version **2**), an implementation of the multi-paradigm language Curry [8, 16] 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 [17]. 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.

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

```
cleancurry
```

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

```
cleancurry -r
```

also deletes the auxiliary files in all subdirectories.

1.3 Restrictions

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

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

- KiCS2 translates all *local declarations* into global functions with additional arguments ("lambda lifting", see Appendix D of the Curry language report). Thus, in the various run-time systems, the definition of functions with local declarations look different from their original definition (in order to see the result of this transformation, you can use the CurryBrowser, see Section 6).

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

- Encapsulated search: The general definition of encapsulated search of the Curry report [15] 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

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

  to compute all values of a given argument expression. There is also the library `SearchTree` (Section A.2.28) 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.2.27) for more details.

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

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

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

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

  is non-deterministically evaluated to the two solutions

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

The current implementation of KiCS2 supports only flat module names, i.e., the notation `Dir.Mod.f` is not supported. In order to allow the structuring of modules in different directories, 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 "*kics2home*`/lib`" and "*kics2home*`/lib/meta`". This search path can be extended by setting the environment variable `CURRYPATH` (which can be also set in a KiCS2 session by the option "`:set path`", see below) to a list of directory names separated by colons ("`:`"). In addition, a local standard search path can be defined in the "`.kics2rc`" file (see Section 2.6). Thus, modules to be loaded are searched in the following directories (in this order, i.e., the first occurrence of a module file in this search path is imported):

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

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

2 Using the Interactive Environment of KiCS2

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

2.1 Invoking KiCS2

To start KiCS2, execute the command "`kics2`" (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 Command 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` *m* Add module *m* to the set of currently loaded modules so that its exported entities are available in the top-level environment.

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

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

```
xs++ys =:= [1,2]  where xs,ys free
```

in order to obtain the following three possible bindings:

```
{xs = [], ys = [1,2]} Success
{xs = [1], ys = [2]} Success
{xs = [1,2], ys = []} Success
```

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

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

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

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

:eval *expr* Same as *expr*. This command might be useful when putting commands as arguments when invoking `kics2`.

`:quit` Exit the system.

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

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

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

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

`:edit` Load the source code of the current main module into a text editor. If the variable `editcommand` is set in the configuration file "`.kics2rc`" (see Section 2.6), its value is used as an editor command, otherwise the environment variable "`EDITOR`" is used as the editor program.

`:edit` *file* Load file *file* into a text editor which is defined as in the command "`:edit`".

`:show` Show the source text of the currently loaded Curry program. If the variable `showcommand` is set in the configuration file "`.kics2rc`" (see Section 2.6), its value is used as a command to show the source text, otherwise the "`cat`" is used.

`:show` *m* Show the source text of module *m* which must be accessible via the current load path.

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

`: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.

`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).

$\boxed{\texttt{dfs}}$ Similarly to `bfs` but use a depth-first search strategy to compute and print the values of the initial expression.

$\boxed{\texttt{ids}}$ 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.

$\boxed{\texttt{ids}\ n}$ Similarly to `ids` but use an initial depth bound of n.

$\boxed{\texttt{par}}$ 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.

$\boxed{\texttt{par}\ n}$ Similarly to `par` but use n parallel threads.

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

$\boxed{\texttt{choices}\ n}$ Show the internal choice structure (according to the implementation described in [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.

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

$\boxed{\texttt{v}n}$ Set the verbosity level to n. The following values are allowed for n:

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

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

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

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

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

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

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

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

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

`+/-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.

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

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

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

```
:set rts -H512m
```

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

`args` *arguments* Define run-time arguments passed to the executable generated by the Glasgow Haskell Compiler. For instance, setting the option

```
:set args first second
```

has the effect that the I/O operation `getArgs` (see library `System` (Section A.2.30) 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 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).

2.6 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.

2.7 Emacs Interface

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

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

3 Extensions

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

3.1 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]`.

3.2 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:

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

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

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

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

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

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

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

In general, a pattern of the form (f $t_1 \ldots t_n$) ($n > 0$) 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@[(42,_)] ++ _) = x
```

is considered as syntactic sugar for the expanded definition

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

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

3.3 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.

3.4 Records

A record is a data structure for bundling several data of various types. It consists of typed data fields where each field is associated with a unique label. These labels can be used to construct, select or update fields in a record.

Unlike labeled data fields in Haskell, records are not syntactic sugar but a real extension of the language[2]. The basic concept is described in [19] but the current version does not yet provide all features mentioned there. The restrictions are explained in Section 3.4.7.

3.4.1 Record Type Declaration

It is necessary to declare a record type before a record can be constructed or used. The declaration has the following form:

type $R\ \alpha_1\ \ldots\ \alpha_n$ = { l_1 :: τ_1,..., l_m :: τ_m }

It introduces a new n-ary record type R which represents a record consisting of m fields. Each field has a unique label l_i representing a value of the type τ_i. Labels are identifiers which refer to the corresponding fields. The following examples define some record types:

```
type Person = {name :: String, age :: Int}
type Address = {person :: Person, street :: String, city :: String}
type Branch a b = {left :: a, right :: b}
```

It is possible to summarize different labels which have the same type. For instance, the record `Address` can also be declared as follows:

```
type Address = {person :: Person, street,city :: String}
```

The fields can occur in an arbitrary order. The example above can also be written as

```
type Address = {street,city :: String, person :: Person}
```

The record type can be used in every type expression to represent the corresponding record, e.g.

```
data BiTree = Node (Branch BiTree BiTree) | Leaf Int

getName :: Person → String
getName ...
```

Labels can only be used in the context of records. They do not share the name space with functions/constructors/variables or type constructors/type variables. For instance it is possible to use the same identifier for a label and a function at the same time. Label identifiers cannot be shadowed by other identifiers.

Like in type synonym declarations, recursive or mutually dependent record declarations are not allowed. Records can only be declared at the top level. Further restrictions are described in section 3.4.7.

[2] The current version allows to transform records into abstract data types. Future extensions may not have this facility.

3.4.2 Record Construction

The record construction generates a record with initial values for each data field. It has the following form:

{ l_1 := v_1,..., l_m := v_m }

It generates a record where each label l_i refers to the value v_i. The type of the record results from the record type declaration where the labels l_i are defined. A mix of labels from different record types is not allowed. All labels must be specified with exactly one assignment. Examples for record constructions are

```
{name := "Johnson", age := 30}     -- generates a record of type 'Person'
{left := True, right := 20}        -- generates a record of type 'Branch'
```

Assignments to labels can occur in an arbitrary order. For instance a record of type `Person` can also be generated as follows:

```
{age := 30, name := "Johnson"}     -- generates a record of type 'Person'
```

Unlike labeled fields in record type declarations, record constructions can be used in expressions without any restrictions (as well as all kinds of record expressions). For instance the following expression is valid:

```
{person := {name := "Smith", age := 20},   -- generates a record of
 street := "Main Street",                  -- type 'Address'
 city   := "Springfield"}
```

3.4.3 Field Selection

The field selection is used to extract data from records. It has the following form:

r :> l

It returns the value to which the label l refers to from the record expression r. The label must occur in the declaration of the record type of r. An example for a field selection is:

```
pers :> name
```

This returns the value of the label `name` from the record `pers` (which has the type `Person`). Sequential application of field selections are also possible:

```
addr :> person :> age
```

The value of the label `age` is extracted from a record which itself is the value of the label `person` in the record `addr` (which has the type `Address`).

3.4.4 Field Update

Records can be updated by reassigning a new value to a label:

{l_1 := v_1,..., l_k := v_k | r}

The label l_i is associated with the new value v_i which replaces the current value in the record r. The labels must occur in the declaration of the record type of r. In contrast to record constructions, it is not necessary to specify all labels of a record. Assignments can occur in an arbitrary order. It is not allowed to specify more than one assignment for a label in a record update. Examples for record updates are:

```
{name := "Scott", age := 25 | pers}
{person := {name := "Scott", age := 25 | pers} | addr}
```

In these examples `pers` is a record of type `Person` and `addr` is a record of type `Address`.

3.4.5 Records in Pattern Matching

It is possible to apply pattern matching to records (e.g., in functions, let expressions or case branches). Two kinds of record patterns are available:

{l_1 = p_1,..., l_n = p_n}
{l_1 = p_1,..., l_k = p_k | _}

In both cases each label l_i is specified with a pattern p_i. All labels must occur only once in the record pattern. The first case is used to match the whole record. Thus, all labels of the record must occur in the pattern. The second case is used to match only a part of the record. Here it is not necessary to specify all labels. This case is represented by a vertical bar followed by the underscore (anonymous variable). It is not allowed to use a pattern term instead of the underscore.

When trying to match a record against a record pattern, the patterns of the specified labels are matched against the corresponding values in the record expression. On success, all pattern variables occurring in the patterns are replaced by their actual expression. If none of the patterns matches, the computation fails.

Here are some examples of pattern matching with records:

```
isSmith30 :: Person → Bool
isSmith30 {name = "Smith", age = 30} = True

startsWith :: Char → Person → Bool
startsWith c {name = (d:_) | _} = c == d

getPerson :: Address → Person
getPerson {person = p | _} = p
```

As shown in the last example, a field selection can also be obtained by pattern matching.

3.4.6 Export of Records

Exporting record types and labels is very similar to exporting data types and constructors. There are three ways to specify an export:

- `module` M `(...,` R`, ...) where`
  exports the record R without any of its labels.
- `module` M `(...,` R`(..), ...) where`
  exports the record R together with all its labels.

- `module` M `(...,` R`(`l_1`,...,`l_k`), ...)  where`
  exports the record R together with the labels l_1, ..., l_k.

Note that imported labels cannot be overwritten in record declarations of the importing module. It is also not possible to import equal labels from different modules.

3.4.7 Restrictions in the Usage of Records

In contrast to the basic concept in [19], KiCS2/Curry provides a simpler version of records. Some of the features described there are currently not supported or even restricted.

- Labels must be unique within the whole scope of the program. In particular, it is not allowed to define the same label within different records, not even when they are imported from other modules. However, it is possible to use equal identifiers for other entities without restrictions, since labels have an independent name space.
- The record type representation with labeled fields can only be used as the right-hand-side of a record type declaration. It is not allowed to use it in any other type annotation.
- Records are not extensible or reducible. The structure of a record is specified in its record declaration and cannot be modified at the runtime of the program.
- Empty records are not allowed.
- It is not allowed to use a pattern term at the right side of the vertical bar in a record pattern except for the underscore (anonymous pattern variable).
- Labels cannot be sequentially associated with multiple values (record fields do not behave like stacks).

4 Recognized Syntax of Curry

The KiCS2 Curry compiler accepts a slightly extended version of the grammar specified in the Curry Report [17]. Furthermore, the syntax recognized by KiCS2 differs from that specified in the Curry Report regarding numeric or character literals. We therefore present the complete description of the syntax below, whereas syntactic extensions are highlighted.

4.1 Notational Conventions

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

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

The Curry files are expected to be encoded in 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

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

$$
\begin{array}{rcl}
\mathit{QTypeConstrID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{TypeConstrID} \\
\mathit{QDataConstrID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{DataConstrID} \\
\mathit{QInfixOpID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{InfixOpID} \\
\mathit{QFunctionID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{FunctionID} \\
\mathit{QVariableID} & ::= & [\mathit{ModuleID}\ \texttt{.}]\ \mathit{VariableID}
\end{array}
$$

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:

`..` `:` `::` `=` `\\` `|` `<-` $\rightarrow$ `@` `~` `:>` `:=`

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.

$$
\begin{array}{rcl}
\mathit{Int} & ::= & \mathit{Decimal} \\
 & | & \texttt{0o}\ \mathit{Octal} \mid \texttt{0O}\ \mathit{Octal} \\
 & | & \texttt{0x}\ \mathit{Hexadecimal} \mid \texttt{0X}\ \mathit{Hexadecimal} \\
 & & \\
\mathit{Float} & ::= & \mathit{Decimal}\ \texttt{.}\ \mathit{Decimal}\ [\mathit{Exponent}] \\
 & | & \mathit{Decimal}\ \mathit{Exponent} \\
\mathit{Exponent} & ::= & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \mathit{Decimal} \\
 & & \\
\mathit{Decimal} & ::= & \mathit{Digit}\ [\mathit{Decimal}] \\
\mathit{Octal} & ::= & \mathit{Octit}\ [\mathit{Octal}] \\
\mathit{Hexadecimal} & ::= & \mathit{Hexit}\ [\mathit{Hexadecimal}] \\
 & & \\
\mathit{Digit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\mathit{Octit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \\
\mathit{Hexit} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \\
 & & \\
\mathit{Char} & ::= & \texttt{'}(\mathit{Graphic}_{\langle\texttt{'}\mid\texttt{\textbackslash}\rangle} \mid \mathit{Space} \mid \mathit{Escape}_{\langle\texttt{\textbackslash\&}\rangle})\texttt{'} \\
\mathit{String} & ::= & \texttt{"}\{\mathit{Graphic}_{\langle\texttt{"}\mid\texttt{\textbackslash}\rangle} \mid \mathit{Space} \mid \mathit{Escape} \mid \mathit{Gap}\}\texttt{"} \\
\mathit{Escape} & ::= & \texttt{\textbackslash}\ (\mathit{CharEsc} \mid \mathit{Ascii} \mid \mathit{Decimal} \mid \texttt{o}\ \mathit{Octal} \mid \texttt{x}\ \mathit{Hexadecimal}) \\
\mathit{CharEsc} & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{f} \mid \texttt{n} \mid \texttt{r} \mid \texttt{t} \mid \texttt{v} \mid \texttt{\textbackslash} \mid \texttt{"} \mid \texttt{'} \mid \texttt{\&} \\
\mathit{Ascii} & ::= & \texttt{\^{}}\ \mathit{Cntrl} \mid \texttt{NUL} \mid \texttt{SOH} \mid \texttt{STX} \mid \texttt{ETX} \mid \texttt{EOT} \mid \texttt{ENQ} \mid \texttt{ACK} \\
 & | & \texttt{BEL} \mid \texttt{BS} \mid \texttt{HT} \mid \texttt{LF} \mid \texttt{VT} \mid \texttt{FF} \mid \texttt{CR} \mid \texttt{SO} \mid \texttt{SI} \mid \texttt{DLE}
\end{array}
$$

	|	`DC1` | `DC2` | `DC3` | `DC4` | `NAK` | `SYN` | `ETB` | `CAN`
	|	`EM` | `SUB` | `ESC` | `FS` | `GS` | `RS` | `US` | `SP` | `DEL`
Cntrl	::=	*AsciiLarge* | `@` | `[` | `\` | `]` | `^` | `_`
AsciiLarge	::=	`A` | ... | `Z`
Gap	::=	`\` *WhiteChar* {*WhiteChar*} `\`

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) * 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) * 2
```

or also as

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

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

4.4 Context Free Grammar

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

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

$$
\begin{array}{rcll}
\textit{Exports} & ::= & \texttt{(}\ \textit{Export}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{Export}_n\ \texttt{)} & (n \geq 0) \\
\textit{Export} & ::= & \textit{QFunctionName} & \\
 & | & \textit{QTypeConstrID}\ [\texttt{(}\ \textit{ConsLabelName}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{ConsLabelName}_n\ \texttt{)}] & (n \geq 0) \\
 & | & \textit{QTypeConstrID}\ \texttt{(..)} & \\
 & | & \texttt{module}\ \textit{ModuleID} & \\
\textit{ConsLabelName} & ::= & \textit{LabelID} \mid \textit{DataConstr} & \\
 & & & \\
\textit{Block} & ::= & \texttt{\{}\ [\textit{ImportDecl}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \textit{ImportDecl}_k\ \texttt{;}] & (\textit{no fixity declarations here}) \\
 & & \quad \textit{BlockDeclaration}_1\ \texttt{;}\ \ldots\ \texttt{;}\ \textit{BlockDeclaration}_n\ \texttt{\}} & (k, n \geq 0) \\
 & & & \\
\textit{ImportDecl} & ::= & \texttt{import}\ [\texttt{qualified}]\ \textit{ModuleID}\ [\texttt{as}\ \textit{ModuleID}]\ [\textit{ImportRestr}] & \\
\textit{ImportRestr} & ::= & \texttt{(}\ \textit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{Import}_n\ \texttt{)} & (n \geq 0) \\
 & | & \texttt{hiding}\ \texttt{(}\ \textit{Import}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{Import}_n\ \texttt{)} & (n \geq 0) \\
\textit{Import} & ::= & \textit{FunctionName} & \\
 & | & \textit{TypeConstrID}\ [\texttt{(}\ \textit{ConsLabelName}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{ConsLabelName}_n\ \texttt{)}] & (n \geq 0) \\
 & | & \textit{TypeConstrID}\ \texttt{(..)} & \\
 & & & \\
\textit{BlockDeclaration} & ::= & \textit{TypeSynonymDecl} & \\
 & | & \textit{DataDeclaration} & \\
 & | & \textit{FixityDeclaration} & \\
 & | & \textit{FunctionDeclaration} & \\
 & & & \\
\textit{TypeSynonymDecl} & ::= & \texttt{type}\ \textit{SimpleType}\ \texttt{=}\ (\textit{TypeExpr} \mid \textit{RecordType}) & \\
\textit{SimpleType} & ::= & \textit{TypeConstrID}\ \textit{TypeVarID}_1\ \ldots\ \textit{TypeVarID}_n & (n \geq 0) \\
\textit{TypeConstrID} & ::= & \text{see lexicon} & \\
 & & & \\
\textit{RecordType} & ::= & \texttt{\{}\ \textit{LabelDecl}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{LabelDecl}_n\ \texttt{\}} & (n > 0) \\
\textit{LabelDecl} & ::= & \textit{LabelID}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{LabelID}_n\ \texttt{::}\ \textit{TypeExpr} & (n > 0) \\
\textit{LabelID} & ::= & \text{see lexicon} & \\
 & & & \\
\textit{DataDeclaration} & ::= & \texttt{data}\ \textit{SimpleType} & (\textit{external data type}) \\
 & | & \texttt{data}\ \textit{SimpleType}\ \texttt{=}\ \textit{ConstrDecl}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \textit{ConstrDecl}_n & (n > 0) \\
\textit{ConstrDecl} & ::= & \textit{DataConstr}\ \textit{SimpleTypeExpr}_1\ \ldots\ \textit{SimpleTypeExpr}_n & (n \geq 0) \\
 & | & \textit{SimpleTypeExpr}\ \textit{ConsOp}\ \textit{SimpleTypeExpr} & (\textit{infix data constructor}) \\
 & & & \\
\textit{TypeExpr} & ::= & \textit{SimpleTypeExpr}\ [\texttt{->}\ \textit{TypeExpr}] & \\
\textit{SimpleTypeExpr} & ::= & \textit{QTypeConstrID}\ \textit{SimpleTypeExpr}_1\ \ldots\ \textit{SimpleTypeExpr}_n & (n \geq 0) \\
 & | & \textit{TypeVarID} & \\
 & | & \texttt{()} & (\textit{unit type}) \\
 & | & \texttt{(}\ \textit{TypeExpr}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{TypeExpr}_n\ \texttt{)} & (\textit{tuple type},\ n > 1) \\
 & | & \texttt{[}\ \textit{TypeExpr}\ \texttt{]} & (\textit{list type}) \\
 & | & \texttt{(}\ \textit{TypeExpr}\ \texttt{)} & (\textit{parenthesized type}) \\
\textit{TypeVarID} & ::= & \text{see lexicon} & \\
 & & & \\
\textit{FixityDeclaration} & ::= & \textit{FixityKeyword}\ \textit{Digit}\ \textit{InfixOpID}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{InfixOpID}_n & (n > 0) \\
\textit{FixityKeyword} & ::= & \texttt{infixl} \mid \texttt{infixr} \mid \texttt{infix} & \\
\textit{InfixOpID} & ::= & \text{see lexicon} & \\
 & & & \\
\textit{FunctionDeclaration} & ::= & \textit{Signature} \mid \textit{External} \mid \textit{Equat} & \\
\textit{External} & ::= & \textit{FunctionNames}\ \texttt{external} & (\textit{externally defined functions}) \\
\textit{Signature} & ::= & \textit{FunctionNames}\ \texttt{::}\ \textit{TypeExpr} & \\
\textit{FunctionNames} & ::= & \textit{FunctionName}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \textit{FunctionName}_n & (n > 0) \\
\textit{Equat} & ::= & \textit{FunLHS}\ \texttt{=}\ \textit{TypedExpr}\ [\texttt{where}\ \textit{LocalDefs}] & \\
 & | & \textit{FunLHS}\ \textit{CondExprs}\ [\texttt{where}\ \textit{LocalDefs}] &
\end{array}
$$

<table>
<tr><td>FunLHS</td><td>::=</td><td>FunctionName SimplePat$_1$ … SimplePat$_n$</td><td>($n \geq 0$)</td></tr>
<tr><td></td><td>|</td><td>SimplePat InfixOpID SimplePat</td><td></td></tr>
<tr><td>CondExprs</td><td>::=</td><td><code>|</code> InfixExpr <code>=</code> TypedExpr [CondExprs]</td><td></td></tr>
<tr><td>Pattern</td><td>::=</td><td>ConsPattern [QConsOp Pattern]</td><td>(infix constructor pattern)</td></tr>
<tr><td>ConsPattern</td><td>::=</td><td>GDataConstr SimplePat$_1$ … SimplePat$_n$</td><td>(constructor pattern)</td></tr>
<tr><td></td><td>|</td><td>SimplePat</td><td></td></tr>
<tr><td>SimplePat</td><td>::=</td><td>Variable</td><td></td></tr>
<tr><td></td><td>|</td><td><code>_</code></td><td>(wildcard)</td></tr>
<tr><td></td><td>|</td><td>QDataConstr</td><td></td></tr>
<tr><td></td><td>|</td><td>Literal</td><td></td></tr>
<tr><td></td><td>|</td><td><code>-</code> Int</td><td>(negative pattern)</td></tr>
<tr><td></td><td>|</td><td><code>-.</code> Float</td><td>(negative float pattern)</td></tr>
<tr><td></td><td>|</td><td><code>()</code></td><td>(empty tuple pattern)</td></tr>
<tr><td></td><td>|</td><td><code>(</code> Pattern$_1$ <code>,</code> … <code>,</code> Pattern$_n$ <code>)</code></td><td>($n > 1$)</td></tr>
<tr><td></td><td>|</td><td><code>(</code> Pattern <code>)</code></td><td>(parenthesized pattern)</td></tr>
<tr><td></td><td>|</td><td><code>[</code> Pattern$_1$ <code>,</code> … <code>,</code> Pattern$_n$ <code>]</code></td><td>($n \geq 0$)</td></tr>
<tr><td></td><td>|</td><td>Variable <code>@</code> SimplePat</td><td>(as-pattern)</td></tr>
<tr><td></td><td>|</td><td><code>~</code> SimplePat</td><td>(irrefutable pattern)</td></tr>
<tr><td></td><td>|</td><td><code>(</code> SimplePat QFunOp SimplePat <code>)</code></td><td>(infix functional pattern)</td></tr>
<tr><td></td><td>|</td><td><code>(</code> QFunctionName SimplePat$_1$ … SimplePat$_n$ <code>)</code></td><td>(functional pattern, $n > 0$)</td></tr>
<tr><td></td><td>|</td><td><code>{</code> FieldPat$_1$ <code>,</code> … <code>,</code> FieldPat$_n$ [<code>|</code> <code>_</code>] <code>}</code></td><td>(record pattern)</td></tr>
<tr><td>FieldPat</td><td>::=</td><td>LabelID <code>=</code> Pattern</td><td></td></tr>
<tr><td>LocalDefs</td><td>::=</td><td>{ValueDeclaration$_1$ <code>;</code> … <code>;</code> ValueDeclaration$_n$ }</td><td>($n > 0$)</td></tr>
<tr><td>ValueDeclaration</td><td>::=</td><td>FunctionDeclaration</td><td></td></tr>
<tr><td></td><td>|</td><td>PatternDeclaration</td><td></td></tr>
<tr><td></td><td>|</td><td>VariableID$_1$ <code>,</code> … <code>,</code> VariableID$_n$ <code>free</code></td><td>($n > 0$)</td></tr>
<tr><td></td><td>|</td><td>FixityDeclaration</td><td></td></tr>
<tr><td>PatternDeclaration</td><td>::=</td><td>Pattern <code>=</code> TypedExpr [<code>where</code> LocalDefs]</td><td></td></tr>
<tr><td>TypedExpr</td><td>::=</td><td>InfixExpr <code>::</code> TypeExpr</td><td>(expression type signature)</td></tr>
<tr><td></td><td>|</td><td>InfixExpr</td><td></td></tr>
<tr><td>InfixExpr</td><td>::=</td><td>Expr QOp InfixExpr</td><td>(infix operator application)</td></tr>
<tr><td></td><td>|</td><td><code>-</code> InfixExpr</td><td>(unary int minus)</td></tr>
<tr><td></td><td>|</td><td><code>-.</code> InfixExpr</td><td>(unary float minus)</td></tr>
<tr><td></td><td>|</td><td>Expr</td><td></td></tr>
<tr><td>Expr</td><td>::=</td><td><code>\</code> SimplePat$_1$ … SimplePat$_n$ <code>-></code> TypedExpr</td><td>(lambda expression, $n > 0$)</td></tr>
<tr><td></td><td>|</td><td><code>let</code> LocalDefs <code>in</code> TypedExpr</td><td>(let expression)</td></tr>
<tr><td></td><td>|</td><td><code>if</code> TypedExpr <code>then</code> TypedExpr <code>else</code> TypedExpr</td><td>(conditional)</td></tr>
<tr><td></td><td>|</td><td><code>case</code> TypedExpr <code>of</code> {Alt$_1$ <code>;</code> … <code>;</code> Alt$_n$ }</td><td>(case expression, $n \geq 0$)</td></tr>
<tr><td></td><td>|</td><td><code>fcase</code> TypedExpr <code>of</code> {Alt$_1$ <code>;</code> … <code>;</code> Alt$_n$ }</td><td>(fcase expression, $n \geq 0$)</td></tr>
<tr><td></td><td>|</td><td><code>do</code> { Stmt$_1$ <code>;</code> … <code>;</code> Stmt$_n$ <code>;</code> TypedExpr }</td><td>(do expression, $n \geq 0$)</td></tr>
<tr><td></td><td>|</td><td>FunctExpr</td><td></td></tr>
<tr><td>FunctExpr</td><td>::=</td><td>[FunctExpr] BasicExpr</td><td>(function application)</td></tr>
<tr><td></td><td>|</td><td>FunctExpr <code>:></code> LabelID</td><td>(record selection)</td></tr>
<tr><td>BasicExpr</td><td>::=</td><td>QVariableID</td><td>(variable)</td></tr>
<tr><td></td><td>|</td><td><code>_</code></td><td>(anonymous free variable)</td></tr>
<tr><td></td><td>|</td><td>QFunctionName</td><td>(qualified function)</td></tr>
<tr><td></td><td>|</td><td>GDataConstr</td><td>(general constructor)</td></tr>
<tr><td></td><td>|</td><td>Literal</td><td></td></tr>
</table>

	|	`(` *TypedExpr* `)`	(*parenthesized expression*)
	|	`(` $TypedExpr_1$ `,` ... `,` $TypedExpr_n$ `)`	(*tuple*, $n > 1$)
	|	`[` $TypedExpr_1$ `,` ... `,` $TypedExpr_n$ `]`	(*finite list*, $n > 0$)
	|	`[` *TypedExpr* [`,` *TypedExpr*] `..` [*TypedExpr*] `]`	(*arithmetic sequence*)
	|	`[` *TypedExpr* `|` $Qual_1$ `,` ... `,` $Qual_n$ `]`	(*list comprehension*, $n \geq 1$)
	|	`(` *InfixExpr* *QOp* `)`	(*left section*)
	|	`(` $QOp_{\langle -,-.\rangle}$ *InfixExpr* `)`	(*right section*)
	|	`{` $FBind_1$ `,` ... `,` $FBind_n$ `}`	(*labeled construction*, $n > 0$)
	|	`{` $FBind_1$ `,` ... `,` $FBind_n$ `|` *TypedExpr* `}`	(*labeled update*, $n > 0$)
Alt	::=	*Pattern* `->` *TypedExpr* [`where` *LocalDefs*]	
	|	*Pattern* *GdAlts* [`where` *LocalDefs*]	
GdAlts	::=	`|` *TypedExpr* `->` *TypedExpr* [*GdAlts*]	
FBind	::=	*LabelID* `:=` *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., "`<`"). 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

[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.

references to names of operations or data types in Curry programs which are translated into links inside the generated HTML documentation. Such references have to be enclosed in single quotes. For instance, the text `'conc'` refers to the Curry operation `conc` inside the current module whereas the text `'Prelude.reverse'` refers to the operation `reverse` of the module `Prelude`. If one wants to write single quotes without this specific meaning, one can escape them with a backslash:

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

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

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

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

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

module Example where

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

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

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

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 ...Mod`n : This command generates only the index pages (i.e., a main index page and index pages for all functions and constructors defined in the modules `Mod1`, `M2`,. . . ,`Mod`n and their imported modules) in the directory `docdir`.

6 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 [18] or pattern matching is translated into case expressions [8, 20]. 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.

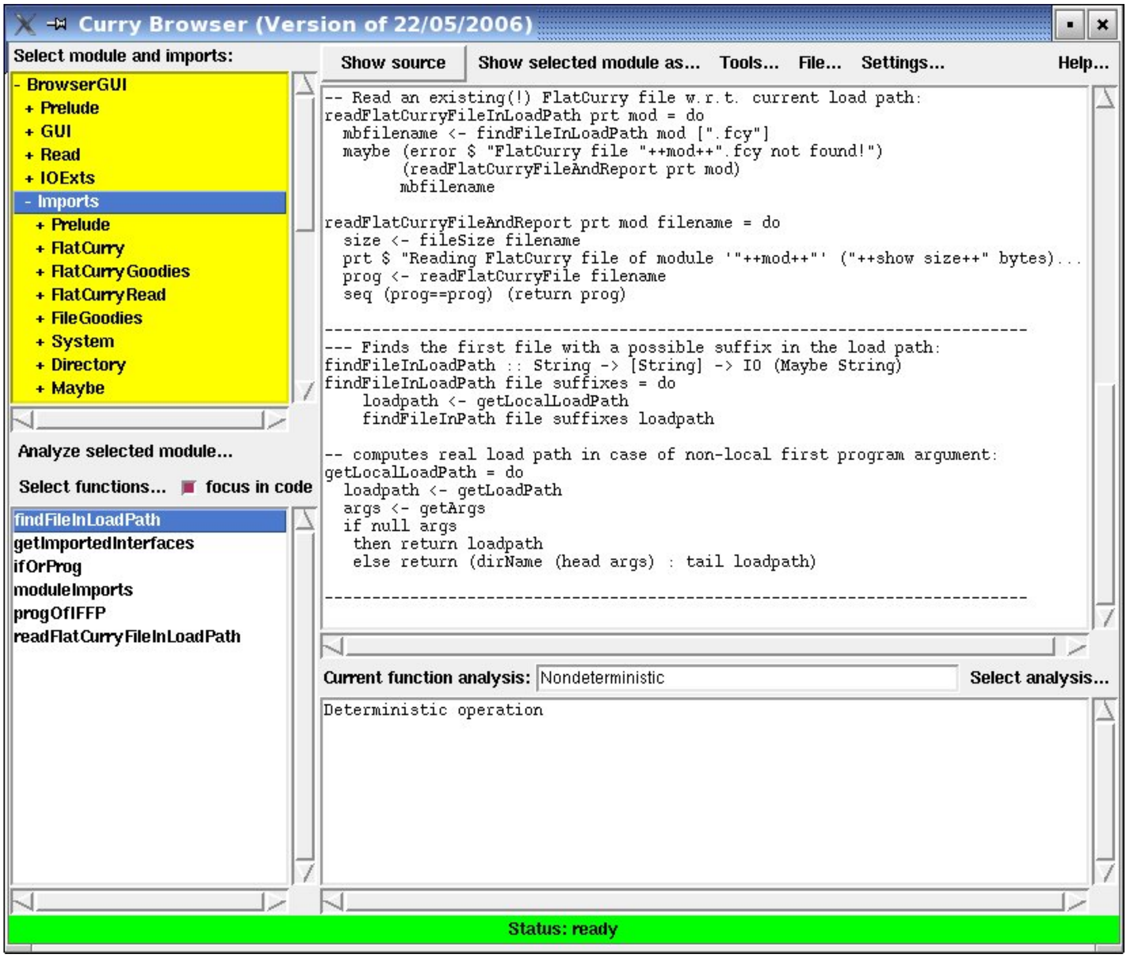

Figure 1: Snapshot of the main window of CurryBrowser

analyses are local, i.e., they need only to consider the local definition of this function (e.g., "Calls directly," "Overlapping rules," "Pattern completeness"), where other analyses are global, i.e., they consider the definitions of all functions directly or indirectly called by this function (e.g., "Depends on," "Solution complete," "Set-valued"). Finally, there are a few additional tools integrated into CurryBrowser, for instance, to visualize the import relation between all modules as a dependency graph. These tools are available through the "Tools" menu.

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

7 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:

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

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

```
import List
import Assertion

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

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

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

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

We can execute a test suite by the command

```
currytest TestList
```

(`currytest` is a program stored in *kics2home*`/bin` where *kics2home* is the installation directory of KiCS2; see Section 1.1). 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

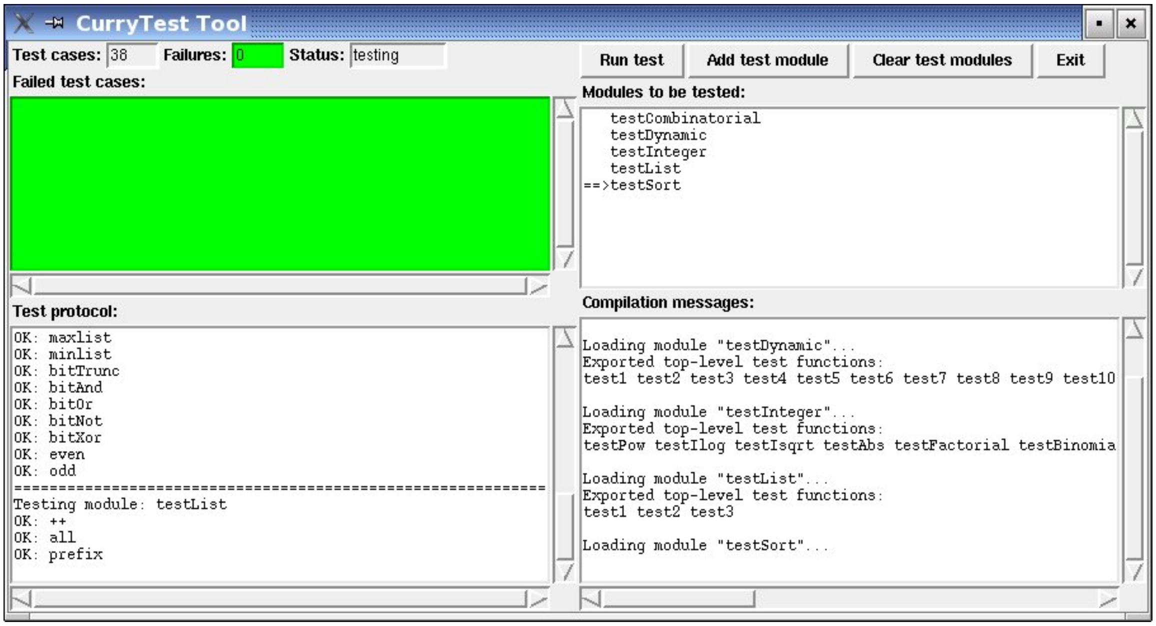

Figure 2: Snapshot of CurryTest's graphical interface

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.

8 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 myerd.xmi
```

(`erd2curry` is a program stored in *kics2home*`/bin` where *kics2home* is the installation directory of KiCS2; see Section 1.1). 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].

If one does not want to use the Umbrello UML Modeller, 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*`/tools/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
```

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 2.6, according to your local environment). This can be done by the command

```
erd2curry -v myerd.term
```

Inclusion in the Curry application: To compile the generated database code, either include the directory *kics2home*`/tools/erd2curry` into your Curry load path (e.g., by setting the environment variable "`CURRYPATH`", see also Section 1.4) or copy the file *kics2home*`/tools/erd2curry/ERDGeneric.curry` into the directory of the generated database code.

9 Technical Problems

One can implement distributed systems with KiCS2 by the use of the library `NamedSocket` (Section A.2.19) 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.4.6). 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

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

[2] S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.

[3] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.

[4] B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

[5] B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

[6] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing equational constraints in a functional language. In *Proc. of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011)*, pages 22–33. INFSYS Research Report 1843-11-06 (TU Wien), 2011.

[7] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.

[8] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[9] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[10] M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[11] M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[12] M. Hanus. A generic analysis environment for declarative programs. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 43–48. ACM Press, 2005.

[13] M. Hanus. CurryBrowser: A generic analysis environment for Curry programs. In *Proc. of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE'06)*, pages 61–74, 2006.

[14] M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.

[15] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.

[16] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

[17] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

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

[19] D. Leijen. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05)*, 2005.

[20] P. Wadler. Efficient compilation of pattern-matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.

A Libraries of the KiCS2 Distribution

The KiCS2 distribution comes with an extensive collection of libraries for application programming. The libraries for meta-programming by representing Curry programs as datatypes in Curry are described in the following subsection in more detail. The complete set of libraries with all exported types and functions are described in the further subsections. For a more detailed online documentation of all libraries of KiCS2, see `http://www-ps.informatik.uni-kiel.de/kics2/lib/index.html`.

A.1 AbstractCurry and FlatCurry: Meta-Programming in Curry

To support meta-programming, i.e., the manipulation of Curry programs in Curry, there are system modules `FlatCurry` (Section A.5.5) 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 documentation[6] 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],
```

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

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

A.2 General Libraries

A.2.1 Library AllSolutions

This module contains a collection of functions for obtaining lists of solutions to constraints. These operations are useful to encapsulate non-deterministic operations between I/O actions in order to connects 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 anomalities in the old concept.

Exported functions:

`getAllValues :: a → IO [a]`

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

`getOneValue :: a → IO (Maybe a)`

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

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

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

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

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

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

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

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

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

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

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

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

`showTestMod :: Int → String → IO ()`

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

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

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

`showTestEnd :: Int → IO ()`

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

`showTestCompileError :: Int → IO ()`

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

A.2.3 Library Char

Library with some useful functions on characters.

Exported functions:

`isUpper :: Char → Bool`

Returns true if the argument is an uppercase letter.

`isLower :: Char → Bool`

Returns true if the argument is an lowercase letter.

`isAlpha :: Char → Bool`

Returns true if the argument is a letter.

`isDigit :: Char → Bool`

Returns true if the argument is a decimal digit.

`isAlphaNum :: Char → Bool`

Returns true if the argument is a letter or digit.

`isOctDigit :: Char → Bool`

Returns true if the argument is an octal digit.

`isHexDigit :: Char → Bool`

Returns true if the argument is a hexadecimal digit.

`isSpace :: Char → Bool`

Returns true if the argument is a white space.

`toUpper :: Char → Char`

Converts lowercase into uppercase letters.

`toLower :: Char → Char`

Converts uppercase into lowercase letters.

`digitToInt :: Char → Int`

Converts a (hexadecimal) digit character into an integer.

`intToDigit :: Int → Char`

Converts an integer into a (hexadecimal) digit character.

A.2.4 Library Combinatorial

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

Exported functions:

`permute :: [a] → [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 [1,2,3], [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.

`(>=:) :: a → a → Success`

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

`andC :: [Success] → Success`

Evaluates the conjunction of a list of constraints.

`orC :: [Success] → Success`

Evaluates the disjunction of a list of constraints.

`allC :: (a → Success) → [a] → Success`

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

`anyC :: (a → Success) → [a] → Success`

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

A.2.6 Library CSV

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

Exported functions:

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

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

`showCSV :: [[String]] → String`

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

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

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

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

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

`readCSV :: String → [[String]]`

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

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

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

A.2.7 Library Directory

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

Exported functions:

`doesFileExist :: String → IO Bool`

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

`doesDirectoryExist :: String → IO Bool`

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

`fileSize :: String → IO Int`

Returns the size of the file.

`getModificationTime :: String → IO ClockTime`

Returns the modification time of the file.

`getCurrentDirectory :: IO String`

Returns the current working directory.

`setCurrentDirectory :: String → IO ()`

Sets the current working directory.

`getDirectoryContents :: String → IO [String]`

Returns the list of all entries in a directory.

`createDirectory :: String → IO ()`

Creates a new directory with the given name.

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

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

`removeDirectory :: String → IO ()`

Deletes a directory from the file system.

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

Renames a directory.

`getHomeDirectory :: IO String`

Return the home directory of the current user.

`getTemporaryDirectory :: IO String`

Return the temporary directory of the operating system.

`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.2.8 Library FileGoodies

A collection of useful operations when dealing with files.

Exported functions:

`separatorChar :: Char`

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

`pathSeparatorChar :: Char`

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

`suffixSeparatorChar :: Char`

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

`isAbsolute :: String → Bool`

Is the argument an absolute name?

`dirName :: String → String`

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

`baseName :: String → String`

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

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

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

`stripSuffix :: String → String`

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

`fileSuffix :: String → String`

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

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

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

`splitPath :: String → [String]`

Splits a path string into list of directory names.

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

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

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

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

A.2.9 Library Float

A collection of operations on floating point numbers.

Exported functions:

`(+.) :: Float → Float → Float`

Addition on floats.

`(-.) :: Float → Float → Float`

Subtraction on floats.

`(*.) :: Float → Float → Float`

Multiplication on floats.

`(/.) :: Float → Float → Float`

Division on floats.

`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.

`sqrt :: Float → Float`

Square root.

`log :: Float → Float`

Natural logarithm.

`exp :: Float → Float`

Natural exponent.

`sin :: Float → Float`

Sine.

`cos :: Float → Float`

Cosine.

`tan :: Float → Float`

Tangent.

`atan :: Float → Float`

Arc tangent.

A.2.10 Library Global

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

```
g :: Global t
g = global v spec
```

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

Exported types:

`data Global`

The abstract type of a global entity.

Exported constructors:

`data GlobalSpec`

The storage mechanism for the global entity.

Exported constructors:

- `Temporary :: GlobalSpec`

  `Temporary`

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

- `Persistent :: String → GlobalSpec`

  `Persistent f`

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

Exported functions:

`global :: a → GlobalSpec → Global a`

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

`readGlobal :: Global a → IO a`

Reads the current value of a global.

`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.2.11 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

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

  `StreamHandler hdl handler`

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

- `RemoveStreamHandler :: Handle → ReconfigureItem`

  `RemoveStreamHandler hdl`

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

`data Event`

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

Exported constructors:

- `DefaultEvent :: Event`

  `DefaultEvent`

  – the default event of the widget

- `MouseButton1 :: Event`

  `MouseButton1`

  – left mouse button pressed

- `MouseButton2 :: Event`

  `MouseButton2`

  – middle mouse button pressed

- `MouseButton3 :: Event`

  `MouseButton3`

  – right mouse button pressed

- `KeyPress :: Event`

  `KeyPress`

  – any key is pressed

- `Return :: Event`

  `Return`

– return key is pressed

`data ConfCollection`

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

Exported constructors:

- `CenterAlign :: ConfCollection`

  `CenterAlign`

  – centered alignment

- `LeftAlign :: ConfCollection`

  `LeftAlign`

  – left alignment

- `RightAlign :: ConfCollection`

  `RightAlign`

  – right alignment

- `TopAlign :: ConfCollection`

  `TopAlign`

  – top alignment

- `BottomAlign :: ConfCollection`

  `BottomAlign`

  – bottom alignment

`data MenuItem`

The data type for specifying items in a menu.

Exported constructors:

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

  `MButton`

  – a button with an associated command and a label string

- `MSeparator :: MenuItem`

  `MSeparator`

  – a separator between menu entries

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

  `MMenuButton`

  – a submenu with a label string

`data CanvasItem`

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

Exported constructors:

- `CLine :: [(Int,Int)] → String → CanvasItem`
- `CPolygon :: [(Int,Int)] → String → CanvasItem`
- `CRectangle :: (Int,Int) → (Int,Int) → String → CanvasItem`
- `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 message stream. This operation is useful to run a GUI that should react on user events as well as messages sent to an external port.

`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 message stream. This operation is useful to run a GUI that should react on user events as well as messages sent to an external port.

`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 sent to an external port.

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

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

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

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

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

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

`exitGUI :: GuiPort → IO ()`

An event handler for terminating the GUI.

`getValue :: WidgetRef → GuiPort → IO String`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`focusInput :: WidgetRef → GuiPort → IO ()`

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

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

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

`popup_message :: String → IO ()`

A simple popup message.

`Cmd :: (GuiPort → IO ()) → ConfItem`

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

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

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

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

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

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

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

`TextEditScroll :: [ConfItem] → Widget`

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

`ListBoxScroll :: [ConfItem] → Widget`

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

`CanvasScroll :: [ConfItem] → Widget`

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

`EntryScroll :: [ConfItem] → Widget`

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

`getOpenFile :: IO String`

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

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

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

`getSaveFile :: IO String`

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

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

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

`chooseColor :: IO String`

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

A.2.12 Library Integer

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

Exported functions:

`pow :: Int → Int → Int`

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

`ilog :: Int → Int`

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

`isqrt :: Int → Int`

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

`factorial :: Int → Int`

The value of *factorial n* is the factorial of *n*. Fails if $n < 0$.

`binomial :: Int → Int → Int`

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

`abs :: Int → Int`

The value of *abs n* is the absolute value of n.

`max3 :: a → a → a → a`

Returns the maximum of the three arguments.

`min3 :: a → a → a → a`

Returns the minimum of the three arguments.

`maxlist :: [a] → a`

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

`minlist :: [a] → a`

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

`bitTrunc :: Int → Int → Int`

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

`bitAnd :: Int → Int → Int`

Returns the bitwise AND of the two arguments.

`bitOr :: Int → Int → Int`

Returns the bitwise inclusive OR of the two arguments.

`bitNot :: Int → Int`

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

`bitXor :: Int → Int → Int`

Returns the bitwise exclusive OR of the two arguments.

`even :: Int → Bool`

Returns whether an integer is even

`odd :: Int → Bool`

Returns whether an integer is odd

A.2.13 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).

`hReady :: Handle → IO Bool`

Checks whether an input is available on a given handle.

`hGetChar :: Handle → IO Char`

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

`hGetLine :: Handle → IO String`

Reads a line from an input handle and returns it. Throws an error if the end of file has been reached while reading the *first* character. If the end of file is reached later in the line, it ist treated as a line terminator and the (partial) line is returned.

`hGetContents :: Handle → IO String`

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

`getContents :: IO String`

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

`hPutChar :: Handle → Char → IO ()`

Puts a character to an output handle.

`hPutStr :: Handle → String → IO ()`

Puts a string to an output handle.

`hPutStrLn :: Handle → String → IO ()`

Puts a string with a newline to an output handle.

`hPrint :: Handle → a → IO ()`

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

`hIsReadable :: Handle → IO Bool`

Is the handle readable?

`hIsWritable :: Handle → IO Bool`

Is the handle writable?

A.2.14 Library IOExts

Library with some useful extensions to the IO monad.

Exported types:

`data IORef`

Mutable variables containing values of some type. The values are not evaluated when they are assigned to an IORef.

Exported constructors:

Exported functions:

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

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

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

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

`connectToCommand :: String → IO Handle`

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

`readCompleteFile :: String → IO String`

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

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

> An action that updates the contents of a file.

`exclusiveIO :: String → IO a → IO a`

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

`setAssoc :: String → String → IO ()`

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

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

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

`newIORef :: a → IO (IORef a)`

> Creates a new IORef with an initial 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.2.15 Library JavaScript

A library to represent JavaScript programs.

Exported types:

`data JSExp`

Type of JavaScript expressions.

Exported constructors:

- `JSString :: String → JSExp`

  `JSString`

  - string constant

- `JSInt :: Int → JSExp`

  `JSInt`

  - integer constant

- `JSBool :: Bool → JSExp`

  `JSBool`

  - Boolean constant

- `JSIVar :: Int → JSExp`

  `JSIVar`

  - indexed variable

- `JSIArrayIdx :: Int → Int → JSExp`

  `JSIArrayIdx`

  - array access to index array variable

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

  `JSOp`

– infix operator expression

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

  `JSFCall`

  – function call

- `JSApply :: JSExp → JSExp → JSExp`

  `JSApply`

  – function call where the function is an expression

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

  `JSLambda`

  – (anonymous) function with indexed variables as arguments

`data JSStat`

Type of JavaScript statements.

Exported constructors:

- `JSAssign :: JSExp → JSExp → JSStat`

  `JSAssign`

  – assignment

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

  `JSIf`

  – conditional

- `JSSwitch :: JSExp → [JSBranch] → JSStat`

  `JSSwitch`

  – switch statement

- `JSPCall :: String → [JSExp] → JSStat`

  `JSPCall`

  – procedure call

- `JSReturn :: JSExp → JSStat`

  `JSReturn`

  – return statement

- `JSVarDecl :: Int → JSStat`

  `JSVarDecl`

– local variable declaration

`data JSBranch`

Exported constructors:

- `JSCase :: String → [JSStat] → JSBranch`

  `JSCase`

  – case branch

- `JSDefault :: [JSStat] → JSBranch`

  `JSDefault`

  – default branch

`data JSFDecl`

Exported constructors:

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

Exported functions:

`showJSExp :: JSExp → String`

Shows a JavaScript expression as a string in JavaScript syntax.

`showJSStat :: Int → JSStat → String`

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

`showJSFDecl :: JSFDecl → String`

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

`jsConsTerm :: String → [JSExp] → JSExp`

Representation of constructor terms in JavaScript.

A.2.16 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:

`data Query`

Queries can read but not write to the database.

Exported constructors:

`data Transaction`

Transactions can modify the database and are executed atomically.

Exported constructors:

`data Dynamic`

Result type of database predicates.

Exported constructors:

`data ColVal`

Abstract type for value restrictions

Exported constructors:

`data TError`

The type of errors that might occur during a transaction.

Exported constructors:

- `TError :: TErrorKind → String → TError` `data TErrorKind`

The various kinds of transaction errors.

Exported constructors:

- `KeyNotExistsError :: TErrorKind`
- `NoRelationshipError :: TErrorKind`
- `DuplicateKeyError :: TErrorKind`
- `KeyRequiredError :: TErrorKind`
- `UniqueError :: TErrorKind`
- `MinError :: TErrorKind`
- `MaxError :: TErrorKind`
- `UserDefinedError :: TErrorKind`
- `ExecutionError :: TErrorKind`

Exported functions:

`runQ :: Query a → IO a`

Runs a database query in the IO monad.

`transformQ :: (a → b) → Query a → Query b`

Applies a function to the result of a database query.

`runT :: Transaction a → IO (Either a TError)`

Runs a transaction atomically in the IO monad.

Transactions are *immediate*, which means that locks are acquired on all databases as soon as the transaction is started. After one transaction is started, no other database connection will be able to write to the database or start a transaction. Other connections *can* read the database during a transaction of another process.

The choice to use immediate rather than deferred transactions is conservative. It might also be possible to allow multiple simultaneous transactions that lock tables on the first database access (which is the default in SQLite). However this leads to unpredictable order in which locks are taken when multiple databases are involved. The current implementation fixes the locking order by sorting databases by their name and locking them in order immediately when a transaction begins.

More information on [7] _transaction.html”>transactions in SQLite is available online.

[7] `http://sqlite.org/lang`

`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 transaction sequentially, and collects their results.

`mapT_ :: (a → Transaction b) → [a] → Transaction ()`

Applies a function that yields transactions to all elements of a list, executes the transactions sequentially, and ignores their results.

`persistentSQLite :: String → String → [String] → Int → a → Dynamic`

This function is used instead of `dynamic` or `persistent` to declare predicates whose facts are stored in an SQLite database.

If the provided database or the table do not exist they are created automatically when the declared predicate is accessed for the first time.

Multiple column names can be provided if the second argument of the predicate is a tuple with a matching arity. Other record types are not supported. If no column names are provided a table with a single column called `info` is created. Columns of name *rowid* are not supported and lead to a run-time error.

`existsDBKey :: (Int → a → Dynamic) → Int → Query Bool`

Checks whether the predicate has an entry with the given key.

`allDBKeys :: (Int → a → Dynamic) → Query [Int]`

Returns a list of all stored keys. Do not use this function unless the database is small.

`allDBInfos :: (Int → a → Dynamic) → Query [a]`

Returns a list of all info parts of stored entries. Do not use this function unless the database is small.

`allDBKeyInfos :: (Int → a → Dynamic) → Query [(Int,a)]`

Returns a list of all stored entries. Do not use this function unless the database is small.

`(@=) :: Int → a → ColVal`

Constructs a value restriction for the column given as first argument

`someDBKeys :: (Int → a → Dynamic) → [ColVal] → Query [Int]`

Returns a list of those stored keys where the corresponding info part matches the gioven value restriction. Safe to use even on large databases if the number of results is small.

`someDBInfos :: (Int → a → Dynamic) → [ColVal] → Query [a]`

Returns a list of those info parts of stored entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

`someDBKeyInfos :: (Int → a → Dynamic) → [ColVal] → Query [(Int,a)]`

Returns a list of those entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

someDBKeyProjections :: (Int → a → Dynamic) → [Int] → [ColVal] → Query [(Int,b)]

Returns a list of column projections on those entries that match the given value restrictions for columns. Safe to use even on large databases if the number of results is small.

getDBInfo :: (Int → a → Dynamic) → Int → Query (Maybe a)

Queries the information stored under the given key. Yields Nothing if the given key is not present.

getDBInfos :: (Int → a → Dynamic) → [Int] → Query (Maybe [a])

Queries the information stored under the given keys. Yields Nothing if a given key is not present.

deleteDBEntry :: (Int → a → Dynamic) → Int → Transaction ()

Deletes the information stored under the given key. If the given key does not exist this transaction is silently ignored and no error is raised.

deleteDBEntries :: (Int → a → Dynamic) → [Int] → Transaction ()

Deletes the information stored under the given keys. No error is raised if (some of) the keys do not exist.

updateDBEntry :: (Int → a → Dynamic) → Int → a → Transaction ()

Updates the information stored under the given key. The transaction is aborted with a KeyNotExistsError if the given key is not present in the database.

newDBEntry :: (Int → a → Dynamic) → a → Transaction Int

Stores new information in the database and yields the newly generated key.

newDBKeyEntry :: (Int → a → Dynamic) → Int → a → Transaction ()

Stores a new entry in the database under a given key. The transaction fails if the key already exists.

cleanDB :: (Int → a → Dynamic) → Transaction ()

Deletes all entries from the database associated with a predicate.

closeDBHandles :: IO ()

Closes all database connections. Should be called when no more database access will be necessary.

showTError :: TError → String

Transforms a transaction error into a string.

A.2.17 Library List

Library with some useful operations on lists.

Exported functions:

`elemIndex :: a → [a] → Maybe Int`

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

`elemIndices :: a → [a] → [Int]`

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

`find :: (a → Bool) → [a] → Maybe a`

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

`findIndex :: (a → Bool) → [a] → Maybe Int`

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

`findIndices :: (a → Bool) → [a] → [Int]`

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

`nub :: [a] → [a]`

Removes all duplicates in the argument list.

`nubBy :: (a → a → Bool) → [a] → [a]`

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

`delete :: a → [a] → [a]`

Deletes the first occurrence of an element in a list.

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

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

`(\\) :: [a] → [a] → [a]`

Computes the difference of two lists.

`union :: [a] → [a] → [a]`

Computes the union of two lists.

`intersect :: [a] → [a] → [a]`

Computes the intersection of two lists.

`intersperse :: a → [a] → [a]`

Puts a separator element between all elements in a list.

Example: `(intersperse 9 [1,2,3,4]) = [1,9,2,9,3,9,4]`

`intercalate :: [a] → [[a]] → [a]`

`intercalate xs xss` is equivalent to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

`transpose :: [[a]] → [[a]]`

Transposes the rows and columns of the argument.

Example: `(transpose [[1,2,3],[4,5,6]]) = [[1,4],[2,5],[3,6]]`

`permutations :: [a] → [[a]]`

Returns the list of all permutations of the argument.

`partition :: (a → Bool) → [a] → ([a],[a])`

Partitions a list into a pair of lists where the first list contains those elements that satisfy the predicate argument and the second list contains the remaining arguments.

Example: `(partition (<4)></4)>`

`group :: [a] → [[a]]`

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

Example: `(group [1,2,2,3,3,3,4]) = [[1],[2,2],[3,3,3],[4]]`

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

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

`splitOn :: [a] → [a] → [[a]]`

Breaks a the second lsit argument into pieces separated by the first list argument, consuming the delimiter. An empty delimiter is invalid, and will cause an error to be raised.

`split :: (a → Bool) → [a] → [[a]]`

Splits a List into components delimited by separators, where the predicate returns True for a separator element. The resulting components do not contain the separators. Two adjacent separators result in an empty component in the output.

split (==a) ”aabbaca” == [””,””,”bb”,”c”,””] split (==a) ”” == [””]

`inits :: [a] → [[a]]`

Returns all initial segments of a list, starting with the shortest. Example: `inits [1,2,3] == [[],[1],[1,2],[1,2,3]]`

`tails :: [a] → [[a]]`

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

`replace :: a → Int → [a] → [a]`

Replaces an element in a list.

`isPrefixOf :: [a] → [a] → Bool`

Checks whether a list is a prefix of another.

`isSuffixOf :: [a] → [a] → Bool`

Checks whether a list is a suffix of another.

`isInfixOf :: [a] → [a] → Bool`

Checks whether a list is contained in another.

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

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

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

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

`last :: [a] → a`

Returns the last element of a non-empty list.

`init :: [a] → [a]`

Returns the input list with the last element removed.

`sum :: [Int] → Int`

Returns the sum of a list of integers.

`product :: [Int] → Int`

Returns the product of a list of integers.

`maximum :: [a] → a`

Returns the maximum of a non-empty list.

`minimum :: [a] → 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.2.18 Library Maybe

Library with some useful functions on the Maybe datatype

Exported functions:

`isJust :: Maybe a → Bool`

`isNothing :: Maybe a → Bool`

`fromJust :: Maybe a → a`

`fromMaybe :: a → Maybe a → a`

`maybeToList :: Maybe a → [a]`

`listToMaybe :: [a] → Maybe a`

`catMaybes :: [Maybe a] → [a]`

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

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

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

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

monadic sequence for maybe

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

monadic map for maybe

A.2.19 Library NamedSocket

Library to support network programming with sockets that are addressed by symbolic names. In contrast to raw sockets (see library `Socket`), this library uses the Curry Port Name Server to provide sockets that are addressed by symbolic names rather than numbers.
In standard applications, the server side uses the operations `listenOn` and `socketAccept` to provide some service on a named socket, and the client side uses the operation `connectToSocket` to request a service.

Exported types:

`data Socket`

Abstract type for named sockets.

Exported constructors:

Exported functions:

`listenOn :: String → IO Socket`

Creates a server side socket with a symbolic name.

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

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

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

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

`sClose :: Socket → IO ()`

Closes a server socket.

`socketName :: Socket → String`

Returns a the symbolic name of a named socket.

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

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

`connectToSocketWait :: String → IO Handle`

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

`connectToSocket :: String → IO Handle`

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

A.2.20 Library Parser

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

Exported types:

```
type Parser a = [a] → [a]
```

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

Exported functions:

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

Combines two parsers without representation in an alternative manner.

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

Combines two parsers with representation in an alternative manner.

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

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

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

Attaches a representation to a parser without representation.

```
empty :: [a] → [a]
```

The empty parser which recognizes the empty word.

```
terminal :: a → [a] → [a]
```

A parser recognizing a particular terminal symbol.

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

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

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

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

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

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

A.2.21 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 → Bool`

Is the document empty?

`text :: String → Doc`

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

`linesep :: String → Doc`

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

`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 `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.

`(<//>) :: Doc → Doc → Doc`

The document `(x <//> 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 `(<+>)`, `(<$>)` and so on.

`hsep :: [Doc] → Doc`

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

`vsep :: [Doc] → Doc`

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

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

This is layed out as:

```
some text
to
lay
out
```

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

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

This is printed as:

```
some text
     to
     lay
     out
```

`fillSep :: [Doc] → Doc`

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

`fillSep xs = foldr (</>) empty xs`

`sep :: [Doc] → 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 its fits the page, than 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`.)

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

The document (encloseSep l r sep xs) concatenates the documents xs seperated by 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 seperators are put in front of the elements.

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

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

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

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

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

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

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

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

The documents are rendered horizontally.

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

The document `(hEncloseSep l r sep xs)` concatenates the documents `xs` seperated by `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 seperators are put in front of the elements.

`list :: [Doc] → Doc`

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

`tupled :: [Doc] → Doc`

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

`semiBraces :: [Doc] → Doc`

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

`enclose :: Doc → Doc → Doc → Doc`

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

```
enclose l r x = l <> x <> r
```

`squotes :: Doc → Doc`

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

`dquotes :: Doc → Doc`

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

`bquotes :: Doc → Doc`

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

`parens :: Doc → Doc`

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

`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, `"="`.

`pretty :: Int → Doc → String`

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

A.2.22 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 executation, 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.2.23 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.2.24 Library Read

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

Exported functions:

`readNat :: String → Int`

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

`readInt :: String → Int`

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

`readHex :: String → Int`

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

A.2.25 Library ReadNumeric

Library with some functions for reading and converting numeric tokens.

Exported functions:

`readInt :: String → Maybe (Int,String)`

Read a (possibly negative) integer as a first token in a string. The string might contain leadings blanks and the integer is read up to the first non-digit. If the string does not start with an integer token, Nothing is returned, otherwise the result is (Just (v,s)) where v is the value of the integer and s is the remaing string without the integer token.

`readNat :: String → Maybe (Int,String)`

Read a natural number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-digit. If the string does not start with a natural number token, Nothing is returned, otherwise the result is (Just (v,s)) where v is the value of the number and s is the remaing string without the number token.

`readHex :: String → Maybe (Int,String)`

Read a hexadecimal number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-hexadecimal digit. If the string does not start with a hexadecimal number token, Nothing is returned, otherwise the result is (Just (v,s)) where v is the value of the number and s is the remaing string without the number token.

`readOct :: String → Maybe (Int,String)`

Read an octal number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-octal digit. If the string does not start with an octal number token, Nothing is returned, otherwise the result is (Just (v,s)) where v is the value of the number and s is the remaing string without the number token.

A.2.26 Library ReadShowTerm

Library for converting ground terms to strings and vice versa.

Exported functions:

`showTerm :: a → String`

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

`showQTerm :: a → String`

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

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

Transform a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The first argument is a non-empty list of module qualifiers that are tried to prefix the constructor in the string in order to get the qualified constructors (that must be defined in the current program!). In case of a successful parse, the result is a one element list containing a pair of the data term and the remaining unparsed string.

`readUnqualifiedTerm :: [String] → String → a`

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

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

`readsTerm :: String → [(a,String)]`

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

`readTerm :: String → a`

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

`readsQTerm :: String → [(a,String)]`

Transforms a string containing a term in standard prefix notation with qualified constructor names into the corresponding data term. In case of a successful parse, the result is a one element list containing a pair of the data term and the remaining unparsed string.

`readQTerm :: String → a`

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

`readQTermFile :: String → IO a`

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

`readQTermListFile :: String → IO [a]`

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

`writeQTermFile :: String → a → IO ()`

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

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

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

A.2.27 Library SetFunctions

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

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

Intuition: If `f` is an n-ary function, then `(setn f)` is a set-valued function that collects all non-determinism caused by f (but not the non-determinism caused by evaluating arguments!) in a set. Thus, `(setn f a1 ... an)` returns the set of all values of `(f b1 ... bn)` where `b1`,...,`bn` are values of the arguments `a1`,...,`an` (i.e., the arguments are evaluated "outside" this capsule so that the non-determinism caused by evaluating these arguments is not captured in this capsule but yields several results for `(setn...)`. Similarly, logical variables occuring in `a1`,...,`an` are not bound inside this capsule. The set of values returned by a set function is represented by an abstract type `Values` on which several operations are defined in this module. Actually, it is a multiset of values, i.e., duplicates are not removed.

Exported types:

`data Values`

Abstract type representing multisets of values.

Exported constructors:

Exported functions:

`set0 :: a → Values a`

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

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

`set1 :: (a → b) → a → Values b`

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

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

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

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

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

`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`

`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`

`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`

`set7 :: (a → b → c → d → e → f → g → h) → a → b → c → d → e → f → g → Values h`

Combinator to transform a function of arity 7 into a corresponding set function.

`set7With :: (SearchTree a → ValueSequence a) → (b → c → d → e → f → g → h → a) → b → c → d → e → f → g → h → Values a`

`isEmpty :: Values a → Bool`

Is a multiset of values 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.2.28 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`

`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 is defined, i.e., has a value.

`showSearchTree :: SearchTree a → String`

Shows the search tree as an intended line structure

`searchTreeSize :: SearchTree a → (Int,Int,Int)`

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

`allValuesDFS :: SearchTree a → [a]`

Return all values in a search tree via depth-first search

`dfsStrategy :: SearchTree a → ValueSequence a`

`allValuesBFS :: SearchTree a → [a]`

Return all values in a search tree via breadth-first search

`bfsStrategy :: SearchTree a → ValueSequence a`

`allValuesIDS :: SearchTree a → [a]`

Return all values in a search tree via iterative-deepening search.

`idsStrategy :: SearchTree a → ValueSequence a`

`allValuesIDSwith :: Int → (Int → Int) → SearchTree a → [a]`

Return the list of all values in a search tree via iterative-deepening search. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

`idsStrategyWith :: Int → (Int → Int) → SearchTree a → ValueSequence a`

Return all values in a search tree via iterative-deepening search. The first argument is the initial depth bound and the second argument is a function to increase the depth in each iteration.

`getAllValuesWith :: (SearchTree a → ValueSequence a) → a → IO [a]`

Gets all values of an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`. Conceptually, all values are computed on a copy of the expression, i.e., the evaluation of the expression does not share any results. Moreover, the evaluation suspends as long as the expression contains unbound variables.

`someValue :: a → a`

Returns some value for an expression.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

`someValueWith :: (SearchTree a → ValueSequence a) → a → a`

Returns some value for an expression w.r.t. a search strategy. A search strategy is an operation to traverse a search tree and collect all values, e.g., `dfsStrategy` or `bfsStrategy`.

Note that this operation is not purely declarative since the computed value depends on the ordering of the program rules. Thus, this operation should be used only if the expression has a single value. It fails if the expression has no value.

A.2.29 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.2.30 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 (always returns 0), only included for compatibility reasons.

`getArgs :: IO [String]`

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

`getEnviron :: String → IO String`

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

`setEnviron :: String → String → IO ()`

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

`unsetEnviron :: String → IO ()`

Removes an environment variable that has been set by `setEnviron`.

`getHostname :: IO String`

Returns the hostname of the machine running this process.

`getPID :: IO Int`

Returns the process identifier of the current Curry process.

`getProgName :: IO String`

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

`system :: String → IO Int`

Executes a shell command and return with the exit code of the command. An exit status of zero means successful execution.

`exitWith :: Int → IO a`

Terminates the execution of the current Curry program and returns the exit code given by the argument. An exit code of zero means successful execution.

`sleep :: Int → IO ()`

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

`isPosix :: Bool`

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

`isWindows :: Bool`

Is the underlying operating system a Windows system?

A.2.31 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 operationa is independent on the local time.

`toClockTime :: CalendarTime → ClockTime`

Transforms a calendar time (interpreted as UTC time) into a clock time.

`calendarTimeToString :: CalendarTime → String`

Transforms a calendar time into a readable form.

`toDayString :: CalendarTime → String`

Transforms a calendar time into a string containing the day, e.g., "September 23, 2006".

`toTimeString :: CalendarTime → String`

Transforms a calendar time into a string containing the time.

`addSeconds :: Int → ClockTime → ClockTime`

Adds seconds to a given time.

`addMinutes :: Int → ClockTime → ClockTime`

Adds minutes to a given time.

`addHours :: Int → ClockTime → ClockTime`

Adds hours to a given time.

`addDays :: Int → ClockTime → ClockTime`

Adds days to a given time.

`addMonths :: Int → ClockTime → ClockTime`

Adds months to a given time.

`addYears :: Int → ClockTime → ClockTime`

Adds years to a given time.

`daysOfMonth :: Int → Int → Int`

Gets the days of a month in a year.

`validDate :: Int → Int → Int → Bool`

Is a date consisting of year/month/day valid?

`compareDate :: CalendarTime → CalendarTime → Ordering`

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

`compareCalendarTime :: CalendarTime → CalendarTime → Ordering`

Compares two calendar times.

`compareClockTime :: ClockTime → ClockTime → Ordering`

Compares two clock times.

A.2.32 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.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.

`isEmpty :: Queue a → Bool`

Is the queue empty?

`deqHead :: Queue a → a`

The first element of the queue.

`deqLast :: Queue a → a`

The last element of the queue.

`cons :: a → Queue a → Queue a`

Inserts an element at the front of the queue.

`deqTail :: Queue a → Queue a`

Removes an element at the front of the queue.

`snoc :: a → Queue a → Queue a`

Inserts an element at the end 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.

`listToDeq :: [a] → Queue a`

Transforms a list to a double ended queue.

`deqToList :: Queue a → [a]`

Transforms a double ended queue to a list.

`deqLength :: Queue a → Int`

Returns the number of elements in the 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.

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`

Builts a finite map from given list of tuples (key,element). For multiple occurences of key, the last corresponding element of the list is taken.

`addToFM :: FM a b → a → b → FM a b`

Throws away any previous binding and stores the new one given.

`addListToFM :: FM a b → [(a,b)] → FM a b`

Throws away any previous bindings and stores the new ones given. The items are added starting with the first one in the list

`addToFM_C :: (a → a → a) → FM b a → b → a → FM b a`

Instead of throwing away the old binding, addToFM_C combines the new element with the old one.

`addListToFM_C :: (a → a → a) → FM b a → [(b,a)] → FM b a`

Combine with a list of tuples (key,element), cf. addToFM_C

`delFromFM :: FM a b → a → FM a b`

Deletes key from finite map. Deletion doesn't complain if you try to delete something which isn't there `delListFromFM :: FM a b → [a] → FM a b`

Deletes a list of keys from finite map. Deletion doesn't complain if you try to delete something which isn't there

`updFM :: FM a b → a → (b → b) → FM a b`

Applies a function to element bound to given key.

`splitFM :: FM a b → a → Maybe (FM a b,(a,b))`

Combines delFrom and lookup.

`plusFM :: FM a b → FM a b → FM a b`

Efficiently add key/element mappings of two maps into a single one. Bindings in right argument shadow those in the left

`plusFM_C :: (a → a → a) → FM b a → FM b a → FM b a`

Efficiently combine key/element mappings of two maps into a single one, cf. addToFM_C

`minusFM :: FM a b → FM a b → FM a b`

(minusFM a1 a2) deletes from a1 any bindings which are bound in a2

`intersectFM :: FM a b → FM a b → FM a b`

Filters only those keys that are bound in both of the given maps. The elements will be taken from the second map.

`intersectFM_C :: (a → a → b) → FM c a → FM c a → FM c b`

Filters only those keys that are bound in both of the given maps and combines the elements as in addToFM_C.

`foldFM :: (a → b → c → c) → c → FM a b → c`

Folds finite map by given function.

`mapFM :: (a → b → c) → FM a b → FM a c`

Applies a given function on every element in the map.

`filterFM :: (a → b → Bool) → FM a b → FM a b`

Yields a new finite map with only those key/element pairs matching the given predicate.

`sizeFM :: FM a b → Int`

How many elements does given map contain?

`eqFM :: FM a b → FM a b → Bool`

Do two given maps contain the same key/element pairs?

`isEmptyFM :: FM a b → Bool`

Is the given finite map empty?

`elemFM :: a → FM a b → Bool`

Does given map contain given key?

`lookupFM :: FM a b → a → Maybe b`

Retrieves element bound to given key

`lookupWithDefaultFM :: FM a b → b → a → b`

Retrieves element bound to given key. If the element is not contained in map, return default value.

`keyOrder :: FM a b → a → a → Bool`

Retrieves the ordering on which the given finite map is built.

`minFM :: FM a b → Maybe (a,b)`

Retrieves the smallest key/element pair in the finite map according to the basic key ordering.

`maxFM :: FM a b → Maybe (a,b)`

Retrieves the greatest key/element pair in the finite map according to the basic key ordering.

`fmToList :: FM a b → [(a,b)]`

Builds a list of key/element pairs. The list is ordered by the initially given irreflexive order predicate on keys.

`keysFM :: FM a b → [a]`

Retrieves a list of keys contained in finite map. The list is ordered by the initially given irreflexive order predicate on keys.

`eltsFM :: FM a b → [b]`

Retrieves a list of elements contained in finite map. The list is ordered by the initially given irreflexive order predicate on keys.

`fmToListPreOrder :: FM a b → [(a,b)]`

Retrieves list of key/element pairs in preorder of the internal tree. Useful for lists that will be retransformed into a tree or to match any elements regardless of basic order.

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

Sorts a given list by inserting and retrieving from finite map. Duplicates are deleted.

`showFM :: FM a b → String`

Transforms a finite map into a string. For efficiency reasons, the tree structure is shown which is valid for reading only if one uses the same ordering predicate.

`readFM :: (a → a → Bool) → String → FM a b`

Transforms a string representation of a finite map into a finite map. One has two provide the same ordering predicate as used in the original finite map.

A.3.4 Library GraphInductive

Library for inductive graphs (port of a Haskell library by Martin Erwig).
In this library, graphs are composed and decomposed in an inductive way.
The key idea is as follows:
A graph is either *empty* or it consists of *node context* and a *graph g'* which are put together by a constructor `(:&)`.
This constructor `(:&)`, however, is not a constructor in the sense of abstract data type, but more basically a defined constructing funtion.
A *context* is a node together withe the edges to and from this node into the nodes in the graph g'.
For examples of how to use this library, cf. the module `GraphAlgorithms`.

Exported types:

`type Node = Int`

Nodes and edges themselves (in contrast to their labels) are coded as integers.

For both of them, there are variants as labeled, unlabelwd and quasi unlabeled (labeled with ()).

Unlabeled node

`type LNode a = (Int,a)`

Labeled node

`type UNode = (Int,())`

Quasi-unlabeled node

`type Edge = (Int,Int)`

Unlabeled edge

`type LEdge a = (Int,Int,a)`

Labeled edge

`type UEdge = (Int,Int,())`

Quasi-unlabeled edge

`type Context a b = ([(b,Int)],Int,a,[(b,Int)])`

> The context of a node is the node itself (along with label) and its adjacent nodes. Thus, a context is a quadrupel, for node n it is of the form (edges to n,node n,n's label,edges from n)

`type MContext a b = Maybe ([(b,Int)],Int,a,[(b,Int)])`

> maybe context

`type Context' a b = ([(b,Int)],a,[(b,Int)])`

> context with edges and node label only, without the node identifier itself

`type UContext = ([Int],Int,[Int])`

> Unlabeled context.

`type GDecomp a b = (([(b,Int)],Int,a,[(b,Int)]),Graph a b)`

> A graph decompostion is a context for a node n and the remaining graph without that node.

`type Decomp a b = (Maybe ([(b,Int)],Int,a,[(b,Int)]),Graph a b)`

> a decomposition with a maybe context

`type UDecomp a = (Maybe ([Int],Int,[Int]),a)`

> Unlabeled decomposition.

`type Path = [Int]`

> Unlabeled path

`type LPath a = [(Int,a)]`

> Labeled path

`type UPath = [(Int,())]`

> Quasi-unlabeled path

`type UGr = Graph () ()`

> a graph without any labels

`data Graph`

> The type variables of Graph are *nodeLabel* and *edgeLabel*. The internal representation of Graph is hidden.

Exported constructors:

Exported functions:

`(:&) :: ([(a,Int)],Int,b,[(a,Int)]) → Graph b a → Graph b a`

> (:&) takes a node-context and a Graph and yields a new graph.
>
> The according key idea is detailed at the beginning.
>
> nl is the type of the node labels and el the edge labels.
>
> Note that it is an error to induce a context for a node already contained in the graph.

`matchAny :: Graph a b → (([(b,Int)],Int,a,[(b,Int)]),Graph a b)`

> decompose a graph into the `Context` for an arbitrarily-chosen `Node` and the remaining `Graph`.
>
> In order to use graphs as abstract data structures, we also need means to decompose a graph. This decompostion should work as much like pattern matching as possible. The normal matching is done by the function matchAny, which takes a graph and yields a graph decompostion.
>
> According to the main idea, matchAny . (:&) should be an identity.

`empty :: Graph a b`

> An empty `Graph`.

`mkGraph :: [(Int,a)] → [(Int,Int,b)] → Graph a b`

> Create a `Graph` from the list of `LNode`s and `LEdge`s.

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

> Build a `Graph` from a list of `Context`s.

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

> Build a quasi-unlabeled `Graph` from the list of `Node`s and `Edge`s.

`insNode :: (Int,a) → Graph a b → Graph a b`

> Insert a `LNode` into the `Graph`.

`insEdge :: (Int,Int,a) → Graph b a → Graph b a`

> Insert a `LEdge` into the `Graph`.

`delNode :: Int → Graph a b → Graph a b`

> Remove a `Node` from the `Graph`.

`delEdge :: (Int,Int) → Graph a b → Graph a b`

> Remove an `Edge` from the `Graph`.

`insNodes :: [(Int,a)] → Graph a b → Graph a b`

Insert multiple `LNode`s into the `Graph`.

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

Insert multiple `LEdge`s into the `Graph`.

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

Remove multiple `Node`s from the `Graph`.

`delEdges :: [(Int,Int)] → Graph a b → Graph a b`

Remove multiple `Edge`s from the `Graph`.

`isEmpty :: Graph a b → Bool`

test if the given `Graph` is empty.

`match :: Int → Graph a b → (Maybe ([(b,Int)],Int,a,[(b,Int)]),Graph a b)`

match is the complement side of (:&), decomposing a `Graph` into the `MContext` found for the given node and the remaining `Graph`.

`noNodes :: Graph a b → Int`

The number of `Node`s in a `Graph`.

`nodeRange :: Graph a b → (Int,Int)`

The minimum and maximum `Node` in a `Graph`.

`context :: Graph a b → Int → ([(b,Int)],Int,a,[(b,Int)])`

Find the context for the given `Node`. In contrast to "match", "context" causes an error if the `Node` is not present in the `Graph`.

`lab :: Graph a b → Int → Maybe a`

Find the label for a `Node`.

`neighbors :: Graph a b → Int → [Int]`

Find the neighbors for a `Node`.

`suc :: Graph a b → Int → [Int]`

Find all `Node`s that have a link from the given `Node`.

`pre :: Graph a b → Int → [Int]`

Find all `Node`s that link to to the given `Node`.

`lsuc :: Graph a b → Int → [(Int,b)]`

Find all Nodes and their labels, which are linked from the given `Node`.

`lpre :: Graph a b → Int → [(Int,b)]`

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

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

Find all outward-bound `LEdge`s for the given `Node`.

`inn :: Graph a b → Int → [(Int,Int,b)]`

Find all inward-bound `LEdge`s for the given `Node`.

`outdeg :: Graph a b → Int → Int`

The outward-bound degree of the `Node`.

`indeg :: Graph a b → Int → Int`

The inward-bound degree of the `Node`.

`deg :: Graph a b → Int → Int`

The degree of the `Node`.

`gelem :: Int → Graph a b → Bool`

`True` if the `Node` is present in the `Graph`.

`equal :: Graph a b → Graph a b → Bool`

graph equality

`node' :: ([(a,Int)],Int,b,[(a,Int)]) → Int`

The `Node` in a `Context`.

`lab' :: ([(a,Int)],Int,b,[(a,Int)]) → b`

The label in a `Context`.

`labNode' :: ([(a,Int)],Int,b,[(a,Int)]) → (Int,b)`

The `LNode` from a `Context`.

`neighbors' :: ([(a,Int)],Int,b,[(a,Int)]) → [Int]`

All `Node`s linked to or from in a `Context`.

`suc' :: ([(a,Int)],Int,b,[(a,Int)]) → [Int]`

All `Node`s linked to in a `Context`.

`pre' :: ([(a,Int)],Int,b,[(a,Int)]) → [Int]`

All `Node`s linked from in a `Context`.

`lpre' :: ([(a,Int)],Int,b,[(a,Int)]) → [(Int,a)]`

All `Node`s linked from in a `Context`, and the label of the links.

`lsuc' :: ([(a,Int)],Int,b,[(a,Int)]) → [(Int,a)]`

All `Node`s linked from in a `Context`, and the label of the links.

`out' :: ([(a,Int)],Int,b,[(a,Int)]) → [(Int,Int,a)]`

All outward-directed `LEdge`s in a `Context`.

`inn' :: ([(a,Int)],Int,b,[(a,Int)]) → [(Int,Int,a)]`

All inward-directed `LEdge`s in a `Context`.

`outdeg' :: ([(a,Int)],Int,b,[(a,Int)]) → Int`

The outward degree of a `Context`.

`indeg' :: ([(a,Int)],Int,b,[(a,Int)]) → Int`

The inward degree of a `Context`.

`deg' :: ([(a,Int)],Int,b,[(a,Int)]) → Int`

The degree of a `Context`.

`labNodes :: Graph a b → [(Int,a)]`

A list of all `LNode`s in the `Graph`.

`labEdges :: Graph a b → [(Int,Int,b)]`

A list of all `LEdge`s in the `Graph`.

`nodes :: Graph a b → [Int]`

List all `Node`s in the `Graph`.

`edges :: Graph a b → [(Int,Int)]`

List all `Edge`s in the `Graph`.

`newNodes :: Int → Graph a b → [Int]`

List N available `Node`s, ie `Node`s that are not used in the `Graph`.

`ufold :: (([(a,Int)],Int,b,[(a,Int)]) → c → c) → c → Graph b a → c`

Fold a function over the graph.

`gmap :: (([(a,Int)],Int,b,[(a,Int)]) → ([(c,Int)],Int,d,[(c,Int)])) → Graph b a → Graph d c`

Map a function over the graph.

`nmap :: (a → b) → Graph a c → Graph b c`

Map a function over the `Node` labels in a graph.

`emap :: (a → b) → Graph c a → Graph c b`

Map a function over the `Edge` labels in a graph.

`labUEdges :: [(a,b)] → [(a,b,())]`

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

`labUNodes :: [a] → [(a,())]`

add label () to list of nodes

`showGraph :: Graph a b → String`

Represent Graph as String

A.3.5 Library Random

Library for pseudo-random number generation in Curry.
This library provides operations for generating pseudo-random number sequences. For any given seed, the sequences generated by the operations in this module should be **identical** to the sequences generated by the `java.util.Random package`.
The algorithm is taken from http://en.wikipedia.org/wiki/Random_number_generation. There is an assumption that all operations are implicitly executed mod 2^32 (unsigned 32-bit integers) !!! GHC computes between -2^29 and 2^29-1, thus the sequence is NOT as random as one would like.

```
m_w = <choose-initializer>;    /* must not be zero */
m_z = <choose-initializer>;    /* must not be zero */

uint get_random()
{
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w;  /* 32-bit result */
}
```

Exported functions:

`nextInt :: Int → [Int]`

Returns a sequence of pseudorandom, integer values.

`nextIntRange :: Int → Int → [Int]`

Returns a pseudorandom sequence of values between 0 (inclusive) and the specified value (exclusive).

`nextBoolean :: Int → [Bool]`

Returns a pseudorandom sequence of boolean values.

`getRandomSeed :: IO Int`

Returns a time-dependent integer number as a seed for really random numbers. Should only be used as a seed for pseudorandom number sequence and not as a random number since the precision is limited to milliseconds

A.3.6 Library RedBlackTree

Library with an implementation of red-black trees:
Serves as the base for both TableRBT and SetRBT All the operations on trees are generic, i.e., one has to provide two explicit order predicates ("`lessThan`" and "`eq`"below) on elements.

Exported types:

`data RedBlackTree`

A red-black tree consists of a tree structure and three order predicates. These predicates generalize the red black tree. They define 1) equality when inserting into the tree

eg for a set eqInsert is (==), for a multiset it is (-> False) for a lookUp-table it is ((==) . fst) 2) equality for looking up values eg for a set eqLookUp is (==), for a multiset it is (==) for a lookUp-table it is ((==) . fst) 3) the (less than) relation for the binary search tree

Exported constructors:

Exported functions:

`empty :: (a → a → Bool) → (a → a → Bool) → (a → a → Bool) → RedBlackTree a`

The three relations are inserted into the structure by function empty. Returns an empty tree, i.e., an empty red-black tree augmented with the order predicates.

`isEmpty :: RedBlackTree a → Bool`

Test on emptyness

`newTreeLike :: RedBlackTree a → RedBlackTree a`

Creates a new empty red black tree from with the same ordering as a give one.

`lookup :: a → RedBlackTree a → Maybe a`

Returns an element if it is contained in a red-black tree.

`update :: a → RedBlackTree a → RedBlackTree a`

Updates/inserts an element into a RedBlackTree.

`delete :: a → RedBlackTree a → RedBlackTree a`

Deletes entry from red black tree.

`tree2list :: RedBlackTree a → [a]`

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

`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 ("`cmp`" below) on elements.

Exported types:

`type SetRBT a = RedBlackTree a`

Exported functions:

`emptySetRBT :: (a → a → Bool) → RedBlackTree a`

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

`isEmptySetRBT :: RedBlackTree a → Bool`

Test for an empty set.

`elemRBT :: a → RedBlackTree a → Bool`

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

`insertRBT :: a → RedBlackTree a → RedBlackTree a`

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

`insertMultiRBT :: a → RedBlackTree a → RedBlackTree a`

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

`deleteRBT :: a → RedBlackTree a → RedBlackTree a`

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

`setRBT2list :: RedBlackTree a → [a]`

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

`unionRBT :: RedBlackTree a → RedBlackTree a → RedBlackTree a`

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

`intersectRBT :: RedBlackTree a → RedBlackTree a → RedBlackTree a`

Computes the intersection of two (red-black tree) sets. This is done by inserting all elements of the first set contained in the second set into a new set, which order is taken from the first set.

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

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

A.3.8 Library Sort

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

Exported functions:

`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>).

`cmpChar :: Char → Char → Ordering`

Comparison of characters (deprecated, use `Prelude.compare`).

`leqCharIgnoreCase :: Char → Char → Bool`

Less-or-equal on characters ignoring case considerations.

`leqString :: String → String → Bool`

Less-or-equal on strings (deprecated, use Prelude.<=</code></=</code>).

`cmpString :: String → String → Ordering`

Comparison of strings (deprecated, use `Prelude.compare`).

`leqStringIgnoreCase :: String → String → Bool`

Less-or-equal on strings ignoring case considerations.

`leqLexGerman :: String → String → Bool`

Lexicographical ordering on German strings. Thus, upper/lowercase are not distinguished and Umlauts are sorted as vocals.

A.3.9 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.

Exported functions:

`noChildren :: a → ([b],[b] → a)`

Traversal function for constructors without children.

`children :: (a → ([b],[b] → a)) → a → [b]`

Yields the children of a value.

`replaceChildren :: (a → ([b],[b] → a)) → a → [b] → a`

Replaces the children of a value.

`mapChildren :: (a → ([b],[b] → a)) → (b → b) → a → a`

Applies the given function to each child of a value.

`family :: (a → ([a],[a] → a)) → a → [a]`

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

`childFamilies :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → a → [b]`

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

`mapFamily :: (a → ([a],[a] → a)) → (a → a) → a → a`

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

`mapChildFamilies :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (b → b) → a → a`

Applies the given function to each member of the families of the children of a value. The value and its children can have different types. Proceeds bottom-up.

`evalFamily :: (a → ([a],[a] → a)) → (a → Maybe a) → a → a`

[8] `http://www-ps.informatik.uni-kiel.de/~sebf/projects/traversal.html`

Applies the given function to each member of the family of a value as long as possible. On each member of the family of the result the given function will yield `Nothing`. Proceeds bottom-up.

`evalChildFamilies :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (b → Maybe b) → a → a`

Applies the given function to each member of the families of the children of a value as long as possible. Similar to `evalFamily`.

`fold :: (a → ([a],[a] → a)) → (a → [b] → b) → a → b`

Implements a traversal similar to a fold with possible default cases.

`foldChildren :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (a → [c] → d) → (b → [c] → c) → a → d`

Fold the children and combine the results.

`replaceChildrenIO :: (a → ([b],[b] → a)) → a → IO [b] → IO a`

IO version of replaceChildren

`mapChildrenIO :: (a → ([b],[b] → a)) → (b → IO b) → a → IO a`

IO version of mapChildren

`mapFamilyIO :: (a → ([a],[a] → a)) → (a → IO a) → a → IO a`

IO version of mapFamily

`mapChildFamiliesIO :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (b → IO b) → a → IO a`

IO version of mapChildFamilies

`evalFamilyIO :: (a → ([a],[a] → a)) → (a → IO (Maybe a)) → a → IO a`

IO version of evalFamily

`evalChildFamiliesIO :: (a → ([b],[b] → a)) → (b → ([b],[b] → b)) → (b → IO (Maybe b)) → a → IO a`

IO version of evalChildFamilies

A.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 inial keys.

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

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

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

A.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

```
makecurrycgi -m initialForm -o /home/joe/public_html/prog.cgi prog
```

where `prog` is the name of the Curry program with the cgi script, `/home/joe/public_html/prog.cgi` is the desired location of the compiled cgi script, and `initialForm` is the Curry expression (of type IO HtmlForm) computing the HTML form (where makecurrycgi is a shell script stored in *pakcshome*/bin).

Exported types:

`type CgiEnv = CgiRef → String`

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

`type HtmlHandler = (CgiRef → String) → IO HtmlForm`

The type of event handlers in HTML forms.

`data CgiRef`

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

Exported constructors:

`data HtmlExp`

The data type for representing HTML expressions.

Exported constructors:

- `HtmlText :: String → HtmlExp`

  `HtmlText s`

  – a text string without any further structure

- `HtmlStruct :: String → [(String,String)] → [HtmlExp] → HtmlExp`

  `HtmlStruct t as hs`

  – a structure with a tag, attributes, and HTML expressions inside the structure

- `HtmlCRef :: HtmlExp → CgiRef → HtmlExp`

  `HtmlCRef h ref`

  – an input element (described by the first argument) with a cgi reference

- `HtmlEvent :: HtmlExp → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

  `HtmlEvent h hdlr`

  – an input element (first arg) with an associated event handler (tpyically, a submit button)

`data HtmlForm`

The data type for representing HTML forms (active web pages) and return values of HTML forms.

Exported constructors:

- `HtmlForm :: String → [FormParam] → [HtmlExp] → HtmlForm`

  `HtmlForm t ps hs`

  – an HTML form with title t, optional parameters (e.g., cookies) ps, and contents hs

- `HtmlAnswer :: String → String → HtmlForm`

  `HtmlAnswer t c`

  – an answer in an arbitrary format where t is the content type (e.g., "text/plain") and c is the contents

`data FormParam`

The possible parameters of an HTML form. The parameters of a cookie (FormCookie) are its name and value and optional parameters (expiration date, domain, path (e.g., the path "/" makes the cookie valid for all documents on the server), security) which are collected in a list.

Exported constructors:

- `FormCookie :: String → String → [CookieParam] → FormParam`

  `FormCookie name value params`

  – a cookie to be sent to the client's browser

- `FormCSS :: String → FormParam`

  `FormCSS s`

  – a URL for a CSS file for this form

- `FormJScript :: String → FormParam`

  `FormJScript s`

  – a URL for a Javascript file for this form

- `FormOnSubmit :: String → FormParam`

  `FormOnSubmit s`

  – a JavaScript statement to be executed when the form is submitted (i.e., <form ... onsubmit="s">)

- `FormTarget :: String → FormParam`

  `FormTarget s`

  – a name of a target frame where the output of the script should be represented (should only be used for scripts running in a frame)

- `FormEnc :: String → FormParam`

  `FormEnc`

  – the encoding scheme of this form

- `HeadInclude :: HtmlExp → FormParam`

  `HeadInclude he`

  – HTML expression to be included in form header

- `MultipleHandlers :: FormParam`

  `MultipleHandlers`

  – indicates that the event handlers of the form can be multiply used (i.e., are not deleted if the form is submitted so that they are still available when going back in the browser; but then there is a higher risk that the web server process might overflow with unused events); the default is a single use of event handlers, i.e., one cannot use the back button in the browser and submit the same form again (which is usually a reasonable behavior to avoid double submissions of data).

- `BodyAttr :: (String,String) → FormParam`

  `BodyAttr ps`

  – optional attribute for the body element (more than one occurrence is allowed)

`data CookieParam`

The possible parameters of a cookie.

Exported constructors:

- `CookieExpire :: ClockTime → CookieParam`
- `CookieDomain :: String → CookieParam`
- `CookiePath :: String → CookieParam`
- `CookieSecure :: CookieParam`

`data HtmlPage`

The data type for representing HTML pages. The constructor arguments are the title, the parameters, and the contents (body) of the web page.

Exported constructors:

- `HtmlPage :: String → [PageParam] → [HtmlExp] → HtmlPage`

`data PageParam`

The possible parameters of an HTML page.

Exported constructors:

- `PageEnc :: String → PageParam`

  `PageEnc`

  – the encoding scheme of this page

- `PageCSS :: String → PageParam`

  `PageCSS s`

  – a URL for a CSS file for this page

- `PageJScript :: String → PageParam`

  `PageJScript s`

  – a URL for a Javascript file for this page

- `PageMeta :: [(String,String)] → PageParam`

  `PageMeta as`

– meta information (in form of attributes) for this page

- `PageLink :: [(String,String)] → PageParam`

  `PageLink as`

  – link information (in form of attributes) for this page

- `PageBodyAttr :: (String,String) → PageParam`

  `PageBodyAttr attr`

  – optional attribute for the body element of the page (more than one occurrence is allowed)

Exported functions:

`defaultEncoding :: String`

The default encoding used in generated web pages.

`idOfCgiRef :: CgiRef → String`

Internal identifier of a CgiRef (intended only for internal use in other libraries!).

`formEnc :: String → FormParam`

An encoding scheme for a HTML form.

`formCSS :: String → FormParam`

A URL for a CSS file for a HTML form.

`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 anchor for hypertext reference inside a document

`ulist :: [[HtmlExp]] → HtmlExp`

Unordered list

`olist :: [[HtmlExp]] → HtmlExp`

Ordered list

`litem :: [HtmlExp] → HtmlExp`

A single list item (usually not explicitly used)

`dlist :: [([HtmlExp],[HtmlExp])] → HtmlExp`

Description list

`table :: [[[HtmlExp]]] → HtmlExp`

Table with a matrix of items where each item is a list of HTML expressions.

`headedTable :: [[[HtmlExp]]] → HtmlExp`

Similar to `table` but introduces header tags for the first row.

`addHeadings :: HtmlExp → [[HtmlExp]] → HtmlExp`

Add a row of items (where each item is a list of HTML expressions) as headings to a table. If the first argument is not a table, the headings are ignored.

`hrule :: HtmlExp`

Horizontal rule

`breakline :: HtmlExp`

Break a line

`image :: String → String → HtmlExp`

Image

`styleSheet :: String → HtmlExp`

Defines a style sheet to be used in this HTML document.

`style :: String → [HtmlExp] → HtmlExp`

Provides a style for HTML elements. The style argument is the name of a style class defined in a style definition (see `styleSheet`) or in an external style sheet (see form and page parameters `FormCSS` and `PageCSS`).

`textstyle :: String → String → HtmlExp`

Provides a style for a basic text. The style argument is the name of a style class defined in an external style sheet.

`blockstyle :: String → [HtmlExp] → HtmlExp`

Provides a style for a block of HTML elements. The style argument is the name of a style class defined in an external style sheet. This element is used (in contrast to "style") for larger blocks of HTML elements since a line break is placed before and after these elements.

`inline :: [HtmlExp] → HtmlExp`

Joins a list of HTML elements into a single HTML element. Although this construction has no rendering, it is sometimes useful for programming when several HTML elements must be put together.

`block :: [HtmlExp] → HtmlExp`

Joins a list of HTML elements into a block. A line break is placed before and after these elements.

`button :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

Submit button with a label string and an event handler

`resetbutton :: String → HtmlExp`

Reset button with a label string

`imageButton :: String → ((CgiRef → String) → IO HtmlForm) → HtmlExp`

Submit button in form of an imag.

`textfield :: CgiRef → String → HtmlExp`

Input text field with a reference and an initial contents

`password :: CgiRef → HtmlExp`

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

`textarea :: CgiRef → (Int,Int) → String → HtmlExp`

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

`checkbox :: CgiRef → String → HtmlExp`

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

`checkedbox :: CgiRef → String → HtmlExp`

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

`radio_main :: CgiRef → String → HtmlExp`

A main button of a radio (initially "on") with a reference and a value. The value is returned of this button is on. A complete radio button suite always consists of a main button (radio*main) and some further buttons (radio*others) with the same reference. Initially, the main button is selected (or nothing is selected if one uses radio*main*off instead of radio_main). The user can select another button but always at most one button of the radio can be selected. The value corresponding to the selected button is returned in the environment for this radio reference.

`radio_main_off :: CgiRef → String → HtmlExp`

A main button of a radio (initially "off") with a reference and a value. The value is returned of this button is on.

`radio_other :: CgiRef → String → HtmlExp`

A further button of a radio (initially "off") with a reference (identical to the main button of this radio) and a value. The value is returned of this button is on.

`selection :: CgiRef → [(String,String)] → HtmlExp`

A selection button with a reference and a list of name/value pairs. The names are shown in the selection and the value is returned for the selected name.

`selectionInitial :: CgiRef → [(String,String)] → Int → HtmlExp`

A selection button with a reference, a list of name/value pairs, and a preselected item in this list. The names are shown in the selection and the value is returned for the selected name.

`multipleSelection :: CgiRef → [(String,String,Bool)] → HtmlExp`

A selection button with a reference and a list of name/value/flag pairs. The names are shown in the selection and the value is returned if the corresponding name is selected. If flag is True, the corresonding name is initially selected. If more than one name has been selected, all values are returned in one string where the values are separated by newline (`\n`) characters.

`hiddenfield :: String → String → HtmlExp`

A hidden field to pass a value referenced by a fixed name. This function should be used with care since it may cause conflicts with the CGI-based implementation of this library.

`htmlQuote :: String → String`

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

`htmlIsoUmlauts :: String → String`

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

`addAttr :: HtmlExp → (String,String) → HtmlExp`

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

`addAttrs :: HtmlExp → [(String,String)] → HtmlExp`

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

`addClass :: HtmlExp → String → HtmlExp`

Adds a class attribute to an HTML element.

`showHtmlExps :: [HtmlExp] → String`

Transforms a list of HTML expressions into string representation.

`showHtmlExp :: HtmlExp → String`

Transforms a single HTML expression into string representation.

`showHtmlPage :: HtmlPage → String`

Transforms HTML page into string representation.

`getUrlParameter :: IO String`

Gets the parameter attached to the URL of the script. For instance, if the script is called with URL "http://.../script.cgi?parameter", then "parameter" is returned by this I/O action. Note that an URL parameter should be "URL encoded" to avoid the appearance of characters with a special meaning. Use the functions "urlencoded2string" and "string2urlencoded" to decode and encode such parameters, respectively.

`urlencoded2string :: String → String`

Translates urlencoded string into equivalent ASCII string.

`string2urlencoded :: String → String`

Translates arbitrary strings into equivalent urlencoded string.

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

Gets the cookies sent from the browser for the current CGI script. The cookies are represented in the form of name/value pairs since no other components are important here.

`coordinates :: (CgiRef → String) → Maybe (Int,Int)`

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

`runFormServerWithKey :: String → String → IO HtmlForm → IO ()`

The server implementing an HTML form (possibly containing input fields). It receives a message containing the environment of the client's web browser, translates the HTML form w.r.t. this environment into a string representation of the complete HTML document and sends the string representation back to the client's browser by binding the corresponding message argument.

`runFormServerWithKeyAndFormParams :: String → String → [FormParam] → IO HtmlForm → IO ()`

The server implementing an HTML form (possibly containing input fields). It receives a message containing the environment of the client's web browser, translates the HTML form w.r.t. this environment into a string representation of the complete HTML document and sends the string representation back to the client's browser by binding the corresponding message argument.

`showLatexExps :: [HtmlExp] → String`

Transforms HTML expressions into LaTeX string representation.

`showLatexExp :: HtmlExp → String`

Transforms an HTML expression into LaTeX string representation.

`htmlSpecialChars2tex :: String → String`

Convert special HTML characters into their LaTeX representation, if necessary.

`showLatexDoc :: [HtmlExp] → String`

Transforms HTML expressions into a string representation of a complete LaTeX document.

`showLatexDocWithPackages :: [HtmlExp] → [String] → String`

Transforms HTML expressions into a string representation of a complete LaTeX document. The variable "packages" holds the packages to add to the latex document e.g. "ngerman"

`showLatexDocs :: [[HtmlExp]] → String`

Transforms a list of HTML expressions into a string representation of a complete LaTeX document where each list entry appears on a separate page.

`showLatexDocsWithPackages :: [[HtmlExp]] → [String] → String`

Transforms a list of HTML expressions into a string representation of a complete LaTeX document where each list entry appears on a separate page. The variable "packages" holds the packages to add to the latex document (e.g., "ngerman").

`germanLatexDoc :: [HtmlExp] → String`

show german latex document

`intForm :: IO HtmlForm → IO ()`

Execute an HTML form in "interactive" mode.

`intFormMain :: String → String → String → String → Bool → String → IO HtmlForm → IO ()`

Execute an HTML form in "interactive" mode with various parameters.

A.4.3 Library HtmlParser

This module contains a very simple parser for HTML documents.

Exported functions:

`readHtmlFile :: String → IO [HtmlExp]`

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

`parseHtmlString :: String → [HtmlExp]`

Transforms an HTML string into a list of HTML expressions. If the HTML string is a well structured document, the list of HTML expressions should contain exactly one element.

A.4.4 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.4.5 Library Markdown

Library to translate markdown documents into HTML or LaTeX. The slightly restricted subset of the markdown syntax recognized by this implementation is documented in this page.

Exported types:

`type MarkdownDoc = [MarkdownElem]`

A markdown document is a list of markdown elements.

`data MarkdownElem`

The data type for representing the different elements occurring in a markdown document.

Exported constructors:

- `Text :: String → MarkdownElem`

  `Text s`

  – a simple text in a markdown document

- `Emph :: String → MarkdownElem`

  `Emph s`

  – an emphasized text in a markdown document

- `Strong :: String → MarkdownElem`

  `Strong s`

  – a strongly emphaszed text in a markdown document

- `Code :: String → MarkdownElem`

  `Code s`

  – a code string in a markdown document

- `HRef :: String → String → MarkdownElem`

  `HRef s u`

  – a reference to URL `u` with text `s` in a markdown document

- `Par :: [MarkdownElem] → MarkdownElem`

  `Par md`

  – a paragraph in a markdown document

- `CodeBlock :: String → MarkdownElem`

  `CodeBlock s`

  – a code block in a markdown document

- `UList :: [[MarkdownElem]] → MarkdownElem`

  `UList mds`

  – an unordered list in a markdown document

- `OList :: [[MarkdownElem]] → MarkdownElem`

  `OList mds`

  – an ordered list in a markdown document

- `Quote :: [MarkdownElem] → MarkdownElem`

  `Quote md`

  – a quoted paragraph in a markdown document

- `HRule :: MarkdownElem`

  `HRule`

  – a hoirzontal rule in a markdown document

- `Header :: Int → String → MarkdownElem`

  `Header l s`

  – a level `l` header with title `s` in a markdown document

Exported functions:

`fromMarkdownText :: String → [MarkdownElem]`

Parse markdown document from its textual representation.

`removeEscapes :: String → String`

Remove the backlash of escaped markdown characters in a string.

`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.4.6 Library WUI

A library to support the type-oriented construction of Web User Interfaces (WUIs). In contrast to the original WUI library, this library does not use functional patterns and, thus, has a different interface.
The ideas behind the application and implementation of WUIs are described in a paper that is available via this web page.

Exported types:

`type Rendering = [HtmlExp] → HtmlExp`

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

`data WuiHandler`

A handler for a WUI is an event handler for HTML forms possibly with some specific code attached (for future extensions).

Exported constructors:

`data WuiSpec`

The type of WUI specifications. The first component are parameters specifying the behavior of this WUI type (rendering, error message, and constraints on inputs). The second component is a "show" function returning an HTML expression for the edit fields and a WUI state containing the CgiRefs to extract the values from the edit fields. The third component is "read" function to extract the values from the edit fields for a given cgi environment (returned as (Just v)). If the value is not legal, Nothing is returned. The second component of the result contains an HTML edit expression together with a WUI state to edit the value again.

Exported constructors:

`data WTree`

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

Exported constructors:

- `WLeaf :: a → WTree a`
- `WNode :: [WTree a] → WTree a`

Exported functions:

`wuiHandler2button :: String → WuiHandler → HtmlExp`

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

`withRendering :: WuiSpec a → ([HtmlExp] → HtmlExp) → WuiSpec a`

Puts a new rendering function into a WUI specification.

`withError :: WuiSpec a → String → WuiSpec a`

Puts a new error message into a WUI specification.

`withCondition :: WuiSpec a → (a → Bool) → WuiSpec a`

Puts a new condition into a WUI specification.

`transformWSpec :: (a → b,b → a) → WuiSpec a → WuiSpec b`

Transforms a WUI specification from one type to another.

`wHidden :: WuiSpec a`

A hidden widget for a value that is not shown in the WUI. Usually, this is used in components of larger structures, e.g., internal identifiers, data base keys.

`wConstant :: (a → HtmlExp) → WuiSpec a`

A widget for values that are shown but cannot be modified. The first argument is a mapping of the value into a HTML expression to show this value.

`wInt :: WuiSpec Int`

A widget for editing integer values.

`wString :: WuiSpec String`

A widget for editing string values.

`wStringSize :: Int → WuiSpec String`

A widget for editing string values with a size attribute.

`wRequiredString :: WuiSpec String`

A widget for editing string values that are required to be non-empty.

`wRequiredStringSize :: Int → WuiSpec String`

A widget with a size attribute for editing string values that are required to be non-empty.

`wTextArea :: (Int,Int) → WuiSpec String`

A widget for editing string values in a text area. The argument specifies the height and width of the text area.

`wSelect :: (a → String) → [a] → WuiSpec a`

A widget to select a value from a given list of values. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into strings to be shown in the selection widget.

`wSelectInt :: [Int] → WuiSpec Int`

A widget to select a value from a given list of integers (provided as the argument). The current value should be contained in the value list and is preselected.

`wSelectBool :: String → String → WuiSpec Bool`

A widget to select a Boolean value via a selection box. The arguments are the strings that are shown for the values True and False in the selection box, respectively.

`wCheckBool :: [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a check box. The first argument are HTML expressions that are shown after the check box. The result is True if the box is checked.

`wMultiCheckSelect :: (a → [HtmlExp]) → [a] → WuiSpec [a]`

A widget to select a list of values from a given list of values via check boxes. The current values should be contained in the value list and are preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the check box.

`wRadioSelect :: (a → [HtmlExp]) → [a] → WuiSpec a`

A widget to select a value from a given list of values via a radio button. The current value should be contained in the value list and is preselected. The first argument is a mapping from values into HTML expressions that are shown for each item after the radio button.

`wRadioBool :: [HtmlExp] → [HtmlExp] → WuiSpec Bool`

A widget to select a Boolean value via a radio button. The arguments are the lists of HTML expressions that are shown after the True and False radio buttons, respectively.

`wPair :: WuiSpec a → WuiSpec b → WuiSpec (a,b)`

WUI combinator for pairs.

`wTriple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec (a,b,c)`

WUI combinator for triples.

`w4Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec (a,b,c,d)`

WUI combinator for tuples of arity 4.

`w5Tuple :: WuiSpec a → WuiSpec b → WuiSpec c → WuiSpec d → WuiSpec e → WuiSpec (a,b,c,d,e)`

WUI combinator for tuples of arity 5.

`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.

`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.

`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.

`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.

`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.

`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.

`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.

`wJoinTuple :: WuiSpec a → WuiSpec b → WuiSpec (a,b)`

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

`wList :: WuiSpec a → WuiSpec [a]`

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

`wListWithHeadings :: [String] → WuiSpec a → WuiSpec [a]`

Add headings to a standard WUI for list structures:

`wHList :: WuiSpec a → WuiSpec [a]`

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

`wMatrix :: WuiSpec a → WuiSpec [[a]]`

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

`wMaybe :: WuiSpec Bool → WuiSpec a → a → WuiSpec (Maybe a)`

WUI for Maybe values. It is constructed from a WUI for Booleans and a WUI for the potential values. Nothing corresponds to a selection of False in the Boolean WUI. The value WUI is shown after the Boolean WUI.

`wCheckMaybe :: WuiSpec a → [HtmlExp] → a → WuiSpec (Maybe a)`

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

`wRadioMaybe :: WuiSpec a → [HtmlExp] → [HtmlExp] → a → WuiSpec (Maybe a)`

A WUI for Maybe values where radio buttons are used to switch between Nothing and Just. The value WUI is shown after the radio button WUI.

`wEither :: WuiSpec a → WuiSpec b → WuiSpec (Either a b)`

WUI for union types. Here we provide only the implementation for Either types since other types with more alternatives can be easily reduced to this case.

`wTree :: WuiSpec a → WuiSpec (WTree a)`

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

`renderTuple :: [HtmlExp] → HtmlExp`

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

`renderTaggedTuple :: [String] → [HtmlExp] → HtmlExp`

Standard rendering of tuples with a tag for each element. Thus, each is preceded by a tag, that is set in bold, and all elements are vertically aligned.

`renderList :: [HtmlExp] → HtmlExp`

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

`mainWUI :: WuiSpec a → a → (a → IO HtmlForm) → IO HtmlForm`

Generates an HTML form from a WUI data specification, an initial value and an update form.

`wui2html :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp,WuiHandler)`

Generates HTML editors and a handler from a WUI data specification, an initial value and an update form.

`wuiInForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO HtmlForm) → IO HtmlForm`

Puts a WUI into a HTML form containing "holes" for the WUI and the handler.

`wuiWithErrorForm :: WuiSpec a → a → (a → IO HtmlForm) → (HtmlExp → WuiHandler → IO HtmlForm) → (HtmlExp,WuiHandler)`

Generates HTML editors and a handler from a WUI data specification, an initial value and an update form. In addition to wui2html, we can provide a skeleton form used to show illegal inputs.

A.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 XML

Library for processing XML data.
Warning: the structure of this library is not stable and might be changed in the future!

Exported types:

`data XmlExp`

The data type for representing XML expressions.

Exported constructors:

- `XText :: String → XmlExp`

  `XText`

  – a text string (PCDATA)

- `XElem :: String → [(String,String)] → [XmlExp] → XmlExp`

  `XElem`

  – an XML element with tag field, attributes, and a list of XML elements as contents

`data Encoding`

The data type for encodings used in the XML document.

Exported constructors:

- `StandardEnc :: Encoding`
- `Iso88591Enc :: Encoding`

`data XmlDocParams`

The data type for XML document parameters.

Exported constructors:

- `Enc :: Encoding` → `XmlDocParams`

  `Enc`

  – the encoding for a document
- `DtdUrl :: String` → `XmlDocParams`

  `DtdUrl`

  – the url of the DTD for a document

Exported functions:

`tagOf :: XmlExp` → `String`

Returns the tag of an XML element (or empty for a textual element).

`elemsOf :: XmlExp` → `[XmlExp]`

Returns the child elements an XML element.

`textOf :: [XmlExp]` → `String`

Extracts the textual contents of a list of XML expressions. Useful auxiliary function when transforming XML expressions into other data structures.

For instance, textOf [XText ”xy”, XElem ”a” [] `[], XText "bc"] == "xy bc"`

`textOfXml :: [XmlExp]` → `String`

Included for backward compatibility, better use `textOf`!

`xtxt :: String` → `XmlExp`

Basic text (maybe containing special XML chars).

`xml :: String` → `[XmlExp]` → `XmlExp`

XML element without attributes.

`writeXmlFile :: String` → `XmlExp` → `IO ()`

Writes a file with a given XML document.

`writeXmlFileWithParams :: String` → `[XmlDocParams]` → `XmlExp` → `IO ()`

Writes a file with a given XML document and XML parameters.

`showXmlDoc :: XmlExp` → `String`

Show an XML document in indented format as a string.

`showXmlDocWithParams :: [XmlDocParams]` → `XmlExp` → `String` `readXmlFile :: String → IO XmlExp`

Reads a file with an XML document and returns the corresponding XML expression.

`readUnsafeXmlFile :: String → IO (Maybe XmlExp)`

Tries to read a file with an XML document and returns the corresponding XML expression, if possible. If file or parse errors occur, Nothing is returned.

`readFileWithXmlDocs :: String → IO [XmlExp]`

Reads a file with an arbitrary sequence of XML documents and returns the list of corresponding XML expressions.

`parseXmlString :: String → [XmlExp]`

Transforms an XML string into a list of XML expressions. If the XML string is a well structured document, the list of XML expressions should contain exactly one element.

`updateXmlFile :: (XmlExp → XmlExp) → String → IO ()`

An action that updates the contents of an XML file by some transformation on the XML document.

A.4.9 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`

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

[9] `http://www-ps.informatik.uni-kiel.de/~sebf/projects/xmlconv/`

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 repetitions and do not represent XML elements.

`float :: XmlConv NotRepeatable NoElem Float`

Creates an XML converter for float values. Float values must not be used in repetitions and do not represent XML elements.

`char :: XmlConv NotRepeatable NoElem Char`

Creates an XML converter for character values. Character values must not be used in repetitions and do not represent XML elements.

`string :: XmlConv NotRepeatable NoElem String`

Creates an XML converter for string values. String values must not be used in repetitions and do not represent XML elements.

`(!) :: XmlConv a b c → XmlConv a b c → XmlConv a b c`

Parallel composition of XML converters.

`element :: String → XmlConv a b c → XmlConv Repeatable Elem c`

Takes an arbitrary XML converter and returns a converter representing an XML element that contains the corresponding data. XML elements may be used in repetitions.

`empty :: a → XmlConv NotRepeatable NoElem a`

Takes a value and returns an XML converter for this value which is not represented as XML data. Empty XML data must not be used in repetitions and does not represent an XML element.

`attr :: String → (String → a,a → String) → XmlConv NotRepeatable NoElem a`

Takes a name and string conversion functions and returns an XML converter that represents an attribute. Attributes must not be used in repetitions and do not represent an XML element.

`adapt :: (a → b,b → a) → XmlConv c d a → XmlConv c d b`

Converts between arbitrary XML converters for different types.

`opt :: XmlConv a b c → XmlConv NotRepeatable NoElem (Maybe c)`

Creates a converter for arbitrary optional XML data. Optional XML data must not be used in repetitions and does not represent an XML element.

`rep :: XmlConv Repeatable a b → XmlConv NotRepeatable NoElem [b]`

Takes an XML converter representing repeatable data and returns an XML converter that represents repetitions of this data. Repetitions must not be used in other repetitions and do not represent XML elements.

`aInt :: String → XmlConv NotRepeatable NoElem Int`

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

`aFloat :: String → XmlConv NotRepeatable NoElem Float`

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

`aChar :: String → XmlConv NotRepeatable NoElem Char`

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

`aString :: String → XmlConv NotRepeatable NoElem String`

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

`aBool :: String → String → String → XmlConv NotRepeatable NoElem Bool`

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

`eInt :: String → XmlConv Repeatable Elem Int`

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

`eFloat :: String → XmlConv Repeatable Elem Float`

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

`eChar :: String → XmlConv Repeatable Elem Char`

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

`eString :: String → XmlConv Repeatable Elem String`

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

`eBool :: String → String → XmlConv Repeatable Elem Bool`

Creates an XML converter for boolean elements. Boolean elements may be used in repetitions.

`eEmpty :: String → a → XmlConv Repeatable Elem a`

Takes a name and a value and creates an empty XML element that represents the given value. The created element may be used in repetitions.

`eOpt :: String → XmlConv a b c → XmlConv Repeatable Elem (Maybe c)`

Creates an XML converter that represents an element containing optional XML data. The created element may be used in repetitions.

`eRep :: String → XmlConv Repeatable a b → XmlConv Repeatable Elem [b]`

Creates an XML converter that represents an element containing repeated XML data. The created element may be used in repetitions.

`seq1 :: (a → b) → XmlConv c d a → XmlConv c NoElem b`

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

`repSeq1 :: (a → b) → XmlConv Repeatable c a → XmlConv NotRepeatable NoElem [b]`

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions but does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

`eSeq1 :: String → (a → b) → XmlConv c d a → XmlConv Repeatable Elem b`

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

`eRepSeq1 :: String → (a → b) → XmlConv Repeatable c a → XmlConv Repeatable Elem [b]`

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

`seq2 :: (a → b → c) → XmlConv d e a → XmlConv f g b → XmlConv NotRepeatable NoElem c`

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

`repSeq2 :: (a → b → c) → XmlConv Repeatable d a → XmlConv Repeatable e b → XmlConv NotRepeatable NoElem [c]`

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

`eSeq2 :: String → (a → b → c) → XmlConv d e a → XmlConv f g b → XmlConv Repeatable Elem c`

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

`eRepSeq2 :: String → (a → b → c) → XmlConv Repeatable d a → XmlConv Repeatable e b → XmlConv Repeatable Elem [c]`

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

`seq3 :: (a → b → c → d) → XmlConv e f a → XmlConv g h b → XmlConv i j c → XmlConv NotRepeatable NoElem d`

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

`repSeq3 :: (a → b → c → d) → XmlConv Repeatable e a → XmlConv Repeatable f b → XmlConv Repeatable g c → XmlConv NotRepeatable NoElem [d]`

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

`eSeq3 :: String → (a → b → c → d) → XmlConv e f a → XmlConv g h b → XmlConv i j c → XmlConv Repeatable Elem d`

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

`eRepSeq3 :: String → (a → b → c → d) → XmlConv Repeatable e a → XmlConv Repeatable f b → XmlConv Repeatable g c → XmlConv Repeatable Elem [d]`

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

`seq4 :: (a → b → c → d → e) → XmlConv f g a → XmlConv h i b → XmlConv j k c → XmlConv l m d → XmlConv NotRepeatable NoElem e`

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

`repSeq4 :: (a → b → c → d → e) → XmlConv Repeatable f a → XmlConv Repeatable g b → XmlConv Repeatable h c → XmlConv Repeatable i d → XmlConv NotRepeatable NoElem [e]`

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

`eSeq4 :: String → (a → b → c → d → e) → XmlConv f g a → XmlConv h i b → XmlConv j k c → XmlConv l m d → XmlConv Repeatable Elem e`

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

`eRepSeq4 :: String → (a → b → c → d → e) → XmlConv Repeatable f a → XmlConv Repeatable g b → XmlConv Repeatable h c → XmlConv Repeatable i d → XmlConv Repeatable Elem [e]`

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

`seq5 :: (a → b → c → d → e → f) → XmlConv g h a → XmlConv i j b → XmlConv k l c → XmlConv m n d → XmlConv o p e → XmlConv NotRepeatable NoElem f`

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

`repSeq5 :: (a → b → c → d → e → f) → XmlConv Repeatable g a → XmlConv Repeatable h b → XmlConv Repeatable i c → XmlConv Repeatable j d → XmlConv Repeatable k e → XmlConv NotRepeatable NoElem [f]`

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

`eSeq5 :: String → (a → b → c → d → e → f) → XmlConv g h a → XmlConv i j b → XmlConv k l c → XmlConv m n d → XmlConv o p e → XmlConv Repeatable Elem f`

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in 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 Elem [f]`

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

`seq6 :: (a → b → c → d → e → f → g) → XmlConv h i a → XmlConv j k b → XmlConv l m c → XmlConv n o d → XmlConv p q e → XmlConv r s f → XmlConv NotRepeatable NoElem g`

Creates an XML converter representing a sequence of arbitrary XML data. The sequence must not be used in repetitions and does not represent an XML element.

`repSeq6 :: (a → b → c → d → e → f → g) → XmlConv Repeatable h a → XmlConv Repeatable i b → XmlConv Repeatable j c → XmlConv Repeatable k d → XmlConv Repeatable l e → XmlConv Repeatable m f → XmlConv NotRepeatable NoElem [g]`

Creates an XML converter that represents a repetition of a sequence of repeatable XML data. The repetition may be used in other repetitions and does not represent an XML element. This combinator is provided because converters for repeatable sequences cannot be constructed by the seq combinators.

`eSeq6 :: String → (a → b → c → d → e → f → g) → XmlConv h i a → XmlConv j k b → XmlConv l m c → XmlConv n o d → XmlConv p q e → XmlConv r s f → XmlConv Repeatable Elem g`

Creates an XML converter for compound values represented as an XML element with children that correspond to the values components. The element can be used in repetitions.

`eRepSeq6 :: String → (a → b → c → d → e → f → g) → XmlConv Repeatable h a → XmlConv Repeatable i b → XmlConv Repeatable j c → XmlConv Repeatable k d → XmlConv Repeatable l e → XmlConv Repeatable m f → XmlConv Repeatable Elem [g]`

Creates an XML converter for repetitions of sequences represented as an XML element that can be used in repetitions.

A.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:

`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.

`type CTVarIName = (Int,String)`

The data type for representing type variables. They are represented by (i,n) where i is a type variable index which is unique inside a function and n is a name (if possible, the name written in the source program).

`type CField a = (String,a)`

Labeled record fields

`type CLabel = String`

Identifiers for record labels (extended syntax).

`type CVarIName = (Int,String)`

Data types for representing object variables. Object variables occurring in expressions are represented by (Var i) where i is a variable index.

`data CurryProg`

Data type for representing a Curry module in the intermediate form. A value of this data type has the form `(CProg modname imports typedecls functions opdecls)` where modname: name of this module, imports: list of modules names that are imported, typedecls, opdecls, functions: see below

Exported constructors:

- `CurryProg :: String → [String] → [CTypeDecl] → [CFuncDecl] → [COpDecl] → CurryProg` `data CVisibility`

Data type to specify the visibility of various entities.

Exported constructors:

- `Public :: CVisibility`
- `Private :: CVisibility`

`data CTypeDecl`

Data type for representing definitions of algebraic data types and type synonyms.

A data type definition of the form

`data t x1...xn = ...| c t1....tkc |...`

is represented by the Curry term

(CType t v [i1,...,in] [...(CCons c kv v `[t1,...,tkc])...]`)

where each `ij` is the index of the type variable `xj`.

Note: the type variable indices are unique inside each type declaration and are usually numbered from 0

Thus, a data type declaration consists of the name of the data type, a list of type parameters and a list of constructor declarations.

Exported constructors:

- `CType :: (String,String) → CVisibility → [(Int,String)] → [CConsDecl] → CTypeDecl`
- `CTypeSyn :: (String,String) → CVisibility → [(Int,String)] → CTypeExpr → CTypeDecl`

`data CConsDecl`

A constructor declaration consists of the name and arity of the constructor and a list of the argument types of the constructor.

Exported constructors:

- `CCons :: (String,String) → Int → CVisibility → [CTypeExpr] → CConsDecl`

`data CTypeExpr`

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:

- `CTVar :: (Int,String)` → `CTypeExpr`
- `CFuncType :: CTypeExpr` → `CTypeExpr` → `CTypeExpr`
- `CTCons :: (String,String)` → `[CTypeExpr]` → `CTypeExpr`
- `CRecordType :: [(String,CTypeExpr)]` → `(Maybe (Int,String))` → `CTypeExpr`

`data COpDecl`

Data type for operator declarations. An operator declaration "fix p n" in Curry corresponds to the AbstractCurry term (COp n fix p).

Exported constructors:

- `COp :: (String,String)` → `CFixity` → `Int` → `COpDecl`

`data CFixity`

Data type for operator associativity

Exported constructors:

- `CInfixOp :: CFixity`
- `CInfixlOp :: CFixity`
- `CInfixrOp :: CFixity`

`data CFuncDecl`

Data type for representing function declarations.

A function declaration in AbstractCurry is a term of the form

(CFunc name arity visibility type (CRules eval `[CRule rule1,...,rulek]`))

and represents the function `name` defined by the rules `rule1,...,rulek`.

Note: the variable indices are unique inside each rule

External functions are represented as `(CFunc name arity type (CExternal s))` where s is the external name associated to this function.

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` → `CRules` → `CFuncDecl`

- CmtFunc :: String → (String,String) → Int → CVisibility → CTypeExpr → CRules → CFuncDecl

data CRules

A rule is either a list of formal parameters together with an expression (i.e., a rule in flat form), a list of general program rules with an evaluation annotation, or it is externally defined

Exported constructors:

- CRules :: CEvalAnnot → [CRule] → CRules

- CExternal :: String → CRules

data CEvalAnnot

Data type for classifying evaluation annotations for functions. They can be either flexible (default), rigid, or choice.

Exported constructors:

- CFlex :: CEvalAnnot

- CRigid :: CEvalAnnot

- CChoice :: CEvalAnnot

data CRule

The most general form of a rule. It consists of a list of patterns (left-hand side), a list of guards ("success" if not present in the source text) with their corresponding right-hand sides, and a list of local declarations.

Exported constructors:

- CRule :: [CPattern] → [(CExpr,CExpr)] → [CLocalDecl] → CRule

data CLocalDecl

Data type for representing local (let/where) declarations

Exported constructors:

- CLocalFunc :: CFuncDecl → CLocalDecl

- CLocalPat :: CPattern → CExpr → [CLocalDecl] → CLocalDecl

- CLocalVar :: (Int,String) → CLocalDecl

data CExpr

Data type for representing Curry expressions.

Exported constructors:

- `CVar :: (Int,String)` → `CExpr`
- `CLit :: CLiteral` → `CExpr`
- `CSymbol :: (String,String)` → `CExpr`
- `CApply :: CExpr` → `CExpr` → `CExpr`
- `CLambda :: [CPattern]` → `CExpr` → `CExpr`
- `CLetDecl :: [CLocalDecl]` → `CExpr` → `CExpr`
- `CDoExpr :: [CStatement]` → `CExpr`
- `CListComp :: CExpr` → `[CStatement]` → `CExpr`
- `CCase :: CExpr` → `[CBranchExpr]` → `CExpr`
- `CRecConstr :: [(String,CExpr)]` → `CExpr`
- `CRecSelect :: CExpr` → `String` → `CExpr`
- `CRecUpdate :: [(String,CExpr)]` → `CExpr` → `CExpr`

`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 CPattern`

Data type for representing pattern expressions.

Exported constructors:

- `CPVar :: (Int,String)` → `CPattern`
- `CPLit :: CLiteral` → `CPattern`
- `CPComb :: (String,String)` → `[CPattern]` → `CPattern`
- `CPAs :: (Int,String)` → `CPattern` → `CPattern`

- `CPFuncComb :: (String,String) → [CPattern] → CPattern`
- `CPLazy :: CPattern → CPattern`
- `CPRecord :: [(String,CPattern)] → (Maybe CPattern) → CPattern`

`data CBranchExpr`

Data type for representing branches in case expressions.

Exported constructors:

- `CBranch :: CPattern → CExpr → CBranchExpr`
- `CGuardedBranch :: CPattern → [(CExpr,CExpr)] → CBranchExpr`

`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`

Exported functions:

`readCurry :: String → IO CurryProg`

I/O action which parses a Curry program and returns the corresponding typed Abstract Curry program. Thus, the argument is the file name without suffix ".curry" or ".lcurry") and the result is a Curry term representing this program.

`readUntypedCurry :: String → IO CurryProg`

I/O action which parses a Curry program and returns the corresponding untyped Abstract Curry program. Thus, the argument is the file name without suffix ".curry" or ".lcurry") and the result is a Curry term representing this program.

`readCurryWithParseOptions :: String → FrontendParams → IO CurryProg`

I/O action which reads a typed Curry program from a file (with extension ".acy") with respect to some parser options. This I/O action is used by the standard action `readCurry`. It is currently predefined only in Curry2Prolog.

`readUntypedCurryWithParseOptions :: String → FrontendParams → IO CurryProg`

I/O action which reads an untyped Curry program from a file (with extension ".uacy") with respect to some parser options. For more details see function `readCurryWithParseOptions`

`abstractCurryFileName :: String → String`

Transforms a name of a Curry program (with or without suffix ".curry" or ".lcurry") into the name of the file containing the corresponding AbstractCurry program.

`untypedAbstractCurryFileName :: String → String`

Transforms a name of a Curry program (with or without suffix ".curry" or ".lcurry") into the name of the file containing the corresponding untyped AbstractCurry program.

`readAbstractCurryFile :: String → IO CurryProg`

I/O action which reads an AbstractCurry program from a file in ".acy" format. In contrast to `readCurry`, this action does not parse a source program. Thus, the argument must be the name of an existing file (with suffix ".acy") containing an AbstractCurry program in ".acy" format and the result is a Curry term representing this program. It is currently predefined only in Curry2Prolog.

`tryReadACYFile :: String → IO (Maybe CurryProg)`

Tries to read an AbstractCurry file and returns

- Left err , where err specifies the error occurred
- Right prog, where prog is the AbstractCurry program

`writeAbstractCurryFile :: String → CurryProg → IO ()`

Writes an AbstractCurry program into a file in ".acy" format. The first argument must be the name of the target file (with suffix ".acy").

A.5.2 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.5.3 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.5.4 Library CurryStringClassifier

The Curry string classifier is a simple tool to process strings containing Curry source code. The source string is classified into the following categories:

- moduleHead - module interface, imports, operators
- code - the part where the actual program is defined
- big comment - parts enclosed in {- ... -}
- small comment - from "–" to the end of a line
- text - a string, i.e. text enclosed in "..."
- letter - the given string is the representation of a character
- meta - containing information for meta programming

For an example to use the state scanner cf. addtypes, the tool to add function types to a given program.

Exported types:

`type Tokens = [Token]`

`data Token`

The different categories to classify the source code.

Exported constructors:

- `SmallComment :: String` → `Token`
- `BigComment :: String` → `Token`
- `Text :: String` → `Token`
- `Letter :: String` → `Token`
- `Code :: String` → `Token`
- `ModuleHead :: String` → `Token`
- `Meta :: String` → `Token`

Exported functions:

`isSmallComment :: Token → Bool`

test for category "SmallComment"

`isBigComment :: Token → Bool`

test for category "BigComment"

`isComment :: Token → Bool`

test if given token is a comment (big or small)

`isText :: Token → Bool`

test for category "Text" (String)

`isLetter :: Token → Bool`

test for category "Letter" (Char)

`isCode :: Token → Bool`

test for category "Code"

`isModuleHead :: Token → Bool`

test for category "ModuleHead", ie imports and operator declarations

`isMeta :: Token → Bool`

test for category "Meta", ie between {+ and +}

`scan :: String → [Token]`

Divides the given string into the six categories. For applications it is important to know whether a given part of code is at the beginning of a line or in the middle. The state scanner organizes the code in such a way that every string categorized as "Code" **always** starts in the middle of a line.

`plainCode :: [Token] → String`

Yields the program code without comments (but with the line breaks for small comments).

`unscan :: [Token] → String`

Inverse function of scan, i.e., unscan (scan x) = x. unscan is used to yield a program after changing the list of tokens.

`readScan :: String → IO [Token]`

return tokens for given filename

`testScan :: String → IO ()`

test whether (unscan . scan) is identity

A.5.5 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 translation_table)
```

where `modname` is the name of this module, `imports` is the list of modules names that are imported, `typedecls`, `opdecls`, `functions`, translation of type names and constructor/function names are explained see below

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.

A data type definition of the form

```
data t x1...xn = ...| c t1....tkc |...
```

is represented by the FlatCurry term

```
(Type t [i1,...,in] [...(Cons c kc [t1,...,tkc])...])
```

where each `ij` is the index of the type variable `xj`.

Note: the type variable indices are unique inside each type declaration and are usually numbered from 0

Thus, a data type declaration consists of the name of the data type, a list of type parameters and a list of constructor declarations.

Exported constructors:

- `Type :: (String,String) → Visibility → [Int] → [ConsDecl] → TypeDecl`
- `TypeSyn :: (String,String) → Visibility → [Int] → TypeExpr → TypeDecl`

`data ConsDecl`

A constructor declaration consists of the name and arity of the constructor and a list of the argument types of the constructor.

Exported constructors:

- `Cons :: (String,String) → Int → Visibility → [TypeExpr] → ConsDecl`

`data TypeExpr`

Data type for type expressions. A type expression is either a type variable, a function type, or a type constructor application.

Note: the names of the predefined type constructors are "Int", "Float", "Bool", "Char", "IO", "Success", "()" (unit type), "(,...,)" (tuple types), "[]" (list type)

Exported constructors:

- `TVar :: Int → TypeExpr`
- `FuncType :: TypeExpr → TypeExpr → TypeExpr`
- `TCons :: (String,String) → [TypeExpr] → TypeExpr`

`data OpDecl`

Data type for operator declarations. An operator declaration `fix p n` in Curry corresponds to the FlatCurry term `(Op n fix p)`.

Exported constructors:

- `Op :: (String,String) → Fixity → Int → OpDecl`

`data Fixity`

Data types for the different choices for the fixity of an operator.

Exported constructors:

- `InfixOp :: Fixity`
- `InfixlOp :: Fixity`
- `InfixrOp :: Fixity`

`data FuncDecl`

Data type for representing function declarations.

A function declaration in FlatCurry is a term of the form

```
(Func name k type (Rule [i1,...,ik] e))
```

and represents the function `name` with definition

```
name :: type
name x1...xk = e
```

where each `ij` is the index of the variable `xj`.

Note: the variable indices are unique inside each function declaration and are usually numbered from 0

External functions are represented as

```
(Func name arity type (External s))
```

where s is the external name associated to this function.

Thus, a function declaration consists of the name, arity, type, and rule.

Exported constructors:

- `Func :: (String,String) → Int → Visibility → TypeExpr → Rule → FuncDecl`

`data Rule`

A rule is either a list of formal parameters together with an expression or an "External" tag.

Exported constructors:

- `Rule :: [Int] → Expr → Rule`
- `External :: String → Rule`

`data CaseType`

Data type for classifying case expressions. Case expressions can be either flexible or rigid in Curry.

Exported constructors:

- `Rigid :: CaseType`
- `Flex :: CaseType`

`data CombType`

Data type for classifying combinations (i.e., a function/constructor applied to some arguments).

Exported constructors:

- `FuncCall :: CombType`

  `FuncCall`

  – a call to a function where all arguments are provided

- `ConsCall :: CombType`

  `ConsCall`

  – a call with a constructor at the top, all arguments are provided

- `FuncPartCall :: Int → CombType`

  `FuncPartCall`

  – a partial call to a function (i.e., not all arguments are provided) where the parameter is the number of missing arguments

- `ConsPartCall :: Int → CombType`

  `ConsPartCall`

  – a partial call to a constructor (i.e., not all arguments are provided) where the parameter is the number of missing arguments

`data Expr`

Data type for representing expressions.

Remarks:

if-then-else expressions are represented as function calls:

```
(if e1 then e2 else e3)
```

is represented as

```
(Comb FuncCall ("Prelude","if_then_else") [e1,e2,e3])
```

Higher-order applications are represented as calls to the (external) function `apply`. For instance, the rule

```
app f x = f x
```

is represented as

```
(Rule  [0,1] (Comb FuncCall ("Prelude","apply") [Var 0, Var 1]))
```

A conditional rule is represented as a call to an external function `cond` where the first argument is the condition (a constraint). For instance, the rule

```
equal2 x | x=:=2 = success
```

is represented as

```
(Rule [0]
      (Comb FuncCall ("Prelude","cond")
            [Comb FuncCall ("Prelude","=:=") [Var 0, Lit (Intc 2)],
             Comb FuncCall ("Prelude","success") []]))
```

Exported constructors:

- `Var :: Int → Expr`

  `Var`

  – variable (represented by unique index)

- `Lit :: Literal → Expr`

  `Lit`

  – literal (Int/Float/Char constant)

- `Comb :: CombType → (String,String) → [Expr] → Expr`

  `Comb`

  – application `(f e1 ... en)` of function/constructor `f` with `n<=arity(f)`

- `Let :: [(Int,Expr)] → Expr → Expr`
- `Free :: [Int] → Expr → Expr`

  `Free`

  – introduction of free local variables

- `Or :: Expr → Expr → Expr`

  `Or`

  – disjunction of two expressions (used to translate rules with overlapping left-hand sides)

- `Case :: CaseType → Expr → [BranchExpr] → Expr`

  `Case`

  – case distinction (rigid or flex)

- `Typed :: Expr → TypeExpr → Expr`

`data BranchExpr`

Data type for representing branches in a case expression.

Branches "(m.c x1...xn) `->` e" in case expressions are represented as

```
(Branch (Pattern (m,c) [i1,...,in]) e)
```

where each `ij` is the index of the pattern variable `xj`, or as

```
(Branch (LPattern (Intc i)) e)
```

for integers as branch patterns (similarly for other literals like float or character constants).

Exported constructors:

- `Branch :: Pattern → Expr → BranchExpr`

`data Pattern`

Data type for representing patterns in case expressions.

Exported constructors:

- `Pattern :: (String,String) → [Int] → Pattern`

- `LPattern :: Literal → Pattern`

`data Literal`

Data type for representing literals occurring in an expression or case branch. It is either an integer, a float, or a character constant.

Exported constructors:

- `Intc :: Int → Literal`
- `Floatc :: Float → Literal`
- `Charc :: Char → Literal`

Exported functions:

`readFlatCurry :: String → IO Prog`

I/O action which parses a Curry program and returns the corresponding FlatCurry program. Thus, the argument is the file name without suffix ".curry" (or ".lcurry") and the result is a FlatCurry term representing this program.

`readFlatCurryWithParseOptions :: String → FrontendParams → IO Prog`

I/O action which reads a FlatCurry program from a file with respect to some parser options. This I/O action is used by the standard action `readFlatCurry`. It is currently predefined only in Curry2Prolog.

`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 program, i.e., a FlatCurry program containing only "Public" entities and function definitions without rules (i.e., external functions). The argument is the file name without suffix ".curry" (or ".lcurry") and the result is a FlatCurry term representing the interface of this program.

`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.5.6 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

`progFuncs :: Prog → [FuncDecl]`

get functions from program

`progOps :: Prog → [OpDecl]`

get infix operators from program

`updProg :: (String → String) → ([String] → [String]) → ([TypeDecl] → [TypeDecl]) → ([FuncDecl] → [FuncDecl]) → ([OpDecl] → [OpDecl]) → Prog → Prog`

update program

`updProgName :: (String → String) → Prog → Prog`

update name of program

`updProgImports :: ([String] → [String]) → Prog → Prog`

update imports of program

`updProgTypes :: ([TypeDecl] → [TypeDecl]) → Prog → Prog`

update type declarations of program

`updProgFuncs :: ([FuncDecl] → [FuncDecl]) → Prog → Prog`

update functions of program

`updProgOps :: ([OpDecl] → [OpDecl]) → Prog → Prog`

update infix operators of program

`allVarsInProg :: Prog → [Int]`

get all program variables (also from patterns)

`updProgExps :: (Expr → Expr) → Prog → Prog`

lift transformation on expressions to program

`rnmAllVarsInProg :: (Int → Int) → Prog → Prog`

rename programs variables

`updQNamesInProg :: ((String,String) → (String,String)) → Prog → Prog`

update all qualified names in program

`rnmProg :: String → Prog → Prog`

rename program (update name of and all qualified names in program)

`trType :: ((String,String) → Visibility → [Int] → [ConsDecl] → a) → ((String,String) → Visibility → [Int] → TypeExpr → a) → TypeDecl → a`

transform type declaration

`typeName :: TypeDecl → (String,String)`

get name of type declaration

`typeVisibility :: TypeDecl → Visibility`

get visibility of type declaration

`typeParams :: TypeDecl → [Int]`

get type parameters of type declaration

`typeConsDecls :: TypeDecl → [ConsDecl]`

get constructor declarations from type declaration

`typeSyn :: TypeDecl → TypeExpr`

get synonym of type declaration

`isTypeSyn :: TypeDecl → Bool`

is type declaration a type synonym?

`updType :: ((String,String) → (String,String)) → (Visibility → Visibility) → ([Int] → [Int]) → ([ConsDecl] → [ConsDecl]) → (TypeExpr → TypeExpr) → TypeDecl → TypeDecl`

update type declaration

`updTypeName :: ((String,String) → (String,String)) → TypeDecl → TypeDecl`

update name of type declaration

`updTypeVisibility :: (Visibility → Visibility) → TypeDecl → TypeDecl`

update visibility of type declaration

`updTypeParams :: ([Int] → [Int]) → TypeDecl → TypeDecl`

update type parameters of type declaration

`updTypeConsDecls :: ([ConsDecl] → [ConsDecl]) → TypeDecl → TypeDecl`

update constructor declarations of type declaration

`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 varoables in let declaration

`letBody :: Expr → Expr`

get body of let declaration

`freeVars :: Expr → [Int]`

get variable indices from declaration of free variables

`freeExpr :: Expr → Expr`

get expression from declaration of free variables

`orExps :: Expr → [Expr]`

get expressions from or-expression

`caseType :: Expr → CaseType`

get case-type of case expression

`caseExpr :: Expr → Expr`

get scrutinee of case expression

`caseBranches :: Expr → [BranchExpr]`

get branch expressions from case expression

`isVar :: Expr → Bool`

is expression a variable?

`isLit :: Expr → Bool`

is expression a literal expression?

`isComb :: Expr → Bool`

is expression combined?

`isLet :: Expr → Bool`

is expression a let expression?

`isFree :: Expr → Bool`

is expression a declaration of free variables?

`isOr :: Expr → Bool`

is expression an or-expression?

`isCase :: Expr → Bool`

is expression a case expression?

`trExpr :: (Int → a) → (Literal → a) → (CombType → (String,String) → [a] → a) → ([(Int,a)] → a → a) → ([Int] → a → a) → (a → a → a) → (CaseType → a → [b] → a) → (Pattern → a → b) → (a → TypeExpr → a) → Expr → a` transform expression

`updVars :: (Int → Expr) → Expr → Expr`

update all variables in given expression

`updLiterals :: (Literal → Expr) → Expr → Expr`

update all literals in given expression

`updCombs :: (CombType → (String,String) → [Expr] → Expr) → Expr → Expr`

update all combined expressions in given expression

`updLets :: ([(Int,Expr)] → Expr → Expr) → Expr → Expr`

update all let expressions in given expression

`updFrees :: ([Int] → Expr → Expr) → Expr → Expr`

update all free declarations in given expression

`updOrs :: (Expr → Expr → Expr) → Expr → Expr`

update all or expressions in given expression

`updCases :: (CaseType → Expr → [BranchExpr] → Expr) → Expr → Expr`

update all case expressions in given expression

`updBranches :: (Pattern → Expr → BranchExpr) → Expr → Expr`

update all case branches in given expression

`updTypeds :: (Expr → TypeExpr → Expr) → Expr → Expr`

update all typed expressions in given expression

`isFuncCall :: Expr → Bool`

is expression a call of a function where all arguments are provided?

`isFuncPartCall :: Expr → Bool`

is expression a partial function call?

`isConsCall :: Expr → Bool`

is expression a call of a constructor?

`isConsPartCall :: Expr → Bool`

is expression a partial constructor call?

`isGround :: Expr → Bool`

is expression fully evaluated?

`allVars :: Expr → [Int]`

get all variables (also pattern variables) in expression

`rnmAllVars :: (Int → Int) → Expr → Expr`

rename all variables (also in patterns) in expression

`updQNames :: ((String,String) → (String,String)) → Expr → Expr`

update all qualified names in expression

`trBranch :: (Pattern → Expr → a) → BranchExpr → a`

transform branch expression

`branchPattern :: BranchExpr → Pattern`

get pattern from branch expression

`branchExpr :: BranchExpr → Expr`

get expression from branch expression

`updBranch :: (Pattern → Pattern) → (Expr → Expr) → BranchExpr → BranchExpr`

update branch expression

`updBranchPattern :: (Pattern → Pattern) → BranchExpr → BranchExpr`

update pattern of branch expression

`updBranchExpr :: (Expr → Expr) → BranchExpr → BranchExpr`

update expression of branch expression

`trPattern :: ((String,String) → [Int] → a) → (Literal → a) → Pattern → a`

transform pattern

`patCons :: Pattern → (String,String)`

get name from constructor pattern

`patArgs :: Pattern → [Int]`

get arguments from constructor pattern

`patLiteral :: Pattern → Literal`

get literal from literal pattern

`isConsPattern :: Pattern → Bool`

is pattern a constructor pattern?

`updPattern :: ((String,String) → (String,String)) → ([Int] → [Int]) → (Literal → Literal) → Pattern → Pattern`

update pattern

`updPatCons :: ((String,String) → (String,String)) → Pattern → Pattern`

update constructors name of pattern

`updPatArgs :: ([Int] → [Int]) → Pattern → Pattern`

update arguments of constructor pattern

`updPatLiteral :: (Literal → Literal) → Pattern → Pattern`

update literal of pattern

`patExpr :: Pattern → Expr`

build expression from pattern

A.5.7 Library FlatCurryRead

This library defines operations to read a FlatCurry programs or interfaces together with all its imported modules in the current load path.

Exported functions:

`readFlatCurryWithImports :: String → IO [Prog]`

Reads a FlatCurry program together with all its imported modules. The argument is the name of the main module (possibly with a directory prefix).

`readFlatCurryWithImportsInPath :: [String] → String → IO [Prog]`

Reads a FlatCurry program together with all its imported modules in a given load path. The arguments are a load path and the name of the main module.

`readFlatCurryIntWithImports :: String → IO [Prog]`

Reads a FlatCurry interface together with all its imported module interfaces. The argument is the name of the main module (possibly with a directory prefix). If there is no interface file but a FlatCurry file (suffix ".fcy"), the FlatCurry file is read instead of the interface.

`readFlatCurryIntWithImportsInPath :: [String] → String → IO [Prog]`

Reads a FlatCurry interface together with all its imported module interfaces in a given load path. The arguments are a load path and the name of the main module. If there is no interface file but a FlatCurry file (suffix ".fcy"), the FlatCurry file is read instead of the interface.

A.5.8 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.5.9 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:

`flatCurry2XmlFile :: Prog → String → IO ()`

Transforms a FlatCurry program term into a corresponding XML file.

`flatCurry2Xml :: Prog → XmlExp`

Transforms a FlatCurry program term into a corresponding XML expression.

`xmlFile2FlatCurry :: String → IO Prog`

Reads an XML file with a FlatCurry program and returns the FlatCurry program.

`xml2FlatCurry :: XmlExp → Prog`

Transforms an XML term into a FlatCurry program.

A.5.10 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:

`data FlexRigidResult`

Datatype for representing a flex/rigid status of an expression.

Exported constructors:

- `UnknownFR :: FlexRigidResult`
- `ConflictFR :: FlexRigidResult`
- `KnownFlex :: FlexRigidResult`
- `KnownRigid :: FlexRigidResult`

Exported functions:

`getFlexRigid :: Expr → FlexRigidResult`

Computes the rigid/flex status of a FlatCurry expression. This function checks all cases in this expression. If the expression has rigid as well as flex cases (which cannot be the case for source level programs but might occur after some program transformations), the result ConflictFR is returned.

A.5.11 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.

`printCProg :: String → IO ()`

(printCProg modulname) pretty prints the typed Abstract Curry program of `modulname` produced by `AbstractCurry.readCurry` and fits it to a page width of 78 characters. The output is standard io.

`printUCProg :: String → IO ()`

(printUCProg modulname) pretty prints the untyped Abstract Curry program of `modulname` produced by `AbstractCurry.readUntypedCurry` and fits it to a page width of 78 characters. The output ist standard io.

`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:

```
_emphasize_
*emphasize*
__strong__
**strong**
```

Furthermore, one can also mark `program code` text by backtick quotes (`):

```
The function `fib` computes Fibonacci numbers.
```

Web links can be put in angle brackets, like in the link `http://www.google.com`:

```
<http://www.google.com>
```

Currently, only links starting with 'http' are recognized (so that one can also use HTML markup). If one wants to put a link under a text, one can put the text in square brackets directly followed by the link in round brackets, as in Google:

```
[Google](http://www.google.com)
```

If one wants to put a character that has a specific meaning in the syntax of Markdown, like * or _, in the output document, it should be escaped with a backslash, i.e., a backslash followed by a special character in the source text is translated into the given character (this also holds for program code, see below). For instance, the input text

```
\_word\_
```

produces the output "_word_". The following backslash escapes are recognized:

```
\   backslash
`   backtick
*   asterisk
_   underscore
{}  curly braces
[]  square brackets
```

```
()  parentheses
#   hash symbol
+   plus symbol
-   minus symbol (dash)
.   dot
    blank
!   exclamation mark
```

B.2 Lists and Block Formatting

An **unordered list** (i.e., without numbering) is introduced by putting a star in front of the list elements (where the star can be preceded by blanks). The individual list elements must contain the same indentation, as in

```
* First list element
  with two lines

* Next list element.

  It contains two paragraphs.

* Final list element.
```

This is formatted as follows:

- First list element with two lines
- Next list element.

  It contains two paragraphs.
- Final list element.

Instead of a star, one can also put dashes or plus to mark unordered list items. Furthermore, one could nest lists. Thus, the input text

```
- Color:
  + Yellow
  + Read
  + Blue
- BW:
  + Black
  + White
```

is formatted as

- Color:

- Yellow
- Read
- Blue

- BW:
  - Black
  - White

Similarly, **ordered lists** (i.e., with numbering each item) are introduced by a number followed by a dot and at least one blank. All following lines belonging to the same numbered item must have the same indent as the first line. The actual value of the number is not important. Thus, the input

```
1. First element

99. Second
    element
```

is formatted as

1. First element

2. Second element

A quotation block is marked by putting a right angle followed by a blank in front of each line:

```
> This is
> a quotation.
```

It will be formatted as a quote element:

> This is a quotation.

A block containing **program code** starts with a blank line and is marked by intending each input line by *at least four spaces* where all following lines must have at least the same indentation as the first non-blank character of the first line:

```
    f x y = let z = (x,y)
             in (z,z)
```

The indentation is removed in the output:

```
f x y = let z = (x,y)
         in (z,z)
```

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 representations 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 *dir* is the directory where the Curry program is stored, the corresponding FlatCurry program is stored in the directory "*dir*`/.curry`".

`prog.fint`: This file contains the interface of the program in the so-called "FlatCurry" representation, i.e., it is similar to `prog.fcy` but contains only exported entities and the bodies of all functions omitted (i.e., "external"). This representation is useful for providing a fast access to module interfaces. This file is implicitly generated when a program is compiled with KiCS2 and stored in the same directory as `prog.fcy`.

`Curry_prog.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). If *dir* is the directory where the Curry program is stored, the corresponding Haskell program is stored in the directory "*dir*`/.curry/.kics2`".

`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.nda`.

`Curry_prog.hs`: This file contains a Haskell program as the result of translating the Curry program with the KiCS2 compiler. This file is stored in the same directory as `Curry_prog.nda`.

`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.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`.

`prog`: This file contains the executable after compiling and saving a program with KiCS2 (see command "`:save`" in Section 2.2).

D External Operations

Currently, KiCS2 has no general interface to external operations, i.e., operations whose semantics is not defined by program rules in a Curry program but by some code written in another programming language. Thus, if an external operation should be added to the system, this operation must be declared as `external` in the Curry source code and an implementation for this external operation must be provided for the run-time system. An external operation is defined as follows in the Curry source code:

1. Add a type declaration for the external operation somewhere in a module defining this operation (usually, the prelude or some system module).

2. For external operations it is not allowed to define any rule since their semantics is determined by an external implementation. Instead of the defining rules, you have to write

   ```
   f external
   ```

   below the type declaration for the external operation `f`.

Furthermore, an implementation of the external operation must be provided in the target language of the KiCS2 compiler, i.e., in Haskell, and inserted in the compiled code. In order to simplify this task, KiCS2 follows some code conventions that are described in the following.

Assume you want to implement your own concatenation for strings in a module `String`. The name and type of this string concatenation should be

```
sconc :: String → String → String
```

Since the primitive Haskell implementation of this operation does not now anything about the operational mechanism of Curry (e.g., needed narrowing, non-deterministic rewriting), the arguments need to be completely evaluated before the primitive implementation is called. This can be easily obtained by the prelude operation `($##)` that applies an operation to the *normal form* of the given argument, i.e., this operation evaluates the argument to its normal form before applying the operation to it.[10] Thus, we define `sconc` by

```
sconc :: String → String → String
sconc s1 s2 = (prim_sconc $## s1) $## s2

prim_sconc :: String → String → String
prim_sconc external
```

so that it is ensured that the external operation `prim_sconc` is always called with complete evaluated arguments.

In order to define the Haskell code implementing `prim_sconc`, one has to satisfy the naming conventions of KiCS2. The KiCS2 compiler generates the following code for the external operation `prim_sconc` (note that the generated Haskell code for the module `String` is stored in the file `.curry/kics2/Curry_String.hs`):

[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`.

```
d_C_prim_sconc :: Curry_Prelude.OP_List Curry_Prelude.C_Char
                → Curry_Prelude.OP_List Curry_Prelude.C_Char
                → ConstStore
                → Curry_Prelude.OP_List Curry_Prelude.C_Char
d_C_prim_sconc x1 x2 x3500 = external_d_C_prim_sconc x1 x2 x3500
```

The type constructors `OP_List` and `C_Char` of the prelude `Curry_Prelude`[11] correspond to the Curry type constructors for lists and characters. The Haskell operation `external_d_C_prim_sconc` is the external operation to be implemented in Haskell by the programmer. The additional argument of type `ConstStore` represents the current set of constraints when this operation is called. This argument is intended to provide a more efficient access to binding constraints and can be ignored in standard operations.

If `String.curry` contains the code of the Curry function `sconc` described above, the Haskell code implementing the external operations occurring in the module `String` must be in the file `External_String.hs` which is located in the same directory as the file `String.curry`. The KiCS2 compiler appends the code contained in `External_String.hs` to the generated code stored in the file `.curry/kics2/Curry_String.hs`.[12]

In order to complete our example, we have to write into the file `External_String.hs` a definition of the Haskell function `external_d_C_prim_sconc`. Thus, we start with the following definitions:

```
import qualified Curry_Prelude as CP

external_d_C_prim_sconc :: CP.OP_List CP.C_Char  → CP.OP_List CP.C_Char
                        → ConstStore  → CP.OP_List CP.C_Char
```

First, we import the standard prelude with the name `CP` in order to shorten the writing of type declarations. In order to write the final code of this operation, we have to convert the Curry-related types (like `C_Char`) into the corresponding Haskell types (like `Char`). Note that the Curry-related types contain information about non-deterministic or constrained values (see [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, `fromCurry` converts a Curry type into the corresponding Haskell type, and `toCurry` converts the Haskell type into the corresponding Curry type. Thus, we complete our example with the definition (note that we simply ignore the final argument representing the constraint store)

```
external_d_C_prim_sconc s1 s2 _ =
  toCurry ((fromCurry s1 ++ fromCurry s2) :: String)
```

Here, we use Haskell's concatenation operation "`++`" to concatenate the string arguments. The type annotation "`:: String`" is necessary because "`++`" is a polymorphic function so that the type inference system of Haskell has problems to determine the right instance of the conversion function.

The conversion between Curry types and Haskell types, i.e., the family of conversion operation `fromCurry` and `toCurry`, is defined in the KiCS2 implementation for all standard data types. In particular, it is also defined on function types so that one can easily implement external Curry I/O

[11] Note that all translated Curry modules are imported in the Haskell code fully qualified in order to avoid name conflicts.

[12] If the file `External_String.hs` contains also some import declarations at the beginning, these import declarations are put after the generated import declarations.

actions by using Haskell I/O actions. For instance, if we want to implement an external operation to print some string as an output line, we start by declaring the external operations in the Curry module `String`:

```
printString :: String → IO ()
printString s = prim_printString $## s

prim_printString :: String → IO ()
prim_printString external
```

Next we add the corresponding implementation in the file `External_String.hs` (where `C_IO` and `OP_Unit` are the names of the Haskell representation of the Curry type constructor `IO` and the Curry data type "`()`", respectively):

```
external_d_C_prim_printString :: CP.OP_List CP.C_Char → ConstStore
                              → CP.C_IO CP.OP_Unit
external_d_C_prim_printString s _ = toCurry putStrLn s
```

Here, Haskell's I/O action `putStrLn` of type "`String -> IO ()`" is transformed into a Curry I/O action "`toCurry putStrLn`" which has the type

```
CP.OP_List CP.C_Char → CP.C_IO CP.OP_Unit
```

When we compile the Curry module `String`, KiCS2 combines these definitions in the target program so that we can immediately use the externally defined operation `printString` in Curry programs.

As we have seen, KiCS2 transforms a name like `primOP` of an external operation into the name `external_d_C_primOP` for the Haskell operation to be implemented, i.e., only a specific prefix is added. However, this is only valid if no special characters occur in the Curry names. Otherwise (in order to generate a correct Haskell program), special characters are translated into specific names prefixed by "`OP_`". For instance, if we declare the external operation

```
(<&>) :: Int → Int → Int
(<&>) external
```

the generated Haskell module contains the code

```
d_OP_lt_ampersand_gt :: Curry_Prelude.C_Int → Curry_Prelude.C_Int
                     → ConstStore → Curry_Prelude.C_Int
d_OP_lt_ampersand_gt x1 x2 x3500 = external_d_OP_lt_ampersand_gt x1 x2 x3500
```

so that one has to implement the operation `external_d_OP_lt_ampersand_gt` in Haskell. If in doubt, one should look into the generated Haskell code about the names and types of the operations to be implemented.

Finally, note that this method to connect functions implemented in Haskell to Curry programs provides the opportunity to connect also operations written in other programming languages to Curry via Haskell's foreign function interface.

Index

<, 81
*., 47
+., 47
---, 27
-., 47
.kics2rc, 14
/., 47
//, 106
:!, 10
:&, 113
:add, 8
:browse, 10
:cd, 9
:edit, 9
:eval, 9
:fork, 10
:help, 8
:interface, 10
:load, 8
:programs, 9
:quit, 9
:reload, 8
:save, 10
:set, 10
:set path, 7
:show, 9
:source, 10
:type, 9
:usedimports, 10
@, 16
@author, 27
@cons, 27
@param, 27
@return, 27
@version, 27
KiCS2, 8
<*>, 81
<+>, 84
<//>, 84
</>, 84
<:, 43
<=:, 43
<$$>, 84
<$>, 84
<>, 84
>:, 43
>=:, 43
>>-, 79
>>>, 81
\\, 75

aBool, 150
abs, 61
AbstractCurry, 38
abstractCurryFileName, 161
aChar, 150
adapt, 150
addAttr, 134
addAttrs, 134
addCanvas, 59
addClass, 134
addCookies, 128
addDays, 104
addFormParam, 129
addHeadings, 132
addHours, 104
addListToFM, 108
addListToFM_C, 108
addMinutes, 104
addMonths, 104
addPageParam, 130
addRegionStyle, 59
address, 131
addSeconds, 104
addSound, 129
addToFM, 108
addToFM_C, 108
addYears, 104
aFloat, 150
aInt, 150
align, 83
allC, 43

allDBInfos, 73
allDBKeyInfos, 73
allDBKeys, 73
AllSolutions, 6
allValuesBFS, 99
allValuesDFS, 99
allValuesIDS, 99
allValuesIDSwith, 99
allVars, 184
allVarsInFunc, 179
allVarsInProg, 174
allVarsInRule, 180
alwaysRequired, 163
anchor, 131
andC, 43
angles, 88
answerEncText, 129
answerText, 129
anyC, 43
appendStyledValue, 59
appendValue, 59
applyAt, 106
args, 13
argTypes, 177
Array, 105
as-pattern, 16
assertEqual, 40
assertEqualIO, 40
assertIO, 40
Assertion, 40
assertSolutions, 40
assertTrue, 40
assertValues, 40
aString, 150
atan, 47
attr, 150

backslash, 89
baseName, 46
best, 6
bfs, 10
bfsStrategy, 99
bindings, 12
binomial, 61
bitAnd, 62
bitNot, 62
bitOr, 62
bitTrunc, 62
bitXor, 62
blink, 131
block, 133
blockstyle, 132
bold, 131
bquotes, 88
braces, 88
brackets, 88
BranchExpr, 171
branchExpr, 184
branchPattern, 184
breakline, 132
buildGr, 113
Button, 60
button, 133

CalendarTime, 103
calendarTimeToString, 104
CanvasItem, 55
CanvasScroll, 60
caseBranches, 182
caseExpr, 182
CaseType, 169
caseType, 182
cat, 86
categorizeByItemKey, 124
catMaybes, 79
CBranchExpr, 160
CConsDecl, 156
center, 131
CEvalAnnot, 158
CExpr, 158
CField, 155
CFixity, 157
CFuncDecl, 157
CgiEnv, 124
CgiRef, 124
char, 88, 149
checkAssertion, 41
checkbox, 133

checkedbox, 133
childFamilies, 122
children, 122
choices, 11
choose, 97
chooseColor, 61
chooseValue, 97
CLabel, 155
cleancurry, 5
cleanDB, 74
CLiteral, 160
CLocalDecl, 158
ClockTime, 103
clockTimeToInt, 104
closeDBHandles, 74
Cmd, 60
cmp, 12
cmpChar, 120
cmpList, 120
cmpString, 121
code, 131
col, 57
colon, 89
Color, 56
ColVal, 70
combArgs, 181
combine, 84, 106
combineSimilar, 106
combName, 181
CombType, 169
combType, 181
comma, 89
Command, 60
comment
    documentation, 27
compareCalendarTime, 105
compareClockTime, 105
compareDate, 105
compose, 84
computeCompactFlatCurry, 163
concurrency, 6
ConfCollection, 54
ConfigButton, 60
ConfItem, 50
connectToCommand, 66
connectToSocket, 80, 101
connectToSocketRepeat, 80
connectToSocketWait, 80
cons, 107
consArgs, 176
consArity, 176
ConsDecl, 167
consName, 176
consVisibility, 176
Context, 112
context, 114
Context', 112
cookieForm, 128
CookieParam, 127
coordinates, 135
COpDecl, 157
copyFile, 45
cos, 47
CPattern, 159
cprogDoc, 188
cprogDocWithPrecedences, 188
createDirectory, 45
createDirectoryIfMissing, 45
CRule, 158
CRules, 158
CStatement, 159
ctDay, 103
ctHour, 103
ctMin, 103
ctMonth, 103
ctSec, 103
ctTZ, 103
CTVarIName, 155
ctYear, 103
CTypeDecl, 156
CTypeExpr, 156
Curry mode, 14
CurryDoc, 27
currydoc, 28
CURRYPATH, 7, 34
CurryProg, 155
CurryTest, 32
currytest, 32

CVarIName, 155
CVisibility, 156
cycle, 78
cyclic structure, 15

database programming, 34
daysOfMonth, 105
debugTcl, 57
Decomp, 112
defaultEncoding, 128
defaultRequired, 163
deg, 115
deg', 116
delEdge, 113
delEdges, 114
delete, 75, 119
deleteBy, 75
deleteDBEntries, 74
deleteDBEntry, 74
deleteRBT, 120, 121
delFromFM, 108
delListFromFM, 109
delNode, 113
delNodes, 114
deqHead, 107
deqInit, 107
deqLast, 107
deqLength, 107
deqReverse, 107
deqTail, 107
deqToList, 107
dfs, 11
dfsStrategy, 99
digitToInt, 42
dirName, 46
dlist, 132
Doc, 82
documentation comment, 27
documentation generator, 27
doesDirectoryExist, 44
doesFileExist, 44
domain, 176
doneT, 72
dot, 89
dquote, 89
dquotes, 88
Dynamic, 70

eBool, 151
eChar, 151
Edge, 111
edges, 116
eEmpty, 151
eFloat, 151
eInt, 150
element, 149
elemFM, 110
elemIndex, 75
elemIndices, 75
elemRBT, 119
elemsOf, 147
eltsFM, 110
Emacs, 14
emap, 117
emphasize, 130
empty, 81, 82, 107, 113, 118, 149
emptyDefaultArray, 106
emptyErrorArray, 106
emptyFM, 108
emptySetRBT, 119
emptyTableRBT, 121
encapsulated search, 6
enclose, 87
encloseSep, 86
Encoding, 146
entity relationship diagrams, 34
EntryScroll, 60
eOpt, 151
eqFM, 109
equal, 115
equals, 89
ERD2Curry, 34
erd2curry, 34
eRep, 151
eRepSeq1, 152
eRepSeq2, 152
eRepSeq3, 153
eRepSeq4, 153

eRepSeq5, 154
eRepSeq6, 154
errorT, 72
eSeq1, 151
eSeq2, 152
eSeq3, 152
eSeq4, 153
eSeq5, 154
eSeq6, 154
eString, 151
evalChildFamilies, 123
evalChildFamiliesIO, 123
evalCmd, 66
evalFamily, 122
evalFamilyIO, 123
even, 62
Event, 53
exclusiveIO, 66
execCmd, 66
existsDBKey, 73
exitGUI, 58
exitWith, 102
exp, 47
expires, 129
Expr, 169
external operation, 194

factorial, 61
failT, 72
family, 122
fileSize, 44
fileSuffix, 46
fillCat, 86
fillEncloseSep, 87
fillSep, 85
filterFM, 109
find, 75
findall, 6
findIndex, 75
findIndices, 75
first, 12
Fixity, 168
FlatCurry, 38
flatCurry2Xml, 187
flatCurry2XmlFile, 187
flatCurryFileName, 172
flatCurryIntName, 172
FlexRigidResult, 187
float, 88, 149
FM, 108
fmSortBy, 110
fmToList, 110
fmToListPreOrder, 110
focusInput, 59
fold, 123
foldChildren, 123
foldFM, 109
foldValues, 97
form, 128
formatMarkdownFileAsPDF, 140
formatMarkdownInputAsPDF, 140
formBodyAttr, 128
formCSS, 128
formEnc, 128
FormParam, 125
freeExpr, 182
freeVars, 182
fromJust, 78
fromMarkdownText, 139
fromMaybe, 79
funcArgs, 179
funcArity, 178
funcBody, 179
FuncDecl, 168
funcName, 178
funcRHS, 179
funcRule, 179
functional pattern, 15
funcType, 178
funcVisibility, 178

garbageCollect, 91
garbageCollectorOff, 91
garbageCollectorOn, 91
GDecomp, 112
gelem, 115
generateCompactFlatCurryFile, 163
germanLatexDoc, 136

getAllFailures, 39
getAllSolutions, 39
getAllValues, 39
getAllValuesWith, 100
getArgs, 101
getAssoc, 66
getClockTime, 103
getContents, 65
getContentsOfUrl, 146
getCookies, 135
getCPUTime, 101
getCurrentDirectory, 45
getCursorPosition, 59
getDB, 72
getDBInfo, 74
getDBInfos, 74
getDirectoryContents, 45
getElapsedTime, 101
getEnviron, 101
getFileInPath, 46
getFlexRigid, 187
getHomeDirectory, 45
getHostname, 102
getLocalTime, 103
getModificationTime, 45
getOneSolution, 39
getOneValue, 39
getOpenFile, 60
getOpenFileWithTypes, 60
getPID, 102
getProcessInfos, 91
getProgName, 102
getRandomSeed, 118
getSaveFile, 61
getSaveFileWithTypes, 61
getSearchTree, 99
getTemporaryDirectory, 45
getUrlParameter, 135
getValue, 58
ghc, 11, 12
ghci, 12
Global, 48
global, 48
global installation, 5
GlobalSpec, 48
gmap, 116
Graph, 112
group, 76, 83
groupBy, 76
GuiPort, 49

h1, 130
h2, 130
h3, 130
h4, 130
h5, 130
Handle, 63
hang, 83
hcat, 85
hClose, 63
headedTable, 132
hempty, 130
hEncloseSep, 87
hFlush, 64
hGetChar, 65
hGetContents, 65
hGetLine, 65
hiddenfield, 134
hIsEOF, 64
hIsReadable, 65
hIsWritable, 65
hPrint, 65
hPutChar, 65
hPutStr, 65
hPutStrLn, 65
hReady, 64
href, 131
hrule, 132
hSeek, 64
hsep, 84
HtmlExp, 124
HtmlForm, 125
HtmlHandler, 124
htmlIsoUmlauts, 134
HtmlPage, 127
htmlQuote, 134
htmlSpecialChars2tex, 136
htxt, 130

htxts, 130
hWaitForInput, 64
hWaitForInputOrMsg, 64
hWaitForInputs, 64
hWaitForInputsOrMsg, 64

i2f, 47
idOfCgiRef, 128
ids, 11
idsStrategy, 99
idsStrategyWith, 99
ilog, 61
image, 132
imageButton, 133
indeg, 115
indeg', 116
init, 77
inits, 76
inline, 133
inn, 115
inn', 116
insEdge, 113
insEdges, 114
insertBy, 77
insertMultiRBT, 119
insertRBT, 119
insNode, 113
insNodes, 113
installation
  global, 5
  local, 5
int, 88, 149
integer, 11
interactive, 8
interactive, 12
intercalate, 76
intersect, 75
intersectFM, 109
intersectFM_C, 109
intersectRBT, 120
intersperse, 76
intForm, 136
intFormMain, 136
intToDigit, 42
IOMode, 63
IORef, 65
ioref, 11
isAbsolute, 46
isAlpha, 41
isAlphaNum, 42
isBigComment, 165
isCase, 182
isCode, 165
isComb, 182
isCombTypeConsCall, 181
isCombTypeConsPartCall, 181
isCombTypeFuncCall, 181
isCombTypeFuncPartCall, 181
isComment, 165
isConsCall, 183
isConsPartCall, 183
isConsPattern, 184
isDefined, 99
isDigit, 42
isEmpty, 82, 96, 107, 114, 118
isEmptyFM, 110
isEmptySetRBT, 119
isEmptyTable, 121
isEOF, 64
isExternal, 179
isFree, 182
isFuncCall, 183
isFuncPartCall, 183
isFuncType, 177
isGround, 183
isHexDigit, 42
isInfixOf, 77
isJust, 78
isLet, 182
isLetter, 165
isLit, 182
isLower, 41
isMeta, 165
isModuleHead, 165
isNothing, 78
isOctDigit, 42
isOr, 182
isPosix, 102

isPrefixOf, 77
isqrt, 61
isRuleExternal, 180
isSmallComment, 165
isSpace, 42
isSuffixOf, 77
isTCons, 177
isText, 165
isTVar, 177
isTypeSyn, 175
isUpper, 41
isVar, 182
isWindows, 102
italic, 131

JSBranch, 69
jsConsTerm, 69
JSExp, 67
JSFDecl, 69
JSStat, 68

keyOrder, 110
keysFM, 110
kics2, 8
kics2rc, 14

lab, 114
lab', 115
labEdges, 116
labNode', 115
labNodes, 116
labUEdges, 117
labUNodes, 117
langle, 89
last, 77
lbrace, 89
lbracket, 89
LEdge, 111
leqChar, 120
leqCharIgnoreCase, 121
leqLexGerman, 121
leqList, 120
leqString, 121
leqStringIgnoreCase, 121
let, 15
letBinds, 181
letBody, 181
line, 82
linebreak, 82
linesep, 82
list, 87
list2CategorizedHtml, 124
ListBoxScroll, 60
listenOn, 80, 100
listenOnFresh, 100
listToDefaultArray, 106
listToDeq, 107
listToErrorArray, 106
listToFM, 108
listToMaybe, 79
litem, 132
Literal, 172
literal, 181
LNode, 111
local installation, 5
log, 47
lookup, 118
lookupFileInPath, 46
lookupFM, 110
lookupRBT, 121
lookupWithDefaultFM, 110
lparen, 88
LPath, 112
lpre, 115
lpre', 116
lsuc, 114
lsuc', 116

MailOption, 137
mainWUI, 145
mapAccumL, 78
mapAccumR, 78
mapChildFamilies, 122
mapChildFamiliesIO, 123
mapChildren, 122
mapChildrenIO, 123
mapFamily, 122
mapFamilyIO, 123
mapFM, 109

mapMaybe, 79
mapMMaybe, 79
mapT, 72
mapT_, 73
mapValues, 97
markdown, 27
MarkdownDoc, 138
MarkdownElem, 138
markdownText2CompleteHTML, 140
markdownText2CompleteLaTeX, 140
markdownText2HTML, 140
markdownText2LaTeX, 140
markdownText2LaTeXWithFormat, 140
match, 114
matchAny, 113
matchHead, 107
matchLast, 107
matrix, 57
max3, 62
maxFM, 110
maximum, 77
maxlist, 62
maxValue, 98
maybeToList, 79
MContext, 112
MenuItem, 54
mergeSort, 120
min3, 62
minFM, 110
minimum, 77
minlist, 62
minusFM, 109
minValue, 98
missingArgs, 181
missingCombArgs, 181
mkGraph, 113
mkUGraph, 113
modifyIORef, 67
modules, 7
multipleSelection, 134

nbsp, 130
neighbors, 114
neighbors', 115
nest, 83
newDBEntry, 74
newDBKeyEntry, 74
newIORef, 66
newNodes, 116
newTreeLike, 118
nextBoolean, 118
nextInt, 117
nextIntRange, 117
nmap, 117
noChildren, 122
Node, 111
node', 115
nodeRange, 114
nodes, 116
noindex, 29
noNodes, 114
nub, 75
nubBy, 75

odd, 62
olist, 131
once, 6
onlyindex, 29
OpDecl, 167
openFile, 63
operation
    external, 194
opFixity, 178
opName, 178
opPrecedence, 178
opt, 150
optimize, 12
Option, 162
option
    in source file, 13
orC, 43
orExps, 182
out, 115
out', 116
outdeg, 115
outdeg', 116

page, 130

pageBodyAttr, 129
pageCSS, 129
pageEnc, 129
pageLinkInfo, 129
pageMetaInfo, 129
PageParam, 127
par, 11, 130
parens, 88
parseHtmlString, 137
Parser, 81
ParserRep, 81
parseXmlString, 148
partition, 43, 76
password, 133
patArgs, 184
patCons, 184
patExpr, 185
Path, 112
path, 7, 10
pathSeparatorChar, 46
patLiteral, 184
Pattern, 171
pattern
    functional, 15
permutations, 76
permute, 42
persistentSQLite, 73
plainCode, 165
plusFM, 109
plusFM_C, 109
popup_message, 59
pow, 61
prdfs, 11
pre, 114, 131
pre', 115
precs, 188
preludePrecs, 188
pretty, 89
prettyCOps, 188
prettyCProg, 188
prettyCTypeExpr, 188
prettyCTypes, 188
printCProg, 188
printMemInfo, 91
printUCProg, 188
printValues, 98
ProcessInfo, 90
product, 77
profileSpace, 91
profileSpaceNF, 91
profileTime, 91
profileTimeNF, 91
Prog, 166
progFuncs, 173
progImports, 173
progName, 173
progOps, 174
program
    documentation, 27
    testing, 32
progTypes, 173
prompt, 12
ProtocolMsg, 40
punctuate, 86
pureio, 11

QName, 155, 166
Query, 70
Queue, 106
quickSort, 120

radio_main, 133
radio_main_off, 134
radio_other, 134
range, 177
rangle, 89
rbrace, 89
rbracket, 89
readAbstractCurryFile, 161
readCompleteFile, 66
readCSV, 44
readCSVFile, 44
readCSVFileWithDelims, 44
readCSVWithDelims, 44
readCurry, 38, 160
readCurryWithParseOptions, 160
readFileWithXmlDocs, 148
readFlatCurry, 38, 172

readFlatCurryFile, 172
readFlatCurryInt, 172
readFlatCurryIntWithImports, 185
readFlatCurryIntWithImportsInPath, 185
readFlatCurryWithImports, 185
readFlatCurryWithImportsInPath, 185
readFlatCurryWithParseOptions, 172
readFM, 111
readGlobal, 48
readHex, 92, 93
readHtmlFile, 137
readInt, 92
readIORef, 67
readNat, 92
readOct, 93
readPropertyFile, 92
readQTerm, 94
readQTermFile, 94
readQTermListFile, 94
readScan, 165
readsQTerm, 94
readsTerm, 94
readsUnqualifiedTerm, 93
readTerm, 94
readUnqualifiedTerm, 94
readUnsafeXmlFile, 148
readUntypedCurry, 160
readUntypedCurryWithParseOptions, 160
readXmlFile, 148
ReconfigureItem, 52
RedBlackTree, 118
redirect, 129
removeDirectory, 45
removeEscapes, 139
removeFile, 45
removeRegionStyle, 59
renameDirectory, 45
renameFile, 45
Rendering, 140
renderList, 145
renderTaggedTuple, 145
renderTuple, 145
rep, 150
replace, 77
replaceChildren, 122
replaceChildrenIO, 123
repSeq1, 151
repSeq2, 152
repSeq3, 152
repSeq4, 153
repSeq5, 153
repSeq6, 154
RequiredSpec, 163
requires, 163
resetbutton, 133
resultType, 177
returnT, 72
rnmAllVars, 184
rnmAllVarsInFunc, 179
rnmAllVarsInProg, 174
rnmAllVarsInRule, 180
rnmAllVarsInTypeExpr, 177
rnmProg, 174
rotate, 107
round, 47
row, 57
rparen, 88
rts, 13
Rule, 168
ruleArgs, 180
ruleBody, 180
ruleExtDecl, 180
runConfigControlledGUI, 57
runControlledGUI, 57
runFormServerWithKey, 135
runFormServerWithKeyAndFormParams, 135
runGUI, 57
runGUIwithParams, 57
runHandlesControlledGUI, 58
runInitControlledGUI, 58
runInitGUI, 57
runInitGUIwithParams, 57
runInitHandlesControlledGUI, 58
runJustT, 72
runPassiveGUI, 57
runQ, 71
runT, 71

satisfy, 81
scan, 165
scanl, 78
scanl1, 78
scanr, 78
scanr1, 78
sClose, 80, 101
SearchTree, 6, 98
searchTreeSize, 99
SeekMode, 63
seeText, 59
select, 97
selection, 134
selectionInitial, 134
selectValue, 97
semi, 89
semiBraces, 87
sendMail, 138
sendMailWithOptions, 138
sep, 85
separatorChar, 46
seq1, 151
seq2, 152
seq3, 152
seq4, 153
seq5, 153
seq6, 154
seqStrActions, 41
sequenceMaybe, 79
sequenceT, 72
sequenceT_, 72
set functions, 6
set0, 95
set0With, 95
set1, 95
set1With, 95
set2, 95
set2With, 95
set3, 96
set3With, 96
set4, 96
set4With, 96
set5, 96
set5With, 96
set6, 96
set6With, 96
set7, 96
set7With, 96
setAssoc, 66
setConfig, 58
setCurrentDirectory, 45
setEnviron, 101
setInsertEquivalence, 119
SetRBT, 119
setRBT2list, 120
setValue, 58
showCProg, 188
showCSV, 44
showCurryExpr, 186
showCurryId, 186
showCurryType, 186
showCurryVar, 186
showExpr, 162
showFlatFunc, 186
showFlatProg, 186
showFlatType, 186
showFM, 111
showFuncDecl, 162
showGraph, 117
showHtmlExp, 135
showHtmlExps, 135
showHtmlPage, 135
showJSExp, 69
showJSFDecl, 69
showJSStat, 69
showLatexDoc, 136
showLatexDocs, 136
showLatexDocsWithPackages, 136
showLatexDocWithPackages, 136
showLatexExp, 136
showLatexExps, 136
showMemInfo, 91
showPattern, 162
showProg, 161
showQNameInModule, 173
showQTerm, 93
showSearchTree, 99
showTerm, 93

showTError, 74
showTestCase, 41
showTestCompileError, 41
showTestEnd, 41
showTestMod, 41
showTypeDecl, 162
showTypeDecls, 161
showTypeExpr, 162
showXmlDoc, 147
showXmlDocWithParams, 147
sin, 47
singleton variables, 6
sizedSubset, 43
sizeFM, 109
sleep, 102
snoc, 107
Socket, 79, 100
socketAccept, 80, 101
socketName, 80
softbreak, 82
softline, 82
solveAll, 6
some, 81
someDBInfos, 73
someDBKeyInfos, 73
someDBKeyProjections, 74
someDBKeys, 73
someSearchTree, 99
someValue, 100
someValueWith, 100
sort, 119
sortBy, 77
sortRBT, 120
sortValues, 98
sortValuesBy, 98
source-file option, 13
space, 89
split, 76
splitBaseName, 46
splitDirectoryBaseName, 46
splitFM, 109
splitOn, 76
splitPath, 46
splitSet, 43
sqrt, 47
squote, 89
squotes, 88
standardForm, 128
standardPage, 130
star, 81
stderr, 63
stdin, 63
stdout, 63
Strategy, 98
string, 88, 149
string2urlencoded, 135
stringList2ItemList, 124
stripSuffix, 46
strong, 131
Style, 55
style, 132
styleSheet, 132
subset, 42
suc, 114
suc', 115
suffixSeparatorChar, 46
sum, 77
supply, 11
system, 102

table, 132
TableRBT, 121
tableRBT2list, 121
tabulator stops, 6
tagOf, 147
tails, 77
tan, 47
tConsArgs, 177
tConsName, 177
teletype, 131
terminal, 81
TError, 70
TErrorKind, 71
testing programs, 32
testScan, 165
text, 82
textarea, 133
TextEditScroll, 60

textfield, 133
textOf, 147
textOfXml, 147
textstyle, 132
time, 12
toCalendarTime, 104
toClockTime, 104
toDayString, 104
Token, 164
Tokens, 164
toLower, 42
toTimeString, 104
toUpper, 42
toUTCTime, 104
trace, 105
Transaction, 70
transformQ, 71
transformWSpec, 141
transpose, 76
Traversable, 122
trBranch, 184
trCombType, 181
trCons, 176
tree2list, 119
trExpr, 182
trFunc, 178
trOp, 178
trPattern, 184
trProg, 173
trRule, 180
trType, 174
trTypeExpr, 177
truncate, 47
try, 6
tryReadACYFile, 161
tupled, 87
TVarIndex, 166
tVarIndex, 176
typeConsDecls, 175
TypeDecl, 167
TypeExpr, 167
typeName, 175
typeParams, 175
typeSyn, 175
typeVisibility, 175

UContext, 112
UDecomp, 112
UEdge, 111
ufold, 116
UGr, 112
ulist, 131
unfoldr, 78
union, 75
unionRBT, 120
unitFM, 108
UNode, 111
unsafePerformIO, 105
unscan, 165
unsetEnviron, 102
untypedAbstractCurryFileName, 161
UPath, 112
Update, 173
update, 106, 118
updateDBEntry, 74
updateFile, 66
updatePropertyFile, 92
updateRBT, 121
updateValue, 58
updateXmlFile, 148
updBranch, 184
updBranches, 183
updBranchExpr, 184
updBranchPattern, 184
updCases, 183
updCombs, 183
updCons, 176
updConsArgs, 176
updConsArity, 176
updConsName, 176
updConsVisibility, 176
updFM, 109
updFrees, 183
updFunc, 179
updFuncArgs, 180
updFuncArity, 179
updFuncBody, 180
updFuncName, 179

updFuncRule, 179
updFuncType, 179
updFuncTypes, 177
updFuncVisibility, 179
updLets, 183
updLiterals, 183
updOp, 178
updOpFixity, 178
updOpName, 178
updOpPrecedence, 178
updOrs, 183
updPatArgs, 185
updPatCons, 185
updPatLiteral, 185
updPattern, 185
updProg, 174
updProgExps, 174
updProgFuncs, 174
updProgImports, 174
updProgName, 174
updProgOps, 174
updProgTypes, 174
updQNames, 184
updQNamesInConsDecl, 176
updQNamesInFunc, 179
updQNamesInProg, 174
updQNamesInRule, 180
updQNamesInType, 175
updQNamesInTypeExpr, 177
updRule, 180
updRuleArgs, 180
updRuleBody, 180
updRuleExtDecl, 180
updTCons, 177
updTVars, 177
updType, 175
updTypeConsDecls, 175
updTypeds, 183
updTypeName, 175
updTypeParams, 175
updTypeSynonym, 175
updTypeVisibility, 175
updVars, 183
urlencoded2string, 135

v, 11
validDate, 105
valueOf, 96
Values, 95
values2list, 98
variables
    singleton, 6
VarIndex, 166
varNr, 181
vcat, 85
verbatim, 131
verbosity, 11
Visibility, 166
vsep, 85

w10Tuple, 144
w11Tuple, 144
w12Tuple, 144
w4Tuple, 143
w5Tuple, 143
w6Tuple, 143
w7Tuple, 143
w8Tuple, 143
w9Tuple, 143
waitForSocketAccept, 80, 101
wCheckBool, 142
wCheckMaybe, 144
wConstant, 142
wEither, 145
where, 15
wHidden, 141
wHList, 144
Widget, 49
WidgetRef, 55
wInt, 142
withCondition, 141
withError, 141
withRendering, 141
wJoinTuple, 144
wList, 144
wListWithHeadings, 144
wMatrix, 144
wMaybe, 144
wMultiCheckSelect, 143

wPair, 143
wRadioBool, 143
wRadioMaybe, 145
wRadioSelect, 143
wRequiredString, 142
wRequiredStringSize, 142
writeAbstractCurryFile, 161
writeAssertResult, 41
writeCSVFile, 44
writeFCY, 173
writeGlobal, 48
writeIORef, 67
writeQTermFile, 94
writeQTermListFile, 94
writeXmlFile, 147
writeXmlFileWithParams, 147
wSelect, 142
wSelectBool, 142
wSelectInt, 142
wString, 142
wStringSize, 142
wTextArea, 142
WTree, 141
wTree, 145
wTriple, 143
wui2html, 145
WuiHandler, 140
wuiHandler2button, 141
wuiInForm, 145
WuiSpec, 141
wuiWithErrorForm, 145

XAttrConv, 148
XElemConv, 148
xml, 147
xml2FlatCurry, 187
XmlDocParams, 146
XmlExp, 146
xmlFile2FlatCurry, 187
xmlRead, 149
XmlReads, 148
xmlReads, 149
xmlShow, 149
XmlShows, 148
xmlShows, 149
XOptConv, 148
XPrimConv, 148
XRepConv, 148
xtxt, 147