# Contents

Preface 5

1 Overview of PAKCS 6
   1.1 General Use 6
   1.2 Restrictions 6
   1.3 Modules in PAKCS 7

2 PAKCS: An Interactive Curry Development System 8
   2.1 Invoking PAKCS 8
   2.2 Commands of PAKCS 9
   2.3 Options of PAKCS 11
   2.4 Using PAKCS in Batch Mode 14
   2.5 Command Line Editing 15
   2.6 Customization 15
   2.7 Emacs Interface 15

3 Extensions 16
   3.1 Recursive Variable Bindings 16
   3.2 Functional Patterns 16
   3.3 Order of Pattern Matching 18
   3.4 Type Classes 19

4 Recognized Syntax of Curry 20
   4.1 Notational Conventions 20
   4.2 Lexicon 20
      4.2.1 Comments 20
      4.2.2 Identifiers and Keywords 20
      4.2.3 Numeric and Character Literals 21
   4.3 Layout 22
   4.4 Context-Free Grammar 23

5 Optimization of Curry Programs 27

6 cypm: The Curry Package Manager 28

7 CurryCheck: A Tool for Testing Properties of Curry Programs 29
   7.1 Installation 29
   7.2 Testing Properties 29
   7.3 Generating Test Data 33
   7.4 Checking Equivalence of Operations 36
   7.5 Checking Contracts and Specifications 38
   7.6 Combining Testing and Verification 39
   7.7 Checking Usage of Specific Operations 39
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>CurryBrowser: A Tool for Analyzing and Browsing Curry Programs</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>8.1 Installation</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>8.2 Basic Usage</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>curry-doc: A Documentation Generator for Curry Programs</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>9.1 Installation</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>9.2 Documentation Comments</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>9.3 Generating Documentation</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>curry-style: A Style Checker for Curry Programs</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>10.1 Installation</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>10.2 Basic Usage</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>10.3 Configuration</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>CurryVerify: A Tool to Support the Verification of Curry Programs</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>11.1 Installation</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>11.2 Basic Usage</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>11.3 Options</td>
<td>50</td>
</tr>
<tr>
<td>12</td>
<td>CurryPP: A Preprocessor for Curry Programs</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>12.1 Installation</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>12.2 Basic Usage</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>12.3 Integrated Code</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>12.3.1 Regular Expressions</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>12.3.2 Format Specifications</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>12.3.3 HTML Code</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>12.3.4 XML Expressions</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>12.4 SQL Statements</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>12.4.1 ER Specifications</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>12.4.2 SQL Statements as Integrated Code</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>12.5 Default Rules</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>12.6 Contracts</td>
<td>62</td>
</tr>
<tr>
<td>13</td>
<td>runcurry: Running Curry Programs</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>13.1 Installation</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>13.2 Using runcurry</td>
<td>65</td>
</tr>
<tr>
<td>14</td>
<td>CASS: A Generic Curry Analysis Server System</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>14.1 Installation</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>14.2 Using CASS to Analyze Programs</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>14.2.1 Batch Mode</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>14.2.2 API Mode</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>14.2.3 Server Mode</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>14.3 Implementing Program Analyses</td>
<td>72</td>
</tr>
</tbody>
</table>
Preface

This document describes PAKCS (formerly called “PACS”), an implementation of the multi-paradigm language Curry, jointly developed at the University of Kiel, the Technical University of Aachen and Portland State University. 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). Moreover, the PAKCS implementation of Curry also supports constraint programming over various constraint domains, the high-level implementation of distributed applications, graphical user interfaces, and web services (as described in more detail in [19, 20, 21]). Since PAKCS compiles Curry programs into Prolog programs, the availability of some of these features might depend on the underlying Prolog system.

We assume familiarity with the ideas and features of Curry as described in the Curry language definition [28]. Therefore, this document only explains the use of the different components of PAKCS and the differences and restrictions of PAKCS (see Section 1.2) compared with the language Curry (Version 0.9.0).

Important Note

This version of PAKCS implements type classes. The concept of type classes is not yet part of the Curry language report. The recognized syntax of type classes is specified in Section 4. Although the implemented concept of type classes is not fully described in this manual, it is quite similar to Haskell 98 [34] so that one can look there to find a detailed description.

Acknowledgements

This work has been supported in part by the DAAD/NSF grant INT-9981317, the NSF grants CCR-0110496 and CCR-0218224, the Acción Integrada hispano-alemana HA1997-0073, and the DFG grants Ha 2457/1-2, Ha 2457/5-1, and Ha 2457/5-2.

Many thanks to the users of PAKCS for bug reports, bug fixes, and improvements, in particular, to Marco Comini, Sebastian Fischer, Massimo Forni, Carsten Heine, Stefan Junge, Frank Huch, Parissa Sadeghi.
1 Overview of PAKCS

1.1 General Use

This version of PAKCS has been tested on Sun Solaris, Linux, and Mac OS X systems. In principle, it should be also executable on other platforms on which a Prolog system like SICStus-Prolog or SWI-Prolog exists (see the file INSTALL.html in the PAKCS directory for a description of the necessary software to install PAKCS).

All executable files required to use the different components of PAKCS are stored in the directory pakcshome/bin (where pakcshome is the installation directory of the complete PAKCS installation). You should add this directory to your path (e.g., by the bash command “export PATH=pakcshome/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 PAKCS creates some auxiliary files (see Section E for details), you need write permission in the directory where you have stored your Curry programs. The auxiliary files for all Curry programs in the current directory can be deleted by the command cleancurry

(this is a shell script stored in the bin directory of the PAKCS installation, see above). The command cleancurry -r
also deletes the auxiliary files in all subdirectories.

1.2 Restrictions

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

- 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.

- PAKCS translates all local declarations into global functions with additional arguments (“lambda lifting”, see Appendix D of the Curry language report). Thus, in the compiled target code, the definition of functions with local declarations look different from their original definition (in order to see the result of this transformation, you can use the CurryBrowser, see Section 8).

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

- Since PAKCS compiles Curry programs into Prolog programs, non-deterministic computations are treated as in Prolog by a backtracking strategy, which is known to be incomplete. Thus, the order of rules could influence the ability to find solutions for a given goal.

- Threads created by a concurrent conjunction are not executed in a fair manner (usually, threads corresponding to leftmost constraints are executed with higher priority).
• Encapsulated search: In order to allow the integration of non-deterministic computations in
programs performing I/O at the top-level, PAKCS supports the search operators findall and findfirst. Note that they are not part of the standard prelude but these and some other operators are available in the library Control.Findall which is part of the package searchtree. In contrast to the general definition of encapsulated search [27], the current implementation suspends the evaluation of findall and findfirst until the argument does not contain unbound global variables. Moreover, the evaluation of findall is strict, i.e., it computes all solutions before returning the complete list of solutions. Since it is known that the result of these search operators might depend on the evaluation strategy due to the combination of sharing and lazy evaluation (see [14] for a detailed discussion), it is recommended to use set functions [7] as a strategy-independent encapsulation of non-deterministic computations. Set functions compute the set of all results of a defined function but do not encapsulate non-determinism occurring in the actual arguments. See the library Control.SetFunctions (available in package setfunctions) for more details.

• There is currently no general connection to external constraint solvers. However, the PAKCS compiler provides constraint solvers for arithmetic and finite domain constraints (see Appendix A).

1.3 Modules in PAKCS

PAKCS searches for imported modules in various directories. By default, imported modules are searched in the directory of the main program and the system module directory “pakcshome/lib”. This search path can be extended by setting the environment variable CURRYPATH (which can be also set in a PAKCS 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 “.pakcsrc” 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 “.pakcsrc” variable “libraries”.
4. The directory “pakcshome/lib”.

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

Note that the standard prelude (pakcshome/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 PAKCS: An Interactive Curry Development System

PAKCS is an interactive system to develop applications written in Curry. It is implemented in Prolog and compiles Curry programs into Prolog programs. It contains various tools, a source-level debugger, solvers for arithmetic constraints over real numbers and finite domain constraints, etc. The compilation process and the execution of compiled programs is fairly efficient if a good Prolog implementation like SICStus-Prolog is used.

2.1 Invoking PAKCS

To start PAKCS, execute the command “pakcs” or “curry” (these are shell scripts stored in \texttt{pakcshome/bin} where \texttt{pakcshome} is the installation directory of PAKCS). When the system is ready (i.e., when the prompt “Prelude>” occurs), the prelude (\texttt{pakcshome/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 PAKCS 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

\texttt{pakcs :load Mod :add List}

starts PAKCS, loads the main module \texttt{Mod}, and adds the additional module \texttt{List}. The invocation

\texttt{pakcs :load Mod :eval config}

starts PAKCS, loads the main module \texttt{Mod}, and evaluates the operation \texttt{config} before the user interaction starts. As a final example, the invocation

\texttt{pakcs :load Mod :save :quit}

starts PAKCS, loads the main module \texttt{Mod}, creates an executable, and terminates PAKCS. This invocation could be useful in “make” files for systems implemented in Curry.

There are also some additional options that can be used when invoking PAKCS:

\texttt{-h} or \texttt{--help} : Print only a help message.

\texttt{-V} or \texttt{--version} : Print the version information of PAKCS and quit.

\texttt{--compiler-name} : Print just the compiler name (\texttt{pakcs}) and quit.

\texttt{--numeric-version} : Print just the version number and quit.

\texttt{--noredline} : Do not use input line editing (see Section 2.5).

\texttt{-Dname=val} (these options must come before any PAKCS command): Overwrite values defined in the configuration file “\texttt{.pakcsrc}” (see Section 2.6), where \texttt{name} is a property defined in the configuration file and \texttt{val} its new value.

\texttt{-q} or \texttt{--quiet} : With this option, PAKCS works silently, i.e., the initial banner and the input prompt are not shown. The output of other information is determined by the options “\texttt{verbose}” and “\texttt{vn}” (see Section 2.3).
One can also invoke PAKCS with some run-time arguments that can be accessed inside a Curry program by the I/O operation `getArgs` (see library `System` (Section A.2.16). These run-time arguments must be written at the end after the separator “--”. For instance, if PAKCS is invoked by

```plaintext
pakcs :load Mod -- first and second
```

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

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

### 2.2 Commands of PAKCS

The **most important commands** of PAKCS 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. If this file does not exist, the system looks for a FlatCurry file `prog.fcy` and compiles from this intermediate representation.
- **:reload** Recompile all currently loaded modules.
- **:add m₁...mₙ** Add modules `m₁,...,mₙ` to the set of currently loaded modules so that their exported entities are available in the top-level environment.
- **:eval expr** Same as `expr`. This command might be useful when putting commands as arguments when invoking `pakcs`.

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

```plaintext
not b where b free
```

in order to obtain the following bindings and results:

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

Without these declarations, an error is reported in order to avoid the unintended introduction of free variables in initial expressions by typos.
There are also a number of **further commands** that are often useful:

- **:type expr** Show the type of the expression `expr`.
- **:browse** Start the CurryBrowser to analyze the currently loaded module together with all its imported modules (see Section 8 for more details).
- **:edit** Load the source code of the current main module into a text editor. If the variable `editcommand` is set in the configuration file “.pakcsrc” (see Section 2.6), its value is used as an editor command, otherwise the environment variable “EDITOR” or a default editor (e.g., “vi”) is used.
- **:edit m** Load the source text of module `m` (which must be accessible via the current load path if no path specification is given) into a text editor which is defined as in the command “:edit”.
- **: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 prog** Similar to “:interface” but shows the interface of the module “prog.curry”. If this module does not exist, this command looks in the system library directory of PAKCS for a module with this name, e.g., the command “:interface FlatCurry” shows the interface of the system module FlatCurry for meta-programming (see Appendix A.1.4).
- **:usedimports** Show all calls to imported functions in the currently loaded module. This might be useful to see which import declarations are really necessary.
- **:modules** Show the list of all currently loaded modules.
- **:programs** Show the list of all Curry programs that are available in the load path.
- **:set option** Set or turn on/off a specific option of the PAKCS 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., `printdepth`) must not contain a prefix.
- **:set** Show a help text on the possible options together with the current values of all options.
- **:show** Show the source text of the currently loaded Curry program. If the variable `showcommand` is set in the configuration file “.pakcsrc” (see Section 2.6), its value is used as a command to show the source text, otherwise the environment variable `PAGER` or the standard command “cat” is used. If the source text is not available (since the program has been directly compiled from a FlatCurry file), the loaded program is decompiled and the decompiled Curry program text is shown.
- **:show m** Show the source text of module `m` which must be accessible via the current load path.
: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.

:cd dir  Change the current working directory to dir.

:dir  Show the names of all Curry programs in the current working directory.

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

: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 must be of type “IO ()”, is evaluated in an independent process which runs in parallel to the current PAKCS process. All output and error messages from this new process are suppressed. This command is useful to test distributed Curry programs (see Appendix A.1.3) 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.

:coosy  Start the Curry Object Observation System COOSy, a tool to observe the execution of Curry programs. This commands starts a graphical user interface to show the observation results and adds to the load path the directory containing the modules that must be imported in order to annotate a program with observation points. Details about the use of COOSy can be found in the COOSy interface (under the “Info” button), and details about the general idea of observation debugging and the implementation of COOSy can be found in [13].

:peval  Translate the currently loaded program module into an equivalent program where some subexpressions are partially evaluated so that these subexpressions are (hopefully) more efficiently executed. An expression e to be partially evaluated must be marked in the source program by (PEVAL e) (where PEVAL is defined as the identity function in the prelude so that it has no semantical meaning).

The partial evaluator translates a source program prog.curry into the partially evaluated program in intermediate representation stored in prog.pe.fcy. The latter program is implicitly loaded by the peval command so that the partially evaluated program is directly available. The corresponding source program can be shown by the show command (see above).

The current partial evaluator is an experimental prototype (so it might not work on all programs) based on the ideas described in [1, 2, 3, 4].

2.3 Options of PAKCS

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

+/-debug  Debug mode. In the debug mode, one can trace the evaluation of an expression, setting spy points (break points) etc. (see the commands for the debug mode described below).
**+/−printfail** Print failures. If this option is set, failures occurring during evaluation (i.e., non-reducible demanded subexpressions) are printed. This is useful to see failed reductions due to partially defined functions or failed unifications. Inside encapsulated search (e.g., inside evaluations of `findall` and `findfirst`), failures are not printed (since they are a typical programming technique there). Note that this option causes some overhead in execution time and memory so that it could not be used in larger applications.

**+/−allfails** If this option is set, all failures (i.e., also failures on backtracking and failures of enclosing functions that fail due to the failure of an argument evaluation) are printed if the option `printfail` is set. Otherwise, only the first failure (i.e., the first non-reducible subexpression) is printed.

**+/−consfail** Print constructor failures. If this option is set, failures due to application of functions with non-exhaustive pattern matching or failures during unification (application of “=:=” are shown. Inside encapsulated search (e.g., inside evaluations of `findall` and `findfirst`), failures are not printed (since they are a typical programming technique there). In contrast to the option `printfail`, this option creates only a small overhead in execution time and memory use.

**+consfail all** Similarly to “+consfail”, but the complete trace of all active (and just failed) function calls from the main function to the failed function are shown.

**+consfail file:f** Similarly to “+consfail all”, but the complete fail trace is stored in the file `f`. This option is useful in non-interactive program executions like web scripts.

**+consfail int** Similarly to “+consfail all”, but after each failure occurrence, an interactive mode for exploring the fail trace is started (see help information in this interactive mode). When the interactive mode is finished, the program execution proceeds with a failure.

**+/−compact** Reduce the size of target programs by using the parser option “--compact” (see Section 18 for details about this option).

**+/−interactive** Turn on/off the interactive mode. In the interactive mode, the next non-deterministic value is computed only when the user requests it. Thus, one has also the possibility to terminate the enumeration of all values after having seen some values. The default value for this option can be set in the configuration file “.pakcsrc” (initially, the interactive mode is turned off).

**+/−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).

**+/−profile** Profile mode. If the profile mode is on, then information about the number of calls, failures, exits etc. are collected for each function during the debug mode (see above) and shown after the complete execution (additionaly, the result is stored in the file `prog.profile` where `prog` is the current main program). The profile mode has no effect outside the debug mode.
Suspend mode (initially, it is off). If the suspend mode is on, all suspended expressions (if there are any) are shown (in their internal representation) at the end of a computation.

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.

Verbose mode (initially, it is off). If the verbose mode is on, the initial expression of a computation is printed before it is evaluated. If the verbose mode is on and the verbosity level (see below) is non-zero, the type of the initial expression is also printed and the output of the evaluation is more detailed.

Parser warnings. If the parser warnings are turned on (default), the parser will print warnings about variables that occur only once in a program rule (see Section 1.2) or locally declared names that shadow the definition of globally declared names. If the parser warnings are switched off, these warnings are not printed during the reading of a Curry program.

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 PAKCS, i.e., the environment variable “CURRYPATH” (see also Section 1.3) is set to path in this invocation of PAKCS.

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

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

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

Set the depth for printing terms to the value n (initially: 0). In this case subterms with a depth greater than n are abbreviated by dots when they are printed as a result of a computation or during debugging. A value of 0 means infinite depth so that the complete terms are printed.

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 loading intermediate files or generating Prolog target files.
- n = 3: Show also messages related to loading Prolog files and libraries into the run-time systems and other intermediate messages and results.

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

```plaintext
pakcs : set safe
```
parser opts  Define additional options passed to the front end of PAKCS, i.e., the parser program
pakcs-home/bin/pakcs-frontend. For instance, setting the option

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

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

args arguments  Define run-time arguments for the evaluation of the main expression. For instance, setting the option

`:set args first second`

has the effect that the I/O operation getArgs (see library System (Section A.2.16) returns the value ["first","second"].

PAKCS can also execute programs in the **debug mode**. The debug mode is switched on by setting the debug option with the command "`:set +debug". In order to switch back to normal evaluation of the program, one has to execute the command "`:set -debug".

In the debug mode, PAKCS offers the following additional options:

+/-single  Turn on/off single mode for debugging. If the single mode is on, the evaluation of an expression is stopped after each step and the user is asked how to proceed (see the options there).

+/-trace  Turn on/off trace mode for debugging. If the trace mode is on, all intermediate expressions occurring during the evaluation of an expressions are shown.

spy f  Set a spy point (break point) on the function f. In the single mode, you can “leap” from spy point to spy point (see the options shown in the single mode).

+/-spy  Turn on/off spy mode for debugging. If the spy mode is on, the single mode is automatically activated when a spy point is reached.

2.4 Using PAKCS in Batch Mode

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

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

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

Here is an example to use PAKCS in this way:
2.5 Command Line Editing

In order to have support for line editing or history functionality in the command line of PAKCS (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 PAKCS 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 PAKCS with the parameter “--noreadline”.

2.6 Customization

In order to customize the behavior of PAKCS to your own preferences, there is a configuration file which is read by PAKCS when it is invoked. When you start PAKCS for the first time, a standard version of this configuration file is copied with the name “.pakcsrc” into your home directory. The file contains definitions of various settings, e.g., about showing warnings, progress messages etc. After you have started PAKCS 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 PAKCS 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 PAKCS 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 “pakcshome/tools/emacs” which also contains the sources of the Curry mode and a short description about the use of this mode.
3 Extensions

PAKCS 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 PAKCS. For instance, the declaration

\[
\text{ones5 = let ones = 1 : ones} \\
\hspace{1cm} \text{in take 5 ones}
\]

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

\[
\text{onetwo n = take n one2} \\
\text{where} \\
\text{one2 = 1 : two1} \\
\text{two1 = 2 : one2}
\]

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

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

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

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

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

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

This definition is not only more compact but also avoids the complete evaluation of the list elements: since a functional pattern is considered as an abbreviation for the set of constructor terms obtained by all evaluations of the functional pattern to normal form (see [6] for an exact definition), the previous definition is conceptually equivalent to the set of rules

\[
\text{last} \ [y] = y \\
\text{last} \ [_,y] = y \\
\text{last} \ [_,_,y] = y \\
\]

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

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

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

\[
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
\]

and is necessary to assign a semantics to functions employing functional patterns (see [6] for more details).

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

\[
eq :: a \to a \to Bool
\]
\[
eq x y = h y
\]
where
\[
g True = x
\]
\[
h (g a) = a
\]

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

\[
eq :: a \to a \to Bool
\]
\[
eq x y = h y
\]
where
\[
h x_1 \mid x \equiv x_1 = True
\]

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

It is also possible to combine functional patterns with as-patterns. Similarly to the meaning of as-patterns in standard constructor patterns, as-patterns in functional patterns are interpreted as a sequence of pattern matching where the variable of the as-pattern is matched before the given pattern is matched. This process can be described by introducing an auxiliary operation for this two-level pattern matching process. For instance, the definition
\[ f \ (\_ + x0[(42,\_)] + \_) = x \]

is considered as syntactic sugar for the expanded definition
\[
\begin{align*}
  f \ (\_ + x + \_) &= f' \ x \\
  \text{where} \\
  f' \ [(42,\_)] &= x
\end{align*}
\]

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

**Optimization of programs containing functional patterns.** Since functions patterns can evaluate to non-linear constructor terms, they are dynamically checked for multiple occurrences of variables which are, if present, replaced by equality constraints so that the constructor term is always linear (see [6] for details). Since these dynamic checks are costly and not necessary for functional patterns that are guaranteed to evaluate to linear terms, there is an optimizer for functional patterns that checks for occurrences of functional patterns that evaluate always to linear constructor terms and replace such occurrences with a more efficient implementation. This optimizer can be enabled by the following possibilities:

- Set the environment variable `FCYPP` to `"--fpopt"` before starting PAKCS, e.g., by the shell command
  \[
  \text{export FCYPP="--fpopt"}
  \]

  Then the functional pattern optimization is applied if programs are compiled and loaded in PAKCS.

- Put an option into the source code: If the source code of a program contains a line with a comment of the form (the comment must start at the beginning of the line)
  \[
  \{-# \ PAKCS\_OPTION\_FCYPP \ --fpopt \ #-\}
  \]

  then the functional pattern optimization is applied if this program is compiled and loaded in PAKCS.

The optimizer also report errors in case of wrong uses of functional patterns (i.e., in case of a function \( f \) defined with functional patterns that recursively depend on \( f \)).

### 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 Type Classes

The concept of type classes is not yet part of the Curry language report. The recognized syntax of type classes is specified in Section 4. Although the implemented concept of type classes is not fully described in this manual, it is quite similar to Haskell 98 [34] so that one can look there to find a detailed description.
4 Recognized Syntax of Curry

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

4.1 Notational Conventions

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

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

The Curry files are expected to be encoded in UTF8. However, source programs are biased towards ASCII for compatibility reasons.

4.2 Lexicon

4.2.1 Comments

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

4.2.2 Identifiers and Keywords

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

\[
\begin{align*}
\text{Letter} & ::= \text{any ASCII letter} \\
\text{Dashes} & ::= -- \{\text{-}\} \\
\text{Ident} & ::= (\text{Letter} \{\text{Letter} \mid \text{Digit} \mid _ \mid ^}\})_{\text{ReservedID}} \\
\text{Symbol} & ::= \sim \mid ! \mid \& \mid \# \mid \$ \mid \% \mid \_ \mid \& \mid \* \mid + \mid - \mid = \mid < \mid > \mid ? \mid . \mid / \mid \| \mid \backslash : \\
\text{ModuleID} & ::= \{\text{Ident} \_\} \text{Ident} \\
\text{ConstrID} & ::= \text{Ident} \\
\text{TypeVarID} & ::= \text{Ident} \mid _ \\
\text{ClassVarID} & ::= \text{Ident}
\end{align*}
\]
ExistVarID ::= Ident
DataConstrID ::= Ident
InfixOpID ::= (Symbol {Symbol})⟨Dashes | ReservedSym⟩
FunctionID ::= Ident
VariableID ::= Ident
LabelID ::= Ident
ClassID ::= Ident
QTypeConstrID ::= [ModuleID .| TypeConstrID
QDataConstrID ::= [ModuleID .| DataConstrID
QInfixOpID ::= [ModuleID .| InfixOpID
QFunctionID ::= [ModuleID .| FunctionID
QLabelID ::= [ModuleID .| LabelID
QClassID ::= [ModuleID .| ClassID

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

ReservedID ::= case | class | data | default | deriving | do | else | external
| fcase | free | if | import | in | infix | infix1 | infixr
| instance | let | module | newtype | of | then | type | where

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

ReservedSym ::= . | : | :: | = | \ | | < - | - > | @ | ~ | =>

4.2.3 Numeric and Character Literals

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

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

Float ::= Decimal . Decimal [Exponent]
| Decimal Exponent

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

Decimal ::= Digit {Digit}
Binary ::= Binit {Binit}
Octal ::= Octit {Octit}
Hexadecimal ::= Hexit {Hexit}

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

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

Char ::= ' (Graphic | \ Space | Escape )'
String ::= " { Graphic | \ Space | Escape | Gap }"
4.3 Layout

Similarly to Haskell, a Curry programmer can use layout information to define the structure of blocks. For this purpose, we define the indentation of a symbol as the column number indicating the start of this symbol, and the indentation of a line is the indentation of its first symbol.\(^1\)

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

\[
\begin{align*}
f \ x & = h \ x \ where \ \{ \ g \ y & = y + 1 \ ; \ h \ z & = (g \ z) \ * \ 2 \ \} \\
\end{align*}
\]

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

\[
\begin{align*}
f \ x & = h \ x \\
& \ where \ g \ y = y + 1 \\
& \ h \ z = (g \ z) \ * \ 2 \\
\end{align*}
\]

or also as

\[
\begin{align*}
f \ x & = h \ x \ where \\
& \ g \ y = y + 1 \\
& \ h \ z = (g \ z) \\
& \ * \ 2 \\
\end{align*}
\]

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

\(^1\)In order to determine the exact column number, we assume a fixed-width font with tab stops at each 8th column.
4.4 Context-Free Grammar

\[
Module ::= \text{module } ModuleID [Exports] \text{ where } Block
\]
\[
Block ::= \{ \text{ImportDecls ; BlockDecl}_1 ; \ldots ; \text{BlockDecl}_n \} \quad (\text{no fixity declarations here, } n \geq 0)
\]
\[
Exports ::= ( Export_1, \ldots, Export_n ) \quad (n \geq 0)
\]
\[
Export ::= \text{QFunction}
\]
\[
\text{ConsLabel} ::= \text{DataConstr} \mid \text{Label}
\]
\[
\text{ImportDecls} ::= \text{ImportDecl}_1 ; \ldots ; \text{ImportDecl}_n \quad (n \geq 1)
\]
\[
\text{ImportDecl} ::= \text{import [qualified] ModuleID [as ModuleID] [ImportSpec]}
\]
\[
\text{ImportSpec} ::= \text{hiding ( Import}_1, \ldots, \text{Import}_n \) \quad (n \geq 0)
\]
\[
\text{BlockDecl} ::= \text{TypeSynDecl} \mid \text{DataDecl} \mid \text{NewtypeDecl} \mid \text{FixityDecl} \mid \text{FunctionDecl} \mid \text{DefaultDecl} \mid \text{ClassDecl} \mid \text{InstanceDecl}
\]
\[
\text{TypeSynDecl} ::= \text{type } \text{SimpleType} = \text{TypeExpr}
\]
\[
\text{SimpleType} ::= \text{TypeConstrID TypeVarID}_1 \ldots \text{TypeVarID}_n \quad (n \geq 0)
\]
\[
\text{DataDecl} ::= \text{external data } \text{SimpleType} \quad (\text{external data type})
\]
\[
\text{ConstrDecls} ::= \text{ConstrDecl}_1 \mid \ldots \mid \text{ConstrDecl}_n \quad (n \geq 1)
\]
\[
\text{ConstrDecl} ::= \text{[ExistVars] [Context }\Rightarrow\text{] ConDecl}
\]
\[
\text{ExistVars} ::= \text{forall } \text{ExistVarID}_1 \ldots \text{ExistVarID}_n . \quad (n \geq 1)
\]
\[
\text{ConDecl} ::= \text{DataConstr SimpleTypeExpr}_1 \ldots \text{SimpleTypeExpr}_n \quad (n \geq 0)
\]
\[
\text{FieldDecl} ::= \text{Label}_1, \ldots, \text{Label}_n :: \text{TypeExpr} \quad (n \geq 0)
\]
\[
\text{DerivingDecl} ::= \langle \text{QClassID}_1, \ldots, \text{QClassID}_n \rangle \quad (n \geq 0)
\]
\[
\text{NewtypeDecl} ::= \text{newtype } \text{SimpleType} = \text{NewConstrDecl} [\text{deriving } \text{DerivingDecl}]
\]
\[
\text{NewConstrDecl} ::= \text{DataConstr SimpleTypeExpr}
\]
\[
\text{QualTypeExpr} ::= \text{[Context }\Rightarrow\text{] TypeExpr}
\]
\[
\text{Context} ::= \text{Constraint}
\]
Constraint ::= QClassID ClassVarID
  | QClassID ( ClassVarID SimpleTypeExpr1 ... SimpleTypeExprn ) (n ≥ 1)
  | ( Constraint1 , ..., Constraintn ) (n ≥ 0)

TypeExpr ::= TypeAppExpr [-> TypeExpr]
TypeAppExpr ::= [TypeAppExpr] SimpleTypeExpr
SimpleTypeExpr ::= TypeVarID
  | GTypeConstr
  | ( TypeExpr1 , ..., TypeExprn ) (tuple type, n ≥ 2)
  | [ TypeExpr ] (list type)
  | ( TypeExpr ) (parenthesized type)
GTypeConstr ::= () (unit type constructor)
  | [] (list type constructor)
  | (->) (function type constructor)
  | (, {, }) (tuple type constructor)

DefaultDecl ::= default ( TypeExpr1 , ..., TypeExprn ) (n ≥ 0)

ClassDecl ::= class [SimpleContext =>] ClassID ClassVarID [where ClsDecls]
ClsDecls ::= { ClsDecl1 ; ... ; ClsDecln } (n ≥ 0)
ClsDecl ::= Signature
  | Equat

SimpleContext ::= SimpleConstraint
  | ( SimpleConstraint1 , ..., SimpleConstraintn ) (n ≥ 0)
SimpleConstraint ::= QCclassID ClassVarID

InstanceDecl ::= instance [SimpleContext =>] QClassID InstType [where InstDecls]
InstDecls ::= { InstDecl1 ; ... ; InstDecln } (n ≥ 0)
InstDecl ::= Equat
InstType ::= GTypeConstr
  | ( GTypeConstr ClassVarID1 ... ClassVarIDn ) (n ≥ 0)
  | ( ClassVarID1 , ..., ClassVarIDn ) (n ≥ 2)
  | [ ClassVarID ]
  | ( ClassVarID -> ClassVarID )

FixityDecl ::= Fixity [Int] Op1 , ..., Opn (n ≥ 1)
Fixity ::= infixl | infixr | infix

FunctionDecl ::= Signature | ExternalDecl | Equation
Signature ::= Functions : QualTypeExpr
ExternalDecl ::= Functions external (externally defined functions)
Functions ::= Function1 , ..., Functionn (n ≥ 1)
Equation ::= FunLhs Rhs
FunLhs ::= Function SimplePat1 ... SimplePatn (n ≥ 0)
  | ConsPattern FunOp ConsPattern
  | ( FunLhs ) SimplePat1 ... SimplePatn
Rhs ::= = Expr [where LocalDecls]
  | CondExprs [where LocalDecls]
CondExprs ::= | InfixExpr = Expr [CondExprs]
LocalDecls ::= { LocalDecl1 ; ... ; LocalDecln } (n ≥ 0)
LocalDecl ::= FunctionDecl
<table>
<thead>
<tr>
<th>PatternDecl</th>
<th>Variable₁, ..., Variableₙ free (ₙ ≥ 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FixityDecl</td>
<td></td>
</tr>
</tbody>
</table>

PatternDecl ::= Pattern Rhs

Pattern ::= ConsPattern [QConOp Pattern] (infix constructor pattern)
ConsPattern ::= GDataConstr SimplePat₁ ... SimplePatₙ (constructor pattern, ₙ ≥ 1)
                | - (Int | Float) (negative pattern)
                | SimplePat
SimplePat ::= Variable
                | - (wildcard)
                | GDataConstr (constructor)
                | Literal (literal)
                | ( Pattern ) (parenthesized pattern)
                | ( Pattern₁ , ..., Patternₙ ) (tuple pattern, ₙ ≥ 2)
                | [ Pattern₁ , ..., Patternₙ ] (list pattern, ₙ ≥ 1)
                | Variable @ SimplePat (as-pattern)
                | ~ SimplePat (irrefutable pattern)
                | ( QFunction SimplePat₁ ... SimplePatₙ ) (functional pattern, ₙ ≥ 1)
                | ( ConsPattern QFunOp Pattern ) (infix functional pattern)
                | QDataConstr { FieldPat₁ , ..., FieldPatₙ } (labeled pattern, ₙ ≥ 0)

FieldPat ::= QLabel = Pattern

Expr ::= InfixExpr :: QualTypeExpr (expression with type signature)
        | InfixExpr
InfixExpr ::= NoOpExpr QOp InfixExpr (infix operator application)
        | - InfixExpr (unary minus)
        | NoOpExpr
NoOpExpr ::= \ SimplePat₁ ... SimplePatₙ -> Expr (lambda expression, ₙ ≥ 1)
            | let LocalDecls in Expr (let expression)
            | if Expr then Expr else Expr (conditional)
            | case Expr of { Alt₁ ; ... ; Altₙ } (case expression, ₙ ≥ 1)
            | fcase Expr of { Alt₁ ; ... ; Altₙ } (fcase expression, ₙ ≥ 1)
            | do { Stmt₁ ; ... ; Stmtₙ ; Expr } (do expression, ₙ ≥ 0)
            | FuncExpr
FuncExpr ::= [FuncExpr] BasicExpr (application)
BasicExpr ::= Variable (variable)
                | - (anonymous free variable)
                | GDataConstr (general constructor)
                | Literal (literal)
                | ( Expr ) (parenthesized expression)
                | ( Expr₁ , ..., Exprₙ ) (tuple, ₙ ≥ 2)
                | [ Expr₁ , ..., Exprₙ ] (finite list, ₙ ≥ 1)
                | [ Expr₁ , ... , Exprₙ ] (finite list, ₙ ≥ 1)
                | [ Expr₁ , ... , Exprₙ ] (finite list, ₙ ≥ 1)
                | [ Expr₁ , ... , Exprₙ ] (finite list, ₙ ≥ 1)
                | [ Expr₁ , ... , Exprₙ ] (finite list, ₙ ≥ 1)
                | [ Expr₁ , ... , Exprₙ ] (finite list, ₙ ≥ 1)
                | [ Expr₁ , ... , Exprₙ ] (finite list, ₙ ≥ 1)
                | [ Expr₁ , ... , Exprₙ ] (finite list, ₙ ≥ 1)
                | ( InfixExpr QOp ) (left section)
                | ( QOp₁) InfixExpr (right section)
QDataConstr { FBind₁ , ..., FBindₙ } (record construction, ₙ ≥ 0)
BasicExpr {QDataConstr} { FBind₁ , ..., FBindₙ } (record update, ₙ ≥ 1)
\[ 
Alt ::= \text{Pattern} \rightarrow \text{Expr} \mid \text{Pattern GdAlts} \\
GdAlts ::= \mid \text{InfixExpr} \rightarrow \text{Expr} \mid \text{GdAlts} \\
FBind ::= \text{QLabel} = \text{Expr} \\
Qual ::= \text{Pattern} \leftarrow \text{Expr} \mid \text{let LocalDecl} \\
Stmt ::= \text{Pattern} \leftarrow \text{Expr} \mid \text{let LocalDecl} \\
\]

\[ 
\text{Literal ::= Int} \mid \text{Float} \mid \text{Char} \mid \text{String} \\
\]

\[ 
\text{GDataConstr ::= \{} \mid ([ ] \mid (\{,\},)) \mid \text{QDataConstr} \\
Variable ::= \text{VariableID} \mid \text{( InfixOpID )} \mid \text{(variable)} \\
Function ::= \text{FunctionID} \mid \text{( InfixOpID )} \mid \text{(function)} \\
QFunction ::= \text{QFunctionID} \mid \text{( QInfixOpID )} \mid \text{(qualified function)} \\
DataConstr ::= \text{DataConstrID} \mid \text{( InfixOpID )} \mid \text{(constructor)} \\
QDataConstr ::= \text{QDataConstrID} \mid \text{( QInfixOpID )} \mid \text{(qualified constructor)} \\
Label ::= \text{LabelID} \mid \text{( InfixOpID )} \mid \text{(label)} \\
QLabel ::= \text{QLabelID} \mid \text{( QInfixOpID )} \mid \text{(qualified label)} \\
VarOp ::= \text{InfixOpID} \mid \text{'VariableID'} \mid \text{(variable operator)} \\
FunOp ::= \text{InfixOpID} \mid \text{'FunctionID'} \mid \text{(function operator)} \\
QFunOp ::= \text{QInfixOpID} \mid \text{'QFunctionID'} \mid \text{(qualified function operator)} \\
ConOp ::= \text{InfixOpID} \mid \text{'DataConstrID'} \mid \text{(constructor operator)} \\
QConOp ::= \text{QInfixOpID} \mid \text{'QDataConstrID'} \mid \text{(qualified constructor operator)} \\
LabelOp ::= \text{InfixOpID} \mid \text{'LabelID'} \mid \text{(label operator)} \\
QLabelOp ::= \text{QInfixOpID} \mid \text{'QLabelID'} \mid \text{(qualified label operator)} \\
\]

\[ 
Op ::= \text{FunOp} \mid \text{ConOp} \mid \text{LabelOp} \mid \text{(operator)} \\
QOp ::= \text{VarOp} \mid \text{QFunOp} \mid \text{QConOp} \mid \text{QLabelOp} \mid \text{(qualified operator)} \\
GConSym ::= : \mid \text{QInfixOpID} \mid \text{(general constructor symbol)} \\
\]
5 Optimization of Curry Programs

After the invocation of the Curry front end, which parses a Curry program and translates it into the intermediate FlatCurry representation, PAKCS applies a transformation to optimize Boolean equalities occurring in the Curry program. The ideas and details of this optimization are described in [9]. Therefore, we sketch only some basic ideas and options to influence this optimization.

Consider the following definition of the operation \texttt{last} to extract the last element in list:

\begin{verbatim}
last xs | xs == _++[x] = x
where x free
\end{verbatim}

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

\begin{verbatim}
last xs | xs =:= _++[x] = x
where x free
\end{verbatim}

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

To support this programming style, PAKCS has a built-in optimization phase on FlatCurry files. For this purpose, the optimizer analyzes the FlatCurry programs for occurrences of “\texttt{==}” and replaces them by “\texttt{=:=}” whenever the result \texttt{False} is not required. The usage of the optimizer can be influenced by setting the property flag \texttt{bindingoptimization} in the configuration file .pakcsrc. The following values are recognized for this flag:

\texttt{no}: Do not apply this transformation.

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

\texttt{full}: Perform a complete “required values” analysis of the program (see [9]) and use this information to optimize programs. In most cases, this does not yield better results so that the \texttt{fast} mode is sufficient.

Hence, to turn off this optimization, one can either modify the flag \texttt{bindingoptimization} in the configuration file .pakcsrc or dynamically pass this change to the invocation of PAKCS by

\begin{verbatim}
... -Dbindingoptimization=no ...
\end{verbatim}
6  cypm: The Curry Package Manager

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

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

> cypm update

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

> cypm list

<table>
<thead>
<tr>
<th>Name</th>
<th>Synopsis</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract-curry</td>
<td>Libraries to deal with AbstractCurry programs</td>
<td>2.0.0</td>
</tr>
<tr>
<td>abstract-haskell</td>
<td>Libraries to represent Haskell programs in Curry</td>
<td>2.0.0</td>
</tr>
<tr>
<td>addtypes</td>
<td>A tool to add missing type signatures in a Curry program</td>
<td>2.0.0</td>
</tr>
<tr>
<td>base</td>
<td>Base libraries for Curry systems</td>
<td>1.0.0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The command

> cypm info PACKAGE

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

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

> cypm install PACKAGE

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

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

> cypm update

... Package 'cpm-xxx' checked out ...

... INFO Installing executable 'cypm' into '/home/joe/.cpm/bin'

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

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

2http://curry-lang.org/tools/cpm


7 CurryCheck: A Tool for Testing Properties of Curry Programs

CurryCheck is a tool that supports the automation of testing Curry programs. The tests to be executed can be unit tests as well as property tests parameterized over some arguments. The tests can be part of any Curry source program and, thus, they are also useful to document the code. CurryCheck is based on EasyCheck [16]. Actually, the properties to be tested are written by combinators proposed for EasyCheck, which are actually influenced by QuickCheck [17] but extended to the demands of functional logic programming.

7.1 Installation

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

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

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

7.2 Testing Properties

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

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

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

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

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

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

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

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

---

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

---
import Test.Prop

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

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

revRevIsId xs = rev (rev xs) == xs

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

> curry-check Rev

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

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

\[
\texttt{revRevIsId} :: \texttt{[Int]} \to \texttt{Prop} \\
\texttt{revRevIsId} \ x = \texttt{rev} \ (\texttt{rev} \ x) == x
\]

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

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

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

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

30
• The property $x <^~ y$ is satisfied if $y$ evaluates to every value of $x$, i.e., the set of values of $x$ must be a subset of the set of values of $y$.

• The combinator “$<^~>$” is satisfied if the multi-set of values of both sides are equal. Hence, this operator can be used to compare the number of computed solutions of two expressions.

• The property always $x$ is satisfied if all values of $x$ are true.

• The property eventually $x$ is satisfied if some value of $x$ is true.

• The property failing $x$ is satisfied if $x$ has no value, i.e., its evaluation fails.

• The property $x \# n$ is satisfied if $x$ has $n$ different values.

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

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

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

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

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

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

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

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

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

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

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

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

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

```haskell
31
```
then we can reformulate our property as follows:

\[
\text{permCount } xs = \text{allDifferent } xs \implies \text{perm } xs \# \text{fac (length } xs)\]

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

\[
\text{sorted} :: \left[\text{Int}\right] \rightarrow \text{Bool}
\]
\[
\text{sorted } [] = \text{True}
\]
\[
\text{sorted } [.] = \text{True}
\]
\[
\text{sorted } (x:y:zs) = x\leq y \&\& \text{sorted } (y:zs)
\]

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

\[
\text{permIsEventuallySorted} :: \left[\text{Int}\right] \rightarrow \text{Prop}
\]
\[
\text{permIsEventuallySorted } xs = \text{eventually} \}$ \text{sorted } (\text{perm } xs)
\]

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

\[
\text{psort} :: \left[\text{Int}\right] \rightarrow \left[\text{Int}\right]
\]
\[
\text{psort } xs \mid \text{sorted } ys = ys
\]
\[
\text{where } ys = \text{perm } xs
\]

Again, we can write some properties:

\[
\text{psortIsAlwaysSorted } xs = \text{always} \}$ \text{sorted } (\text{psort } xs)
\]
\[
\text{psortKeepsLength } xs = \text{length } (\text{psort } xs) \rightsquigarrow \text{length } xs
\]

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

\[
\text{qsort} :: \left[\text{Int}\right] \rightarrow \left[\text{Int}\right]
\]
\[
\text{qsort } [] = []
\]
\[
\text{qsort } (x:l) = \text{qsort } (\text{filter } (\leq x) l) \cdot x : \text{qsort } (\text{filter } (>x) l)
\]

The following property specifies the correctness of quicksort:

\[
\text{qsortIsSorting } xs = \text{qsort } xs \rightsquigarrow \text{psort } xs
\]

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

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

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

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

- a ‘returns‘ x is satisfied if the I/O action a returns the value x.
a 'sameReturns' b is satisfied if the I/O actions a and b return identical values.

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

7.3 Generating Test Data

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

\[
\text{neg_or } b_1 \ b_2 = \neg (b_1 \lor b_2) \iff \neg b_1 \land \neg b_2
\]

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

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

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

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

For a given type, CurryCheck automatically enumerates all values of this type (except for function types). In KiCS2, this is done by exploiting the functional logic features of Curry, i.e., by simply collecting all values of a free variable. For instance, the library `Test.EasyCheck` defines an operation `valuesOf :: a -> [a]` which computes the list of all values of the given argument according to a fixed strategy (in the current implementation: randomized level diagonalization [16]). For instance, we can get 20 values for a list of integers by

```haskell
Test.EasyCheck> take 20 (valuesOf (_::[Int]))
[[],[-1],[-3],[0],[1],[-1,0],[-2],[0,0],[3],[-1,1],[-3,0],[0,1],[2],
[-1,-1],[-5],[0,-1],[5],[-1,2],[-9],[0,2]]
```
Since the features of PAKCS for search space exploration are more limited, PAKCS uses in CurryCheck explicit generators for search tree structures which are defined in the module `SearchTreeGenerators`. For instance, the operations

\[
\begin{align*}
genInt &: \text{SearchTree } \text{Int} \\
genList &: \text{SearchTree } \text{a} \rightarrow \text{SearchTree } [\text{a}]
\end{align*}
\]

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

\[
\text{valuesOfSearchTree} &: \text{SearchTree } \text{a} \rightarrow [\text{a}]
\]

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

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

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

\[
\begin{align*}
data \text{ Nat} &= \text{Z} \mid S \text{ Nat} \\
\text{add} &: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{add } \text{Z} \ n &= n \\
\text{add } (S \ m) \ n &= S(\text{add } m \ n) \\
\text{addIsCommutative } x \ y &= \text{add } x \ y =-\text{add } y \ x \\
\text{addIsAssocitative } x \ y \ z &= \text{add } (\text{add } x \ y) \ z =-\text{add } x \ (\text{add } y \ z)
\end{align*}
\]

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

\[
\begin{align*}
data \text{ Tree } \text{a} &= \text{Leaf } \text{a} \mid \text{Node } [\text{Tree } \text{a}] \\
\text{leaves } (\text{Leaf } x) &= [x] \\
\text{leaves } (\text{Node } ts) &= \text{concatMap leaves ts} \\
\text{mirror } (\text{Leaf } x) &= \text{Leaf } x \\
\text{mirror } (\text{Node } ts) &= \text{Node } (\text{reverse } (\text{map mirror } ts))
\end{align*}
\]

Then we can state and check two properties on mirroring:

\[
\begin{align*}
\text{doubleMirror } t &= \text{mirror } (\text{mirror } t) =-\text{t} \\
\text{leavesOfMirrorAreReversed } t &= \text{leaves } t =-\text{reverse } (\text{leaves } (\text{mirror } t))
\end{align*}
\]

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

\[
\text{sumUp } n = \text{if } n==0 \text{ then 0 else n + sumUp } (n-1)
\]

Since there is also a simple formula to compute this sum, we can check it:
sumUpIsCorrect \( n \) = \( n \geq 0 \Rightarrow \text{sumUp} n \equiv n \cdot (n+1) \div 2 \)

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

\[
\text{genInt} = \text{genCons0} 0 ||| \text{genCons1} (+1) \text{genInt}
\]

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

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

\[
\text{genInt} = \text{genCons0} 0 ||| \text{genCons1} (\lambda n \rightarrow 2(n+1)) \text{genInt} \\
||| \text{genCons1} (\lambda n \rightarrow 2n+1) \text{genInt}
\]

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

\[
\ldots \text{SearchTree} > \text{putStr} \ (\text{showSearchTree} \ (\text{limitSearchTree} 6 \ \text{genInt}))
\]

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

\[
\text{data NonNeg} = \text{NonNeg} \{ \text{nonNeg} :: \text{Int} \}
\]

\[
\text{genNonNeg} = \text{genCons1} \ \text{NonNeg} \ \text{genNN} \\
\text{where}
\begin{align*}
\text{genNN} &= \text{genCons0} 0 ||| \text{genCons1} (\lambda n \rightarrow 2(n+1)) \ \text{genNN} \\
& \quad ||| \text{genCons1} (\lambda n \rightarrow 2n+1) \ \text{genNN}
\end{align*}
\]

Now we can either redefine `sumUpIsCorrect` on this type

\[
\text{sumUpIsCorrectOnNonNeg} \ (\text{NonNeg} \ n) = \text{sumUp} n \equiv n \cdot (n+1) \div 2
\]

or we simply reuse the old definition by

\[
\text{sumUpIsCorrectOnNonNeg} = \text{sumUpIsCorrect} \ . \ \text{nonNeg}
\]
7.4 Checking Equivalence of Operations

CurryCheck supports also equivalence tests for operations. Two operations are considered as equivalent if they can be replaced by each other in any possible context without changing the computed values (this is also called contextual equivalence and precisely defined in [8] for functional logic programs). For instance, the Boolean operations

\[
\begin{align*}
  f_1 &: \text{Bool} \to \text{Bool} \\
  f_1 x &= \text{not} \ (\text{not} \ x) \\
  f_2 &: \text{Bool} \to \text{Bool} \\
  f_2 x &= x
\end{align*}
\]

are equivalent, whereas

\[
\begin{align*}
  g_1 &: \text{Bool} \to \text{Bool} \\
  g_1 \text{False} &= \text{True} \\
  g_1 \text{True} &= \text{True} \\
  g_2 &: \text{Bool} \to \text{Bool} \\
  g_2 x &= \text{True}
\end{align*}
\]

are not equivalent: \(g_1\) failed has no value but \(g_2\) failed evaluates to True.

To check the equivalence of operations, one can use the property combinator \(\leftrightarrow\):

\[
\begin{align*}
  f_1 \text{equiv} \ f_2 &= f_1 \leftrightarrow f_2 \\
  g_1 \text{equiv} \ g_2 &= g_1 \leftrightarrow g_2
\end{align*}
\]

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

CurryCheck transforms such properties into properties where both operations are compared w.r.t. all partial values and partial results. The details are described in [11].

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

\[
\begin{align*}
  \text{ints1} \ n &= n : \text{ints1} \ (n+1) \\
  \text{ints2} \ n &= n : \text{ints2} \ (n+2)
\end{align*}
\]

-- This property will be falsified by CurryCheck:

\[
\text{ints1 equiv ints2} = \text{ints1} \leftrightarrow \text{ints2}
\]

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

\[
\begin{align*}
  \text{psort} &: \text{Ord} \ a \to [a] \to [a] \\
  \text{psort} \ xs \mid \text{sorted} \ ys = ys \\
  \text{where} \ ys &= \text{perm} \ xs
\end{align*}
\]

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

\[
\begin{align*}
  \text{isort} &: \text{Ord} \ a \to [a] \to [a]
\end{align*}
\]
isort xs = idSorted (perm xs)
where idSorted [] = []
idSorted [x] = [x]
idSorted (x:y:ys) | x<=y = x : idSorted (y:ys)

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

\texttt{psort\_equiv\_isort = psort <\sim> (isort :: \texttt{[Int]} \rightarrow \texttt{[Int]})}

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

\texttt{psort\_equiv\_isort\_TERMINATE = psort <\sim> (isort :: \texttt{[Int]} \rightarrow \texttt{[Int]})}

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

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

\textbf{manual:} This is the default mode. In this mode, all equivalence tests are executed with first technique described above, unless the name of the test has the suffix \texttt{\_TERMINATE}.

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

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

\textbf{ground:} In this mode, only ground equivalence is tested, i.e., each equivalence property

\texttt{g1\_equiv\_g2 = g1 <\sim> g2}

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

\texttt{g1\_equiv\_g2 x1 ... xn = g1 x1 ... xn <~> g2 x1 ... xn}

Note this property is more restrictive than contextual equivalence. For instance, the non-equivalence of \texttt{g1} and \texttt{g2} as shown above cannot be detected by testing ground equivalence only.
7.5 Checking Contracts and Specifications

The expressive power of Curry supports writing high-level specifications as well as efficient implementations for a given problem in the same programming language, as discussed in [8]. If a specification or contract is provided for some function, then CurryCheck automatically generates properties to test this specification or contract.

Following the notation proposed in [8], a specification for an operation \( f \) is an operation \( f'\text{spec} \) of the same type as \( f \). A contract consists of a pre- and a postcondition, where the precondition could be omitted. A precondition for an operation \( f \) of type \( \tau \rightarrow \tau' \) is an operation

\[
f'\text{pre} :: \tau \rightarrow \text{Bool}
\]

whereas a postcondition for \( f \) is an operation

\[
f'\text{post} :: \tau \rightarrow \tau' \rightarrow \text{Bool}
\]

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

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

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

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

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

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

```haskell
sortSatisfiesPostCondition :: [Int] \rightarrow \text{Prop}
sortSatisfiesPostCondition x = always (sort'post x (sort x))

sortSatisfiesSpecification :: \text{Prop}
sortSatisfiesSpecification = sort \Rightarrow sort'spec
```

38
7.6 Combining Testing and Verification

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

\[
\text{sortlength } \text{xs} = \text{length (sort xs)} ~\wedge~ \text{length xs}
\]

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

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

- If there is a proof file \( \text{proof-}M\text{-f-IsDeterministic.*} \), a determinism annotation for operation \( M.f \) is not tested.
- If there is a proof file \( \text{proof-}M\text{-f-SatisfiesPostCondition.*} \), a postcondition for operation \( M.f \) is not tested.
- If there is a proof file \( \text{proof-}M\text{-f-SatisfiesSpecification.*} \), a specification for operation \( M.f \) is not tested.

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

- \( \text{proof-Sort-sortlength.tex} \)
- \( \text{PROOF_Sort_sortlength.agda} \)
- \( \text{Proof-Sort_sortlength.smt} \)
- \( \text{ProofSortSortlength.smt} \)

7.7 Checking Usage of Specific Operations

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

- The prelude operation “=:<=” is used to implement functional patterns [6]. It should not be
used in source programs to avoid unintended uses. Hence, CurryCheck reports such unintended uses.

- Set functions [7] are used to encapsulate all non-deterministic results of some function in a set structure. Hence, for each top-level function $f$ of arity $n$, the corresponding set function can be expressed in Curry (via operations defined in the library SetFunctions) by the application “set $n$ f” (this application is used in order to extend the syntax of Curry with a specific notation for set functions). However, it is not intended to apply the operator “set $n$” to lambda abstractions, locally defined operations or operations with an arity different from $n$. Hence, CurryCheck reports such unintended uses of set functions.
8 CurryBrowser: A Tool for Analyzing and Browsing Curry Programs

CurryBrowser is a tool to browse through the modules and operations of a Curry application, show them in various formats, and analyze their properties. Moreover, it is constructed in a way so that new analyzers can easily be connected to CurryBrowser. A detailed description of the ideas behind this tool can be found in [22, 23].

8.1 Installation

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

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

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

8.2 Basic Usage

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

- In the PAKCS environment after loading the module mod and typing the command “:browse”.

- As a shell command (provided that $HOME/.cpm/bin is in your path): curry-browse mod

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

To get an impression of the use of CurryBrowser, Figure 1 shows a snapshot of its use on a particular application (here: the implementation of CurryBrowser). The upper list box in the left column shows the modules and their imports in order to browse through the modules of an application. Similarly to directory browsers, the list of imported modules of a module can be opened or closed by clicking. After selecting a module in the list of modules, its source code, interface, or various other formats of the module can be shown in the main (right) text area. For instance, one can show pretty-printed versions of the intermediate flat programs (see below) in order to see how local function definitions are translated by lambda lifting [29] or pattern matching is translated into case expressions [18, 35]. 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.

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

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

More details about the use of CurryBrowser and all built-in analyses are available through the “Help” menu of CurryBrowser.
9 curry-doc: A Documentation Generator for Curry Programs

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

9.1 Installation

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

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

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

9.2 Documentation Comments

A documentation comment starts at the beginning of a line with “---” (also in literate programs!). All documentation comments immediately before a definition of a datatype or (top-level) function are kept together. 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).

---

5The documentation tool recognizes this association from the first identifier in a program line. If one wants to add a documentation comment to the definition of a function which is an infix operator, the first line of the operator definition should be a type definition, otherwise the documentation comment is not recognized.
@return comment

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

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

The comments can also contain markups in HTML format so that special characters like “<” must be quoted (e.g., “&lt;”). However, header tags like <h1> should not be used since the structuring is generated by CurryDoc. In addition to Markdown or HTML markups, one can also mark references to names of operations or data types in Curry programs which are translated into links 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:

```curry
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.
```

45
--- @cons Leaf - a leaf of the tree
--- @cons Node - an inner node of the tree

\[
data \text{Tree } a = \text{Leaf } a \mid \text{Node } [\text{Tree } a]
\]

### 9.3 Generating Documentation

To generate the documentation, execute the command

\[
curry\text{-doc Example}
\]

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

\[
curry\text{-doc docdir Example}
\]

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

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

\[
curry\text{-doc --noindexhtml docdir Mod}:
\]

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

\[
curry\text{-doc --onlyindexhtml docdir Mod1 Mod2 ...Modn}:
\]

This command generates only the index pages (i.e., a main index page and index pages for all functions and constructors defined in the modules `Mod1, Mod2,...,Modn` and their imported modules) in the directory `docdir`.  

46
10 curry-style: A Style Checker for Curry Programs

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

10.1 Installation

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

```bash
> cypm update
> cypm install casc
```

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

10.2 Basic Usage

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

```
curry-style prog
```

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

10.3 Configuration

CASC can be configured so that not all stylistic rules are checked. For this purpose, one should copy the global configuration file `cascrc` of CASC, which is stored in the main directory of the package, into the home directory under the name ".cascrc". Then one can configure this file according to your own preferences, which are described in this file.

---

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

7 If you installed CASC as described above, the downloaded package is located in the directory `$HOME/.cpm/bin_packages/casc`. 
11 CurryVerify: A Tool to Support the Verification of Curry Programs

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

Currently, only Agda [32] is supported as a target language for verification (but more target languages may be supported in future releases). The basic schemes to translate Curry programs into Agda programs are presented in [12]. That paper also describes the limitations of this approach. Since Curry is a quite rich programming language, not all constructs of Curry are currently supported in the translation process (e.g., no case expressions, local definitions, list comprehensions, do notations, etc). Only a kernel language, where the involved rules correspond to a term rewriting system, are translated into Agda. However, these limitations might be relaxed in future releases. Hence, the current tool should be considered as a first prototypical approach to support the verification of Curry programs.

11.1 Installation

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

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

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

11.2 Basic Usage

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

```
curry-verify prog
```

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

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

import Nat
import Test.Prop

double x = add x x

coin x = x ? S x

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

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

Figure 2: Curry program Double.curry

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

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

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

The Curry program is translated with the default scheme (see further options below) based on the
"planned choice" scheme, described in [12]. The result of this translation is shown in Figure 3.

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

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

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

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

evendoublecoin : (c1 : Choice) → (x : N) → (even (double (coin c1 x))) ≡ tt
evendoublecoin c1 x rewrite even-add-x-x (coin c1 x) = refl
As the proof is complete, we rename this Agda program into PROOF-evendoublecoin.agda so that the proof can be used by further invocations of CurryCheck.

### 11.3 Options

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

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

The options are briefly described in the following.

- `-h`, `-?`, `--help` These options trigger the output of usage information.
- `-q`, `--quiet` Run quietly and produce no informative output. However, the exit code will be non-zero if some translation error occurs.
- `-v [n]`, `--verbosity=[n]` Set the verbosity level to an optional value. The verbosity level 0 is the same as option `-q`. The default verbosity level 1 shows the translation progress. The verbosity level 2 (which is the same as omitting the level) shows also the generated (Agda) program. The verbosity level 3 shows also more details about the translation process.
- `-n`, `--nostore` Do not store the translated program in a file but show it only.
- `-p p`, `--property=p` As a default, all properties occurring in the source program are translated. If this option is provided, only property `p` is translated.
- `-t t`, `--target=t` Define the target language of the translation. Currently, only `t = Agda` is supported, which is also the default.
- `-s s`, `--scheme=s` Define the translation scheme used to represent Curry programs in the target language.

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

- `choice` Use the “planned choice” scheme, see [12] (this is the default). In this scheme, the choices made in a non-deterministic computation are abstracted by passing a parameter for these choices.
- `nondet` Use the “set of values” scheme, see [12], where non-deterministic values are represented in a tree structure.
open import bool

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

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

-- Translated Curry operations:

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

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

double : ℕ → ℕ
double x = add x x

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

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

Figure 3: Agda program TO-PROVE-evendoublecoin.agda
12 CurryPP: A Preprocessor for Curry Programs

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

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

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

Default rules: If this feature is used, one can add a default rule to operations defined in a Curry module. This provides a similar power than sequential rules but with a better operational behavior. The idea of default rules is described in [10].

Contracts: If this feature is used, the Curry preprocessor looks for contracts (i.e., specification, pre- and postconditions) occurring in a Curry module and adds them as assertions that are checked during the execution of the program. Currently, only strict assertion checking is supported which might change the operational behavior of the program. The idea and usage of contracts is described in [8].

12.1 Installation

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

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

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

12.2 Basic Usage

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

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

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

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

52
For instance, in order to use integrated code and default rules in a module and store the transformed program, one has to put the line

\{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode --optF=defaultrules --optF=-o #-\}

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

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

is equivalent to

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

### 12.3 Integrated Code

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

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

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

\{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-\}

The next sections describe the currently supported foreign languages.

#### 12.3.1 Regular Expressions

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

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

\{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-\}

```haskell
import RegExp

isID :: String → Bool
isID = match `regex [a-zA-Z][a-zA-Z0-9_]`
```
12.3.2 Format Specifications

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

{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import Data.Format

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

main = putStrLn $ showSIF "Curry" 42 3.14159

Thus, the execution of main will print the line

Name: Curry | +00042 | +3.142

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

{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import Data.Format

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

main = putStrLn $ showSIF "Curry" 42 3.14159

12.3.3 HTML Code

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

{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

htmlPage :: String → [HtmlExp]
htmlPage name = "html
  <head>
    <title>Simple Test
  </body>
  <h1>Hello {name}!</h1>
  <p>
If a Curry expression computes an HTML expression, i.e., it is of type HtmlExp instead of String, it can be integrated into the HTML syntax by double curly brackets. The following simple example, taken from [21], shows the use of this feature:

```haskell
{-# OPTIONS_CYMACE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

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

where

  tref free

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

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

12.3.4 XML Expressions

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

```haskell
{-# OPTIONS_CYMACE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

import XML

main :: IO ()
main = putStrLn $ showXmlDoc $ head "xml
12.4 SQL Statements

The Curry preprocessor also supports SQL statements in their standard syntax as integrated code. In order to ensure a type-safe integration of SQL statements in Curry programs, SQL queries are type-checked in order to determine their result type and ensure that the entities used in the queries are type correct with the underlying relational database. For this purpose, SQL statements are integrated code require a specification of the database model in form of entity-relationship (ER) model. From this description, a set of Curry data types are generated which are used to represent entities in the Curry program (see Section 12.4.1). The Curry preprocessor uses this information to type check the SQL statements and replace them by type-safe access methods to the database. In the following, we sketch the use of SQL statements as integrated code. A detailed description of the ideas behind this technique can be found in [25]. Currently, only SQLite databases are supported.

12.4.1 ER Specifications

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

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

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

Instead of requiring the use of some graphical ER modeling tool, ERDs must be specified in textual form as a Curry data term, see also [15]. In this representation, an ERD has a name, which is also used as the module name of the generated Curry code, lists of entities and relationships:

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

Each entity consists of a name and a list of attributes, where each attribute has a name, a domain, and specifications about its key and null value property:
data Entity = Entity String [Attribute]

data Attribute = Attribute String Domain Key Null

data Key = NoKey | PKey | Unique

type Null = Bool

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

Figure 4: A simple entity-relationship diagram for university lectures
Thus, each attribute is part of a primary key (PKey), unique (Unique), or not a key (NoKey). Furthermore, it is allowed that specific attributes can have null values, i.e., can be undefined. The domain of each attribute is one of the standard domains or some user-defined type. In the latter case, the first argument of the constructor UserDefined is the qualified type name used in the Curry application program. For each kind of domain, one can also have a default value (modeled by the Maybe type). The constructor KeyDom is not necessary to represent ERDs but it is internally used to transform complex ERDs into relational database schemas.

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

```haskell
data Relationship = Relationship String [REnd]
data REnd = REnd String String Cardinality
data Cardinality = Exactly Int | Between Int MaxValue
data MaxValue = Max Int | Infinite
```

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

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

The PAKCS library Database.ERD contains the ER datatypes described above. Thus, the specification of the conceptual database model must be a data term of type Database.ERD.ERD. Figure 5 on (page 64) shows the complete ER data term specification corresponding to the ERD of Fig. 4.

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

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

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

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

2. A new Curry module Uni.CDBI is generated. It contains the definitions of entities and relationships as Curry data types. Since entities are uniquely identified via a database key, each
entity definition has, in addition to its attributes, this key as the first argument. For instance, the following definitions are generated for our university ERD (among many others):

```haskell
data StudentID = StudentID Int
data Student = Student StudentID String String Int String Int

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

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

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

### 12.4.2 SQL Statements as Integrated Code

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

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

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

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

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

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

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

```haskell
type SQLResult a = Either DBError a
```
This library defines also an operation

\[
\text{fromSQLResult} :: \text{SQLResult} \ a \to \ a
\]

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

\[
\text{allStudents} :: \text{IO} \ \text{[Student]}
\]

\[
\text{allStudents} = \text{liftM} \ \text{fromSQLResult} \ \\
\quad \quad \text{``sql Select * From Student;''}
\]

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

\[
\text{allStudentsAction} :: \text{DBAction} \ \text{[Student]}
\]

\[
\text{allStudentsAction} = \text{``sql* Select * From Student;''}
\]

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

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

\[
\text{youngStudents} :: \text{IO} \ (\text{SQLResult} \ \text{[Student]})
\]

\[
\text{youngStudents} = \text{``sql Select * From Student}
\quad \quad \text{Where Age between 18 and 21;''}
\]

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

\[
\text{studAgeBetween} :: \text{Int} \to \text{Int} \to \text{IO} \ (\text{SQLResult} \ \text{[Student]})
\]

\[
\text{studAgeBetween} \ \text{min} \ \text{max} =
\quad \quad \text{``sql Select * From Student}
\quad \quad \quad \text{Where Age between \{min\} and \{max\};''}
\]

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

\[
\text{studAgeBetween} :: \text{Int} \to \text{Int} \to \text{IO} \ (\text{SQLResult} \ \text{[(String,Int)]})
\]

\[
\text{studAgeBetween} \ \text{min} \ \text{max} =
\quad \quad \text{``sql Select Name, Age}
\quad \quad \quad \text{From Student Where Age between \{min\} and \{max\}}
\quad \quad \quad \quad \text{Order By Name Desc;''}
\]

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

\[
\text{studGoodGrades} :: \text{IO} \ (\text{SQLResult} \ \text{[(String, Float)]})
\]

\[
\text{studGoodGrades} = \text{``sql Select Distinct s.Name, r.Grade}
\quad \quad \text{From Student as s, Result as r}
\quad \quad \quad \text{Where Satisfies s has_a r And r.Grade < 2.0;''}
\]

This query retrieves a list of pairs containing the names and grades of students having a grade better than 2.0. This query is beyond pure SQL since it also includes a condition on the relation \(\text{has_a}\) specified in the ER model (“Satisfies s has_a r”).
The complete SQL syntax supported by the Curry preprocessor is shown in Appendix C. More details about the implementation of this SQL translator can be found in [25, 30].

12.5 Default Rules

An alternative to sequential rules are default rules, i.e., these two options cannot be simultaneously used. Default rules are activated by the preprocessor option “defaultrules”. In this case, one can add to each operation a default rule. A default rule for a function \( f \) is defined as a rule defining the operation “\( f \)'default” (this mechanism avoids any language extension for default rules). A default rule is applied only if no “standard” rule is applicable, either because the left-hand sides’ pattern do not match or the conditions are not satisfiable. The idea and detailed semantics of default rules are described in [10].

Default rules are preferable over the sequential rule selection strategy since they have a better operational behavior. This is due to the fact that the test for the application of default rules is done with the same (sometimes optimal) strategy than the selection of standard rules. Moreover, default rules provide a similar power than sequential rules, i.e., they can be applied if the standard rules have complex (functional) patterns or complex conditions.

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

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

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

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

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

import Combinatorial(permute)
import Integer(abs)

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

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

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

```
> cypm add setfunctions
```

61
before compiling a program containing default rules with the Curry preprocessor.

12.6 Contracts

Contracts are annotations in Curry program to specify the intended meaning and use of operations by other operations or predicates expressed in Curry. The idea of using contracts for the development of reliable software is discussed in [8]. The Curry preprocessor supports dynamic contract checking by transforming contracts, i.e., specifications and pre-/postconditions, into assertions that are checked during the execution of a program. If some contract is violated, the program terminates with an error.

The transformation of contracts into assertions is described in [8]. Note that only strict assertion checking is supported at the moment. Strict assertion checking might change the operational behavior of the program. The notation of contracts is defined in [8]. To transform such contracts into assertions, one has to use the option “contracts” for the preprocessor.

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

```curry
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=contracts #-} ...

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

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

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

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

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

```
Quicksort> sort [3,1,4,2,1]
Postcondition of 'sort' (module Quicksort, line 27) violated for: [1,2,1] → [1,2]
ERROR: Execution aborted due to contract violation!
```

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

62
easily done by executing

> cypm add setfunctions
> cypm add contracts

before compiling a program containing contracts with the Curry preprocessor.
ERD "Uni"

[Entity "Student"
  [Attribute "Name" (StringDom Nothing) NoKey False,
  Attribute "Firstname" (StringDom Nothing) NoKey False,
  Attribute "MatNum" (IntDom Nothing) Unique False,
  Attribute "Email" (StringDom Nothing) Unique False,
  Attribute "Age" (IntDom Nothing) NoKey True],

Entity "Lecture"
  [Attribute "Title" (StringDom Nothing) NoKey False,
  Attribute "Topic" (StringDom Nothing) NoKey True],

Entity "Lecturer"
  [Attribute "Name" (StringDom Nothing) NoKey False,
  Attribute "Firstname" (StringDom Nothing) NoKey False],

Entity "Place"
  [Attribute "Street" (StringDom Nothing) NoKey False,
  Attribute "StrNr" (IntDom Nothing) NoKey False,
  Attribute "RoomNr" (IntDom Nothing) NoKey False],

Entity "Time"
  [Attribute "Time" (DateDom Nothing) Unique False],

Entity "Exam"
  [Attribute "GradeAverage" (FloatDom Nothing) NoKey True],

Entity "Result"
  [Attribute "Attempt" (IntDom Nothing) NoKey False,
  Attribute "Grade" (FloatDom Nothing) NoKey True,
  Attribute "Points" (IntDom Nothing) NoKey True]]

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

Relationship "Participation"
  [REnd "Student" "participated_by" (Between 0 Infinite),
  REnd "Lecture" "participates" (Between 0 Infinite)],

Relationship "Taking"
  [REnd "Result" "has_a" (Between 0 Infinite),
  REnd "Student" "belongs_to" (Exactly 1)],

Relationship "Resulting"
  [REnd "Exam" "result_of" (Exactly 1),
  REnd "Result" "results_in" (Between 0 Infinite)],

Relationship "Belonging"
  [REnd "Exam" "has_a" (Between 0 Infinite),
  REnd "Lecture" "belongs_to" (Exactly 1)],

Relationship "ExamDate"
  [REnd "Exam" "taking_place" (Between 0 Infinite),
  REnd "Time" "at" (Exactly 1)],

Relationship "ExamPlace"
  [REnd "Exam" "taking_place" (Between 0 Infinite),
  REnd "Place" "in" (Exactly 1)]

Figure 5: The ER data term specification of Fig. 4
13 \textbf{runcurry: Running Curry Programs}

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

13.1 \textbf{Installation}

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

\begin{verbatim}
> cypm update
> cypm install runcurry
\end{verbatim}

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

13.2 \textbf{Using runcurry}

Basically, the command \texttt{runcurry} supports three modes of operation:

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

\begin{verbatim}
import System(getArgs)

main = getArgs >>= print
\end{verbatim}

This program can be executed by the shell command

\begin{verbatim}
> runcurry ShowArgs.curry Hello World!
\end{verbatim}

which produces the output

\begin{verbatim}
["Hello","World!"]
\end{verbatim}

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

\begin{verbatim}
> runcurry
main = putStr . unlines . map show . take 8 $ [1..]
\end{verbatim}

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

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

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

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

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

```
#!/usr/bin/env runcurry
import Char(isSpace)
import System(getArgs)

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

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

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

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

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

66
When this command is executed, the command *runcurry* compiles the program and evaluates the expression *main*. Since the compilation might take some time in more complex scripts, one can also save the result of the compilation in a binary file. To obtain this behavior, one has to insert the line

```plaintext
#jit
```

in the script file, e.g., in the second line. With this option, a binary of the compiled program is saved (in the same directory as the script). Now, when the same script is executed the next time, the stored binary file is executed (provided that it is still newer than the script file itself, otherwise it will be recompiled). This feature combines easy scripting with Curry together with fast execution.
CASS (Curry Analysis Server System) is a tool for the analysis of Curry programs. CASS is generic so that various kinds of analyses (e.g., groundness, non-determinism, demanded arguments) can be easily integrated into CASS. In order to analyze larger applications consisting of dozens or hundreds of modules, CASS supports a modular and incremental analysis of programs. Moreover, it can be used by different programming tools, like documentation generators, analysis environments, program optimizers, as well as Eclipse-based development environments. For this purpose, CASS can also be invoked as a server system to get a language-independent access to its functionality. CASS is completely implemented Curry as a master/worker architecture to exploit parallel or distributed execution environments. The general design and architecture of CASS is described in [26]. In the following, CASS is presented from a perspective of a programmer who is interested to analyze Curry programs.

14.1 Installation

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

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

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

14.2 Using CASS to Analyze Programs

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

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

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

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

For instance, consider the following Curry module `Rev.curry`:
append :: [a] → [a] → [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys

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

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

CASS supports three different usage modes to analyze this program.

14.2.1 Batch Mode

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

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

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

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

14.2.2 API Mode

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

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

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

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

---

8 More output is generated when the parameter `debugLevel` is changed in the configuration file `.curryanalysisrc` which is installed in the user’s home directory when CASS is started for the first time.
The CASS module `GenericProgInfo` contains operations to access the analysis information computed by CASS. For instance, the operation

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

```
demandAnalysis :: Analysis DemendedArgs
```

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

```haskell
import AnalysisServer (analyzeGeneric)
import GenericProgInfo (lookupProgInfo)
import Demandedness (demandAnalysis)

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

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

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

### 14.2.3 Server Mode

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

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

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

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

```
GetAnalysis
SetCurryPath <dir1>:<dir2>:...
AnalyzeModule <analysis name> <output type> <module name>
AnalyzeInterface <analysis name> <output type> <module name>
AnalyzeFunction <analysis name> <output type> <module name> <function name>
AnalyzeDataConstructor <analysis name> <output type> <module name> <constructor name>
AnalyzeTypeConstructor <analysis name> <output type> <module name> <type name>
StopServer
```
The output type can be Text, CurryTerm, or XML. The answer to each request can have two formats:

error <error message>

if an execution error occurred, or

ok <n>
<result text>

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

<analysis name> <output type>

For instance, a communication could be:

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

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

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

For instance, if we start CASS by

> cass --server -p 12345

we can communicate with CASS as follows (user inputs are prefixed by “>”):

> telnet localhost 12345
Connected to localhost.
> GetAnalysis
ok 57
Overlapping XML
Overlapping CurryTerm
Overlapping Text
Deterministic XML
...
> AnalyzeModule Demand Text Rev
ok 3
append : demanded arguments: 1
main : demanded arguments: 1,2
rev : demanded arguments: 1
> AnalyzeModule Demand CurryTerm Rev
ok 1
14.3 Implementing Program Analyses

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

\[
\text{Analysis } a
\]

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

\[
\text{overlapAnalysis :: Analysis Bool}
\]

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

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

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

```haskell
import FlatCurry.Types
...

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

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

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

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

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

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

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

    dependencyFuncAnalysis :: String → a → (FuncDecl → [(QName,a)] → a) → Analysis a

Here, the second argument specifies the start value of the fixpoint analysis, i.e., the bottom element of the abstract domain.

For instance, a determinism analysis could be based on an abstract domain described by the data type

    data Deterministic = NDet | Det

Here, Det is interpreted as “the operation always evaluates in a deterministic manner on ground constructor terms.” However, NDet is interpreted as “the operation might evaluate in different ways for given ground constructor terms.” The apparent imprecision is due to the approximation of the analysis. For instance, if the function \( f \) is defined by overlapping rules and the function \( g \) might call \( f \), then \( g \) is judged as non-deterministic (since it is generally undecidable whether \( f \) is actually called by \( g \) in some run of the program).
The determinism analysis requires to examine the current function as well as all directly or indirectly called functions for overlapping rules. Due to recursive function definitions, this analysis cannot be done in one shot—it requires a fixpoint computation. CASS provides such fixpoint computations and requires only the implementation of an operation of type

\[
\text{FuncDecl} \rightarrow [(\text{QName}, a)] \rightarrow a
\]

where “\(a\)” denotes the type of abstract values. The second argument of type \([(\text{QName}, a)]\) represents the currently known analysis values for the functions directly used in this function declaration.

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

\[
\text{detFunc} :: \text{FuncDecl} \rightarrow [(\text{QName}, \text{Deterministic})] \rightarrow \text{Deterministic}
\]

\[
\text{detFunc} \ (\text{Func} \ f \ _ \ _ \_ \ (\text{Rule} \ _ \ e)) \ \text{calledFuncs} =
\begin{align*}
\text{if} & \text{ orInExpr} \ e \ || \ \text{freeVarInExpr} \ e \ || \ \text{any} \ (==\text{NDet}) \ (\text{map} \ \text{snd} \ \text{calledFuncs}) \\
& \text{then} \ \text{NDet} \\
& \text{else} \ \text{Det}
\end{align*}
\]

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

The complete determinism analysis can be specified as

\[
\text{detAnalysis} :: \text{Analysis} \ \text{Deterministic}
\]

\[
\text{detAnalysis} = \text{dependencyFuncAnalysis} \ "\text{Deterministic}" \ \text{Det} \ \text{detFunc}
\]

This definition is sufficient to execute the analysis with CASS since the analysis system takes care of computing fixpoints, calling the analysis functions with appropriate values, analyzing imported modules, etc. Nevertheless, the analysis must be defined so that the fixpoint computation always terminates. This can be achieved by using an abstract domain with finitely many values and ensuring that the analysis function is monotone w.r.t. some ordering on the values.
15 ERD2Curry: A Tool to Generate Programs from ER Specifications

ERD2Curry is a tool to generate Curry code to access and manipulate data persistently stored in relational databases. The Curry code is generated from a description of the logical model of the database in form of an entity relationship diagram. The idea of this tool is described in detail in [15]. Thus, we describe only the basic steps to use this tool.

15.1 Installation

The current implementation of ERD2Curry is a package managed by the Curry Package Manager CPM (see also Section 6). Thus, to install the newest version of ERD2Curry, use the following commands:

```
> cypm update
> cypm install ertools
```

This downloads the newest package, compiles it, and places the executable erd2curry into the directory $HOME/.cpm/bin. Hence it is recommended to add this directory to your path in order to execute ERD2Curry as described below.

15.2 Basic Usage

If one creates an entity relationship diagram (ERD) with the Umbrello UML Modeller, one has to store its XML description in XMI format (as offered by Umbrello) in a file, e.g., “myerd.xmi”. This description can be compiled into a Curry program by the command

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

If MyData is the name of the ERD, the Curry program file “MyData.curry” is generated containing all the necessary database access code as described in [15]. In addition to the generated Curry program file, two auxiliary program files ERDGeneric.curry and KeyDatabase.curry are created in the same directory.

If one does not want to use the Umbrello UML Modeller, which might be the preferred method since the interface to the Umbrello UML Modeller is no longer actively supported, one can also define an ERD in a Curry program as a (exported!) top-level operation of type ERD (w.r.t. the type definition given in the library pakcs/home/lib/Database/ERD.curry). The directory examples in the package ertools contains two examples for such ERD program files:

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

UniERD.curry: This is an ERD model for university lectures as presented in the paper [15].

Figure 6 shows the ER specification stored in the Curry program file “BlogERD.curry”. This ER specification can be compiled into a Curry program by the command

```
erd2curry BlogERD.curry
```

---

9If you installed ERD2Curry as described above, the downloaded ertools package is located in the directory $HOME/.cpm/bin_packages/ertools.
import Database.ERD

blogERD :: ERD
blogERD =
  ERD "Blog"
    [Entity "Entry"
      [Attribute "Title" (StringDom Nothing) Unique False,
        Attribute "Text" (StringDom Nothing) NoKey False,
        Attribute "Author" (StringDom Nothing) NoKey False,
        Attribute "Date" (DateDom Nothing) NoKey False],
    Entity "Comment"
      [Attribute "Text" (StringDom Nothing) NoKey False,
        Attribute "Author" (StringDom Nothing) NoKey False,
        Attribute "Date" (DateDom Nothing) NoKey False],
    Entity "Tag"
      [Attribute "Name" (StringDom Nothing) Unique False]
    ]
  [Relationship "Commenting"
    [REnd "Entry" "commentsOn" (Exactly 1),
      REnd "Comment" "isCommentedBy" (Between 0 Infinite)],
  Relationship "Tagging"
    [REnd "Entry" "tags" (Between 0 Infinite),
      REnd "Tag" "tagged" (Between 0 Infinite)]
  ]

Figure 6: The Curry program BlogERD.curry

There is also the possibility to visualize an ER specification as a graph with the graph visualization program dotty (for this purpose, it might be necessary to adapt the definition of dotviewcommand in your "..paksrc" file, see Section 2.6, according to your local environment). The visualization can be performed by the command

    erd2curry -v BlogERD.curry

76
16 Spicey: An ER-based Web Framework

Spicey is a framework to support the implementation of web-based systems in Curry. Spicey generates an initial implementation from an entity-relationship (ER) description of the underlying data. The generated implementation contains operations to create and manipulate entities of the data model, supports authentication, authorization, session handling, and the composition of individual operations to user processes. Furthermore, the implementation ensures the consistency of the database w.r.t. the data dependencies specified in the ER model, i.e., updates initiated by the user cannot lead to an inconsistent state of the database.

16.1 Installation

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

```bash
> cypm update
> cypm install spicey
```

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

16.2 Basic usage

The idea of this tool, which is part of the distribution of PAKCS, is described in detail in [24]. Thus, we summarize only the basic steps to use this tool in order to generate a web application.

First, one has to create a textual description of the entity-relationship model in a Curry program file as an (exported!) top-level operation type `ERD` (w.r.t. the type definitions defined in the module `Database.ERD` of the package `cdbi`) and store it in some program file, e.g., “MyERD.curry”. The directory `examples` in the package `spicey`¹⁰ contains two examples for such ERD program files:

- `BlogERD.curry`: This is a simple ER model for a blog with entries, comments, and tags, as presented in the paper [24].
- `UniERD.curry`: This is an ER model for university lectures as presented in the paper [15].

Then you can generate the sources of your web application by the command

```bash
> spiceup MyERD.curry
```

with the ERD program as a parameter. You can also provide a file name for the SQLite3 database used by the application generated by Spicey, e.g.,

```bash
> spiceup --db MyData.db MyERD.curry
```

If the parameter “--db DBFILE” is not provided, then `DBFILE` is set to the default name “`ERD.db`” (where `ERD` is the name of the specified ER model). Since this specification will be used in the generated web programs, a relative database file name will be relative to the place where the web

¹⁰If you installed Spicey as described above, the downloaded `spicey` package is located in the directory `$HOME/.cpm/app_packages/spicey`.  

77
programs are stored. In order to avoid such confusion, it might be better to specify an absolute path name for the database file. This path could also be set in the definition of the constant `sqliteDBFile` in the generated Curry program `Model/ERD.curry`.

Spicey generates the web application as a Curry package in a new directory. Thus, change into this directory (e.g., `cd ERD`) and install all required packages by the command

```
> make install
```

The generated file `README.txt` contains some information about the generated project structure. One can compile the generated programs by

```
> make compile
```

In order to generate the executable web application, configure the generated `Makefile` by adapting the variable `WEBSERVERDIR` to the location where the compiled cgi programs should be stored, and run

```
> make deploy
```

After the successful compilation and deployment of all files, the application is executable in a web browser by selecting the URL `<URL of web dir>/spicey.cgi`.

### 16.3 Further remarks

The application generated by Spicey is a schematic initial implementation. It provides an appropriate basic programming structure but it can be extended in various ways. In particular, one can also use embedded SQL statements (see [25] for details) when further developing the Curry code, since the underlying database access operations are generated with the `cdbi` package. The syntax and use of such embedded SQL statements is sketched in [25] and described in the Curry preprocessor.
17 curry-peval: A Partial Evaluator for Curry

peval is a tool for the partial evaluation of Curry programs. It operates on the FlatCurry representation and can thus easily be incorporated into the normal compilation chain. The essence of partial evaluation is to anticipate at compile time (or partial evaluation time) some of the computations normally performed at run time. Typically, partial evaluation is worthwhile for functions or operations where some of the input arguments are already known at compile time, or operations built by the composition of multiple other ones. The theoretical foundations, design and implementation of the partial evaluator is described in detail in [33].

17.1 Installation

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

```bash
> cypm update
> cypm install peval
```

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

17.2 Basic Usage

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

\[
\text{PEVAL :: a}
\]

\[
\text{PEVAL x = x}
\]

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

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

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

power4 x = PEVAL (power 4 x)
```

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

```bash
$ curry-peval Examples/power4.curry
Curry Partial Evaluator
```
Final Partial Evaluation
------------------------

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

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

```
$ curry-peval -v2 Examples/power4.curry
```

Resulting program
-----------------

```haskell
module power4 ( power4.square, power4.even, power4.power, power4.power4 ) where

import Prelude

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

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

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

power4.power4 :: Prelude.Int → Prelude.Int
```

80
power4.power4 v1 = power4._pe0 v1

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

17.3 Options

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

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

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

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

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

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

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

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

-S semantics, --semantics=semantics Allows the use to choose a semantics used during partial evaluation. Note that only the natural semantics can be considered correct for non-confluent programs, which is why it is the default semantics [33]. However, the rInt calculus can also be chosen which is based on term rewriting, thus implementing a run-time choice semantics [4]. The letrw semantics is currently not fully supported, but implements the gist of let-rewriting [31].

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

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

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

After the invocation of the Curry front end to parse Curry programs and translate them into the intermediate FlatCurry representation, one can apply transformations on the FlatCurry files before they are passed to the back end which translates the FlatCurry files into Prolog code. These transformations are invoked by the FlatCurry preprocessor `pakcs/bin/fycpp`. Currently, only the FlatCurry file corresponding to the main module can be transformed.

A transformation can be specified as follows:

1. **Options to `pakcs/bin/fcypp`:**
   - `--fpopt` Apply functional pattern optimization (see `pakcs/tools/optimize/NonStrictOpt.curry` for details).
   - `--compact` Apply code compactification after parsing, i.e., transform the main module and all its imported into one module and delete all non-accessible functions.
   - `--compactexport` Similar to `--compact` but delete all functions that are not accessible from the exported functions of the main module.
   - `--compactmain:f` Similar to `--compact` but delete all functions that are not accessible from the function "f" of the main module.
   - `--fcypp cmd` Apply command `cmd` to the main module after parsing. This is useful to integrate your own transformation into the compilation process. Note that the command "cmd prog" should perform a transformation on the FlatCurry file `prog.fcy`, i.e., it replaces the FlatCurry file by a new one.

2. **Setting the environment variable FCYPP:**
   For instance, setting FCYPP by
   ```bash
   export FCYPP="--fpopt"
   ```
   will apply the functional pattern optimization if programs are compiled and loaded in the PAKCS programming environment.

3. **Putting options into the source code:**
   If the source code contains a line with a comment of the form (the comment must start at the beginning of the line)
   ```
   {-# PAKCS_OPTION_FCYPP <options> #-}
   ```
   then the transformations specified by `<options>` are applied after translating the source code into FlatCurry code. For instance, the functional pattern optimization can be set by the comment
   ```
   {-# PAKCS_OPTION_FCYPP --fpopt #-}
   ```
   in the source code. Note that this comment must be in a single line of the source program. If there are multiple lines containing such comments, only the first one will be considered.
Multiple options: Note that an arbitrary number of transformations can be specified by the methods described above. If several specifications for preprocessing FlatCurry files are used, they are executed in the following order:

1. all transformations specified by the environment variable FCYPP (from left to right)
2. all transformations specified as command line options of fcyp (from left to right)
3. all transformations specified by a comment line in the source code (from left to right)
19 Technical Problems

19.1 SWI-Prolog

Using PAKCS with SWI-Prolog as its back end is slower than SICStus-Prolog and might cause some memory problems, since SWI-Prolog has stronger restrictions on the memory limits for the different stack areas when executing Prolog programs. For instance, if the compiled Curry program terminates with an error message like

```
ERROR: local
```

the Prolog system runs out of the local stack (although there might be enough memory available on the host machine).

In such a case, one can modify the script “\texttt{pakcshome/scripts/makesavedstate}” in order to change the SWI-Prolog default settings for memory limits of generated Curry applications.\footnote{Note that this script is generated during the installation of PAKCS. Hence, it might be necessary to redo the changes after a new installation of PAKCS.} This can be done by changing the definition of the variable \texttt{SWILIMITS} at the beginning of this script. Since different versions of SWI-Prolog have different command-line options, the correct setting depends on the version of SWI-Prolog:

**SWI-Prolog 7.*:** For instance, to set the maximum limit for the local stack to 2 GB (on 64bit machines, the default of SWI-Prolog is 1 GB), one can change the definition in this script to

```
SWILIMITS="-L2G -G0 -T0"
```

**SWI-Prolog 8.*:** For instance, to use 4 GB for all stacks (on 64bit machines, the default of SWI-Prolog is 1 GB), one can change the definition in this script to

```
SWILIMITS="--stack_limit=4g"
```

After this change, recompile (with the PAKCS command “:\texttt{save}”) the Curry application.

19.2 Distributed Programming and Sockets

Due to the fact that Curry is intended to implement distributed systems (see Appendix A.1.3), it might be possible that some technical problems arise due to the use of sockets for implementing these features. Therefore, this section gives some information about the technical requirements of PAKCS and how to solve problems due to these requirements.

There is one fixed port that is used by the implementation of PAKCS:

**Port 8769:** This port is used by the \textbf{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 \texttt{Ports} (see Appendix A.1.3) 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 “\texttt{netstat -a | grep 8769}” (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 Curry port or asking the physical port number of
a Curry port. The demon will be automatically started for the first time on a machine when a
user compiles a program using Curry ports. It can also be manually started and terminated by
the command `curry-cpnsd` (which is available by installing the package `cpns`, e.g., by the command
```
cypm install cpnsd
```
) If the demon is already running, the command “`curry-cpnsd start`” does
nothing (so it can be always executed before invoking a Curry program using ports).

### 19.3 Contact for Help

If you detect any further technical problem, please write to

pakcs@curry-lang.org
References

grams. In *Proc. of the 6th International Conference on Logic for Programming and Automated

logic programs. In *Proc. of the 7th International Conference on Logic for Programming and

ative language. In *Proc. of the 5th International Symposium on Functional and Logic Program-


of the International Symposium on Logic-based Program Synthesis and Transformation (LOP-

the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative

of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL

International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR

on Practical Aspects of Declarative Languages (PADL 2016)*, pages 65–82. Springer LNCS 9585,
2016.

14th International Symposium on Functional and Logic Programming (FLOPS 2018)*, pages

the 24th International Workshop on Functional and (Constraint) Logic Programming (WFLP


A Libraries of the PAKCS Distribution

The PAKCS distribution comes with an extensive collection of libraries for application programming. The libraries for arithmetic constraints over real numbers, finite domain constraints, ports for concurrent and distributed programming, and meta-programming by representing Curry programs 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 PAKCS, see http://www.informatik.uni-kiel.de/~pakcs/lib/index.html.

A.1 Constraints, Ports, Meta-Programming

A.1.1 Arithmetic Constraints

The primitive entities for the use of arithmetic constraints are defined in the system module CLPR (cf. Section 1.3), i.e., in order to use them, the program must contain the import declaration

\[
\text{import CLPR}
\]

Floating point arithmetic is supported in PAKCS via arithmetic constraints, i.e., the equational constraint “2.3 +. x ::= 5.5” is solved by binding x to 3.2 (rather than suspending the evaluation of the addition, as in corresponding constraints on integers like “3+x::=5”). All operations related to floating point numbers are suffixed by “.”. The following functions and constraints on floating point numbers are supported in PAKCS:

\[
(+.) :: \text{Float} \to \text{Float} \to \text{Float}
\]
Addition on floating point numbers.

\[
(-.) :: \text{Float} \to \text{Float} \to \text{Float}
\]
Subtraction on floating point numbers.

\[
(\times.) :: \text{Float} \to \text{Float} \to \text{Float}
\]
Multiplication on floating point numbers.

\[
(\div.) :: \text{Float} \to \text{Float} \to \text{Float}
\]
Division on floating point numbers.

\[
(<.) :: \text{Float} \to \text{Float} \to \text{Bool}
\]
Comparing two floating point numbers with the “less than” relation.

\[
(>.) :: \text{Float} \to \text{Float} \to \text{Bool}
\]
Comparing two floating point numbers with the “greater than” relation.

\[
(\leq .) :: \text{Float} \to \text{Float} \to \text{Bool}
\]
Comparing two floating point numbers with the “less than or equal” relation.

\[
(\geq .) :: \text{Float} \to \text{Float} \to \text{Bool}
\]
Comparing two floating point numbers with the “greater than or equal” relation.
i2f :: Int -> Float
Converting an integer number into a floating point number.

As an example, consider a constraint mortgage which relates the principal p, the lifetime of the mortgage in months t, the monthly interest rate ir, the monthly repayment r, and the outstanding balance at the end of the lifetime b. The financial calculations can be defined by the following two rules in Curry (the second rule describes the repeated accumulation of the interest):

```curry
import CLPR

mortgage p t ir r b | t <= 1 = b =:= p * (1.0 + t * ir) - t * r
mortgage p t ir r b | t > 1 = mortgage (p * (1.0 + ir) - r) (t - 1) ir r b
```

Then we can calculate the monthly payment for paying back a loan of $100,000 in 15 years with a monthly interest rate of 1% by solving the goal

```curry
mortgage 100000.0 180.0 0.01 r 0.0
```

which yields the solution $r=1200.17$.

Note that only linear arithmetic equalities or inequalities are solved by the constraint solver. Non-linear constraints like “x * x =:= 4.0” are suspended until they become linear.

### A.1.2 Finite Domain Constraints

Finite domain constraints are constraints where all variables can only take a finite number of possible values. For simplicity, the domain of finite domain variables are identified with a subset of the integers, i.e., the type of a finite domain variable is Int. The arithmetic operations related to finite domain variables are suffixed by “#”. The following functions and constraints for finite domain constraint solving are currently supported in PAKCS:\textsuperscript{12}

```curry
domain :: [Int] -> Int -> Int -> Bool
The constraint “domain [x₁,…,xₙ] l u” is satisfied if the domain of all variables xᵢ is the interval [l, u].

(#+) :: Int -> Int -> Int
Addition on finite domain values.

(-#) :: Int -> Int -> Int
Subtraction on finite domain values.

(*#) :: Int -> Int -> Int
Multiplication on finite domain values.

(=#) :: Int -> Int -> Bool
Equality of finite domain values.
```

\textsuperscript{12}Note that this library is based on the corresponding library of SICStus-Prolog but does not implement the complete functionality of the SICStus-Prolog library. However, using the PAKCS interface for external functions (see Appendix F), it is relatively easy to provide the complete functionality.
(/=#) :: Int -> Int -> Bool
  Disequality of finite domain values.

(<#) :: Int -> Int -> Bool
  “less than” relation on finite domain values.

(<=#) :: Int -> Int -> Bool
  “less than or equal” relation on finite domain values.

(>#) :: Int -> Int -> Bool
  “greater than” relation on finite domain values.

(>=#) :: Int -> Int -> Bool
  “greater than or equal” relation on finite domain values.

sum :: [Int] -> (Int -> Int -> Bool) -> Int -> Bool
  The constraint “sum [x_1,...,x_n] op x” is satisfied if all x_1 + ... + x_n op x is satisfied, where
  op is one of the above finite domain constraint relations (e.g., “=#”).

scalar_product :: [Int] -> [Int] -> (Int -> Int -> Bool) -> Int -> Bool
  The constraint “scalar_product [c_1,...,c_n] [x_1,...,x_n] op x” is satisfied if all c_1x_1 + ... +
  c_nx_n op x is satisfied, where op is one of the above finite domain constraint relations.

count :: Int -> [Int] -> (Int -> Int -> Bool) -> Int -> Bool
  The constraint “count k [x_1,...,x_n] op x” is satisfied if all k op x is satisfied, where n is
  the number of the x_i that are equal to k and op is one of the above finite domain constraint
  relations.

allDifferent :: [Int] -> Bool
  The constraint “allDifferent [x_1,...,x_n]” is satisfied if all x_i have pairwise different values.

labeling :: [LabelingOption] -> [Int] -> Bool
  The constraint “labeling os [x_1,...,x_n]” non-deterministically instantiates all x_i to the val-
  ues of their domain according to the options os (see the module documentation for further
details about these options).

These entities are defined in the system module CLPFD (cf. Section 1.3), i.e., in order to use it, the
program must contain the import declaration

import CLPFD

As an example, consider the classical “send+more=money” problem where each letter must be replaced
by a different digit such that this equation is valid and there are no leading zeros. The usual way to
solve finite domain constraint problems is to specify the domain of the involved variables followed
by a specification of the constraints and the labeling of the constraint variables in order to start the
search for solutions. Thus, the “send+more=money” problem can be solved as follows:

import CLPFD

smm l =
l := [s,e,n,d,m,o,r,y] &
domain l 0 9 &
s ># 0 &
m ># 0 &
allDifferent l &

1000 *# s +# 100 *# e +# 10 *# n +# d +# 1000 *# m +# 100 *# o +# 10 *# r +# e =# 10000 *# m +# 1000 *# o +# 100 *# n +# 10 *# e +# y &

labeling [FirstFail] l
where s,e,n,d,m,o,r,y free

Then we can solve this problem by evaluating the goal “smm [s,e,n,d,m,o,r,y]” which yields the unique solution \{s=9,e=5,n=6,d=7,m=1,o=0,r=8,y=2\}.

A.1.3 Ports: Distributed Programming in Curry

To support the development of concurrent and distributed applications, PAKCS supports internal and external ports as described in [19]. Since [19] contains a detailed description of this concept together with various programming examples, we only summarize here the functions and constraints supported for ports in PAKCS.

The basic datatypes, functions, and constraints for ports are defined in the system module Ports (cf. Section 1.3), i.e., in order to use ports, the program must contain the import declaration

import Ports

This declaration includes the following entities in the program:

Port a

This is the datatype of a port to which one can send messages of type a.

openPort :: Port a -> [a] -> Bool
The constraint “openPort p s” establishes a new internal port p with an associated message stream s. p and s must be unbound variables, otherwise the constraint fails (and causes a runtime error).

send :: a -> Port a -> Bool
The constraint “send m p” is satisfied if p is constrained to contain the message m, i.e., m will be sent to the port p so that it appears in the corresponding stream.

doSend :: a -> Port a -> IO ()
The I/O action “doSend m p” solves the constraint “send m p” and returns nothing.

openNamedPort :: String -> IO [a]
The I/O action “openNamedPort n” opens a new external port with symbolic name n and returns the associated stream of messages.

connectPort :: String -> IO (Port a)
The I/O action “connectPort n” returns a port with symbolic name n (i.e., n must have the form “portname@machine”) to which one can send messages by the send constraint. Currently,
no dynamic type checking is done for external ports, i.e., sending messages of the wrong type
to a port might lead to a failure of the receiver.

Restrictions: Every expression, possibly containing logical variables, can be sent to a port. How-
ever, as discussed in [19], port communication is strict, i.e., the expression is evaluated to normal
form before sending it by the constraint send. Furthermore, if messages containing logical variables
are sent to external ports, the behavior is as follows:

1. The sender waits until all logical variables in the message have been bound by the receiver.

2. The binding of a logical variable received by a process is sent back to the sender of this logical
variable only if it is bound to a ground term, i.e., as long as the binding contains logical
variables, the sender is not informed about the binding and, therefore, the sender waits.

External ports on local machines: The implementation of external ports assumes that the
host machine running the application is connected to the Internet (i.e., it uses the standard IP
address of the host machine for message sending). If this is not the case and the application should
be tested by using external ports only on the local host without a connection to the Internet, the
environment variable "PAKCS_LOCALHOST" must be set to “yes” before PAKCS is started. In this case,
the IP address 127.0.0.1 and the hostname "localhost" are used for identifying the local machine.

Selection of Unix sockets for external ports: The implementation of ports uses sockets
to communicate messages sent to external ports. Thus, if a Curry program uses the I/O action
openNamedPort to establish an externally visible server, PAKCS selects a Unix socket for the port
communication. Usually, a free socket is selected by the operating system. If the socket number
should be fixed in an application (e.g., because of the use of firewalls that allow only communication
over particular sockets), then one can set the environment variable “PAKCS_SOCKET” to a distinguished
socket number before PAKCS is started. This has the effect that PAKCS uses only this socket
number for communication (even for several external ports used in the same application program).

Debugging: To debug distributed systems, it is sometimes helpful to see all messages sent to
external ports. This is supported by the environment variable “PAKCS_TRACEPORTS”. If this variable
is set to “yes” before PAKCS is started, then all connections to external ports and all messages sent
and received on external ports are printed on the standard error stream.

A.1.4 AbstractCurry and FlatCurry: Meta-Programming in Curry

To support meta-programming, i.e., the manipulation of Curry programs in Curry, there are Curry
packages flatcurry and abstractcurry which define datatypes for the representation of Curry pro-
grams. AbstractCurry.Types (package abstractcurry) is a more direct representation of a Curry
program, whereas FlatCurry.Types (package flatcurry) is a simplified representation where local
function definitions are replaced by global definitions (i.e., lambda lifting has been performed) and
pattern matching is translated into explicit case/or expressions. Thus, FlatCurry.Types can be
used for more back-end oriented program manipulations (or, for writing new back ends for Curry),
whereas `AbstractCurry.Types` is intended for manipulations of programs that are more oriented towards the source program.

There are predefined I/O actions to read AbstractCurry and FlatCurry programs: `AbstractCurry.Files.readCurry` and `FlatCurry.Files.readFlatCurry`. These actions parse the corresponding source program and return a data term representing this program (according to the definitions in the modules `AbstractCurry.Types` and `FlatCurry.Types`). Since all datatypes are explained in detail in these modules, we refer to the online documentation of these modules.

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

```haskell
rev [] = []
rev (x:xs) = (rev xs) ++ [x]
```

Then the I/O action `(FlatCurry.Files.readFlatCurry "test")` returns the following term:

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

### A.2 System Libraries

#### A.2.1 Library Control.Applicative

Exported functions:

- **liftA :: Applicative a ⇒ (c → b) → Apply a c → Apply a b**

  Lift a function to actions. This function may be used as a value for `fmap` in a `Functor` instance.

- **liftA3 :: Applicative c ⇒ (b → a → e → d) → Apply c b → Apply c a → Apply c e → Apply c d**

---

13 [http://www.informatik.uni-kiel.de/~pakcs/lib/FlatCurry.Types.html](http://www.informatik.uni-kiel.de/~pakcs/lib/FlatCurry.Types.html) and [http://www.informatik.uni-kiel.de/~pakcs/lib/AbstractCurry.Types.html](http://www.informatik.uni-kiel.de/~pakcs/lib/AbstractCurry.Types.html)
Lift a ternary function to actions.

\[
\text{when} :: \text{Applicative } a \Rightarrow \text{Bool} \to \text{Apply } a () \to \text{Apply } a ()
\]

\[
\text{sequenceA} :: \text{Applicative } b \Rightarrow [\text{Apply } b a] \to \text{Apply } b [a]
\]

Evaluate each action in the list from left to right, and collect the results. For a version that ignores the results see `sequenceA_`.

\[
\text{sequenceA} :: \text{Applicative } a \Rightarrow [\text{Apply } a b] \to \text{Apply } a ()
\]

Evaluate each action in the structure from left to right, and ignore the results. For a version that doesn’t ignore the results see `sequenceA`.

### A.2.2 Library Control.Monad

**Exported functions:**

\[
\text{filterM} :: \text{Applicative } b \Rightarrow (a \to \text{Apply } b \text{ Bool}) \to [a] \to \text{Apply } b [a]
\]

This generalizes the list-based `filter` function.

\[
(\Rightarrow\Rightarrow) :: \text{Monad } c \Rightarrow (d \to \text{Apply } c b) \to (b \to \text{Apply } c a) \to d \to \text{Apply } c a
\]

Left-to-right composition of Kleisli arrows.

\[
(<\Rightarrow) :: \text{Monad } c \Rightarrow (b \to \text{Apply } c a) \to (d \to \text{Apply } c b) \to d \to \text{Apply } c a
\]

Right-to-left composition of Kleisli arrows. @(`(\Rightarrow\Rightarrow)`@, with the arguments flipped.

\[
\text{forever} :: \text{Applicative } c \Rightarrow \text{Apply } c b \to \text{Apply } c a
\]

Repeat an action indefinitely.

\[
\text{mapAndUnzipM} :: \text{Applicative } b \Rightarrow (a \to \text{Apply } b (d,c)) \to [a] \to \text{Apply } b ([d],[c])
\]

The `mapAndUnzipM` function maps its first argument over a list, returning the result as a pair of lists. This function is mainly used with complicated data structures or a state-transforming monad.

\[
\text{zipWithM} :: \text{Applicative } d \Rightarrow (b \to a \to \text{Apply } d c) \to [b] \to [a] \to \text{Apply } d [c]
\]

The `zipWithM` function generalizes `zipWith` to arbitrary applicative functors.

\[
\text{zipWithM} :: \text{Applicative } d \Rightarrow (b \to a \to \text{Apply } d c) \to [b] \to [a] \to \text{Apply } d ()
\]

`zipWithM` is the extension of `<zipWithM>` which ignores the final result.

\[
\text{foldM} :: \text{Monad } a \Rightarrow (c \to b \to \text{Apply } a c) \to c \to [b] \to \text{Apply } a c
\]
The `codefoldM/code` function is analogous to `codefoldl/code`, except that its result is encapsulated in a monad.

\[
\text{foldM} \_ :: \text{Monad a} \Rightarrow (c \to b \to \text{Apply a c}) \to c \to [b] \to \text{Apply a ()}
\]

Like `codefoldM/code`, but discards the result.

\[
\text{replicateM} :: \text{Applicative b} \Rightarrow \text{Int} \to \text{Apply b a} \to \text{Apply b [a]}
\]

Like `codereplicateM/code`, but discards the result.

\[
\text{unless} :: \text{Applicative a} \Rightarrow \text{Bool} \to \text{Apply a ()} \to \text{Apply a ()}
\]

The reverse of `codewhen/code`.

\[
\text{liftM3} :: \text{Monad b} \Rightarrow (e \to d \to c \to a) \to \text{Apply b e} \to \text{Apply b d} \to \text{Apply b c} \to \text{Apply b a}
\]

### A.2.3 Library Curry.Compiler.Distribution

This module contains definition of constants to obtain information concerning the current distribution of the Curry implementation, e.g., compiler version, run-time version, installation directory.

Exported functions:

- `curryCompiler :: String`
  
The name of the Curry compiler (e.g., "pakcs" or "kics2").

- `curryCompilerMajorVersion :: Int`
  
The major version number of the Curry compiler.

- `curryCompilerMinorVersion :: Int`
  
The minor version number of the Curry compiler.

- `curryCompilerRevisionVersion :: Int`
  
The revision version number of the Curry compiler.

- `curryRuntime :: String`
  
The name of the run-time environment (e.g., "sicstus", "swi", or "ghc").

- `curryRuntimeMajorVersion :: Int`
  
The major version number of the Curry run-time environment.
curryRuntimeMinorVersion :: Int

The minor version number of the Curry run-time environment.

baseVersion :: String

The version number of the base libraries (e.g., "1.0.5").

installDir :: String

Path of the main installation directory of the Curry compiler.

A.2.4 Library Data.Char

Library with some useful functions on characters.

Exported functions:

isAscii :: Char → Bool

Returns true if the argument is an ASCII character.

isLatin1 :: Char → Bool

Returns true if the argument is an Latin-1 character.

isAsciiLower :: Char → Bool

Returns true if the argument is an ASCII lowercase letter.

isAsciiUpper :: Char → Bool

Returns true if the argument is an ASCII uppercase letter.

isControl :: Char → Bool

Returns true if the argument is a control character.

toUpper :: Char → Char

Converts lowercase into uppercase letters.

toLower :: Char → Char

Converts uppercase into lowercase letters.

digitToInt :: Char → Int

Converts a (hexadecimal) digit character into an integer.

intToDigit :: Int → Char

Converts an integer into a (hexadecimal) digit character.
A.2.5 Library Data.Either

Library with some useful operations for the Either data type.

Exported functions:

lefts :: [Either b a] → [b]
   Extracts from a list of Either all the Left elements in order.

rights :: [Either a b] → [b]
   Extracts from a list of Either all the Right elements in order.

isLeft :: Either a b → Bool
   Return True if the given value is a Left-value, False otherwise.

isRight :: Either a b → Bool
   Return True if the given value is a Right-value, False otherwise.

fromLeft :: Either b a → b
   Extract the value from a Left constructor.

fromRight :: Either a b → b
   Extract the value from a Right constructor.

partitionEithers :: [Either b a] → ([b],[a])
   Partitions a list of Either into two lists. All the Left elements are extracted, in order, to the first component of the output. Similarly the Right elements are extracted to the second component of the output.

A.2.6 Library Data.Function

This module provides some utility functions for function application.

Exported functions:

fix :: (a → a) → a
   fix f is the least fixed point of the function f, i.e. the least defined x such that f x = x.

on :: (a → a → b) → (c → a) → c → c → b
   (* ) ‘on’ f = \x y ->&gt; f x * f y. Typical usage: sortBy (compare ‘on’ fst).
A.2.7 Library Data.Functor.Identity

Exported types:

\[
\text{type Identity } a = a
\]

Exported functions:

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

A.2.8 Library Data.IORef

Library with some useful extensions to the IO monad.

Exported types:

\[
data IORef
\]

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

Exported constructors:

Exported functions:

\[
\text{newIORef} :: a \rightarrow \text{IO } (\text{IORef } a)
\]

Creates a new IORef with an initial value.

\[
\text{readIORef} :: \text{IORef } a \rightarrow \text{IO } a
\]

Reads the current value of an IORef.

\[
\text{writeIORef} :: \text{IORef } a \rightarrow a \rightarrow \text{IO } ()
\]

Updates the value of an IORef.

\[
\text{modifyIORef} :: \text{IORef } a \rightarrow (a \rightarrow a) \rightarrow \text{IO } ()
\]

Modify the value of an IORef.

A.2.9 Library Data.List

Library with some useful operations on lists.
Exported functions:

elemIndex :: Eq a ⇒ a → [a] → Maybe Int

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

elemIndices :: Eq a ⇒ a → [a] → [Int]

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

find :: (a → Bool) → [a] → Maybe a

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

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

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

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

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

nub :: Eq a ⇒ [a] → [a]

    Removes all duplicates in the argument list.

nubBy :: (a → a → Bool) → [a] → [a]

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

delete :: Eq a ⇒ a → [a] → [a]

    Deletes the first occurrence of an element in a list.

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

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

(\) :: Eq a ⇒ [a] → [a] → [a]

    Computes the difference of two lists.

union :: Eq a ⇒ [a] → [a] → [a]

    Computes the union of two lists.

unionBy :: (a → a → Bool) → [a] → [a] → [a]

    Computes the union of two lists according to the given equivalence relation

intersect :: Eq a ⇒ [a] → [a] → [a]
Computes the intersection of two lists.

**intersectBy :: (a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow [a] \rightarrow [a]**

Computes the intersection of two lists according to the given equivalence relation

**intersperse :: a \rightarrow [a] \rightarrow [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] \rightarrow [[a]] \rightarrow [a]**

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

**transpose :: [[a]] \rightarrow [[a]]**

Transposes the rows and columns of the argument.

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

**diagonal :: [[a]] \rightarrow [a]**

Diagonalization of a list of lists. Fairly merges (possibly infinite) list of (possibly infinite) lists.

**permutations :: [a] \rightarrow [[a]]**

Returns the list of all permutations of the argument.

**partition :: (a \rightarrow Bool) \rightarrow [a] \rightarrow ([a],[a])**

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

Example: \((\text{partition } (\&lt;4) [8,1,5,2,4,3]) = ([1,2,3],[8,5,4])\)

**group :: Eq a \Rightarrow [a] \rightarrow [[a]]**

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

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

**groupBy :: (a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow [[a]]**

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

**splitOn :: Eq a \Rightarrow [a] \rightarrow [a] \rightarrow [[a]]**

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

**split :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [[a]]**
Splits a list into components delimited by separators, where the predicate returns True for a separator element. The resulting components do not contain the separators. Two adjacent separators result in an empty component in the output.

\[
\text{split} \quad (\text{==codea/code}) \quad \text{"aabbaca"} \quad \Rightarrow \quad ["","","b","c",""]
\]

\[
\text{split} \quad (\text{==codea/code}) \quad \text{""} \quad \Rightarrow \quad ["]
\]

\[
i\text{ints} :: [a] \rightarrow [[a]]
\]

Returns all initial segments of a list, starting with the shortest. Example: \(\text{ints} \quad [1,2,3] \Rightarrow [[],[1],[1,2],[1,2,3]]\)

\[
i\text{tails} :: [a] \rightarrow [[a]]
\]

Returns all final segments of a list, starting with the longest. Example: \(\text{tails} \quad [1,2,3] \Rightarrow [[1,2,3],[2,3],[3],[\text{\textquoteleft\textquoteleft}]]\)

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

Replaces an element in a list.

\[
i\text{sprefixOf} :: \text{Eq}\ a \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}
\]

Checks whether a list is a prefix of another.

\[
i\text{suffixOf} :: \text{Eq}\ a \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}
\]

Checks whether a list is a suffix of another.

\[
i\text{infixOf} :: \text{Eq}\ a \Rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}
\]

Checks whether a list is contained in another.

\[
i\text{sort} :: \text{Ord}\ a \Rightarrow [a] \rightarrow [a]
\]

The default sorting operation, mergeSort, with standard ordering \&lt;=.

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

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

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

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

\[
i\text{last} :: [a] \rightarrow a
\]

Returns the last element of a non-empty list.

\[
i\text{init} :: [a] \rightarrow [a]
\]

Returns the input list with the last element removed.

\[
i\text{sum} :: \text{Num}\ a \Rightarrow [a] \rightarrow a
\]
Returns the sum of a list of integers.

\[
\text{product} :: \text{Num } a \Rightarrow [a] \rightarrow a
\]

Returns the product of a list of integers.

\[
\text{maximum} :: \text{Ord } a \Rightarrow [a] \rightarrow a
\]

Returns the maximum of a non-empty list.

\[
\text{maximumBy} :: (a \rightarrow a \rightarrow \text{Ordering}) \rightarrow [a] \rightarrow a
\]

Returns the maximum of a non-empty list according to the given comparison function

\[
\text{minimum} :: \text{Ord } a \Rightarrow [a] \rightarrow a
\]

Returns the minimum of a non-empty list.

\[
\text{minimumBy} :: (a \rightarrow a \rightarrow \text{Ordering}) \rightarrow [a] \rightarrow a
\]

Returns the minimum of a non-empty list according to the given comparison function

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

\text{scanl} is similar to \text{foldl}, but returns a list of successive reduced values from the left:
\[
\text{scanl } f \ z \ [x1, \ x2, \ ...] = [z, \ z \ f \ x1, \ (z \ f \ x1) \ f \ x2, \ ...]
\]

\[
\text{scanl1} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]
\]

\text{scanl1} is a variant of \text{scanl} that has no starting value argument:
\[
\text{scanl1 } f \ [x1, \ x2, \ ...] = [x1, \ x1 \ f \ x2, \ ...]
\]

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

\text{scanr} is the right-to-left dual of \text{scanl}.

\[
\text{scanr1} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a]
\]

\text{scanr1} is a variant of \text{scanr} that has no starting value argument.

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

The \text{mapAccumL} function behaves like a combination of \text{map} and \text{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.

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

The \text{mapAccumR} function behaves like a combination of \text{map} and \text{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.

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

Builds an infinite list from a finite one.

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

Builds a list from a seed value.
A.2.10 Library Data.Maybe

Library with some useful functions on the Maybe datatype.

Exported functions:

isJust :: Maybe a → Bool
Return True iff the argument is of the form Just _.

isNothing :: Maybe a → Bool
Return True iff the argument is of the form Nothing.

fromJust :: Maybe a → a
Extract the argument from the Just constructor and throw an error if the argument is Nothing.

fromMaybe :: a → Maybe a → a
Extract the argument from the Just constructor or return the provided default value if the argument is Nothing.

listToMaybe :: [a] → Maybe a
Return Nothing on an empty list or Just x where x is the first list element.

maybeToList :: Maybe a → [a]
Return an empty list for Nothing or a singleton list for Just x.

catMaybes :: [Maybe a] → [a]
Return the list of all Just values.

mapMaybe :: (a → Maybe b) → [a] → [b]
Apply a function which may throw out elements using the Nothing constructor to a list of elements.

A.2.11 Library Debug.Trace

This library contains some useful operation for debugging programs.

Exported functions:

trace :: String → a → a
Prints the first argument as a side effect and behaves as identity on the second argument.

traceId :: String → String
Prints the first argument as a side effect and returns it afterwards.
traceShow :: Show b ⇒ b → a → a

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

traceShowId :: Show a ⇒ a → a

Prints the first argument using show and returns it afterwards.

traceIO :: String → IO ()

Output a trace message from the IO monad.

assert :: Bool → String → a → a

Assert a condition w.r.t. an error message. If the condition is not met it fails with the given error message, otherwise the third argument is returned.

assertIO :: Bool → String → IO ()

Assert a condition w.r.t. an error message from the IO monad. If the condition is not met it fails with the given error message.

A.2.12 Library Numeric

Library with some functions for reading and converting numeric tokens.

Exported functions:

readInt :: String → [(Int,String)]

Read a (possibly negative) integer as a first token in a string. The string might contain leadings blanks and the integer is read up to the first non-digit. On success returns [(v,s)], where v is the value of the integer and s is the remaining string without the integer token.

readNat :: String → [(Int,String)]

Read a natural number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-digit. On success returns [(v,s)], where v is the value of the number and s is the remaining string without the number token.

readHex :: String → [(Int,String)]

Read a hexadecimal number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-hexadecimal digit. On success returns [(v,s)], where v is the value of the number and s is the remaining string without the number token.

readOct :: String → [(Int,String)]
Read an octal number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-octal digit. On success returns \([(v,s)]\), where \(v\) is the value of the number and \(s\) is the remaining string without the number token.

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

Read a binary number as a first token in a string. The string might contain leadings blanks and the number is read up to the first non-binary digit. On success returns \([(v,s)]\), where \(v\) is the value of the number and \(s\) is the remaining string without the number token.

### A.2.13 Library Prelude

The standard prelude of Curry with type classes. All exported functions, data types, type classes and methods defined in this module are always available in any Curry program.

**Exported types:**

**data ()**

*Exported constructors:*

- \((::) :: ()\)

**data (,)**

*Exported constructors:*

- \((, :: a \rightarrow b \rightarrow (,) a b\)

**data (,,)**

*Exported constructors:*

- \((,,,, :: a \rightarrow b \rightarrow c \rightarrow (,,) a b c\)

**data (,,,)**

*Exported constructors:*

- \((,,,,,,, :: a \rightarrow b \rightarrow c \rightarrow d \rightarrow (,,,,) a b c d\)
data (,,,,)

Exported constructors:

- (,,,,) :: a → b → c → d → e → (,,,,) a b c d e

data (,,,,)

Exported constructors:

- (,,,,,) :: a → b → c → d → e → f → (,,,,,) a b c d e f

data (,,,,)

Exported constructors:

- (,,,,,,) :: a → b → c → d → e → f → g → (,,,,,,) a b c d e f g

data (,,,,)

Exported constructors:

- (,,,,,,,,) :: a → b → c → d → e → f → g → h → (,,,,,,,,) a b c d e f g h

data (,,,,,,,,)

Exported constructors:

- (,,,,,,,,,) :: a → b → c → d → e → f → g → h → i → (,,,,,,,,,) a b c d e f g h i

data (,,,,,,,,)

Exported constructors:

- (,,,,,,,,,,) :: a → b → c → d → e → f → g → h → i → j → (,,,,,,,,,,) a b c d e f g h i j

data (,,,,,,,,,)

Exported constructors:
Exported constructors:

• \((,,,,,,,,,,,) :: a \to b \to c \to d \to e \to f \to g \to h \to i \to j \to k \to (,,,,,,,,,,,) a b c d e f g h i j k\)

data \((,,,,,,,,,,,)\)

Exported constructors:

• \((,,,,,,,,,,,,) :: a \to b \to c \to d \to e \to f \to g \to h \to i \to j \to k \to l \to (,,,,,,,,,,,,) a b c d e f g h i j k l\)

data \((,,,,,,,,,,,,,)\)

Exported constructors:

• \((,,,,,,,,,,,,,,) :: a \to b \to c \to d \to e \to f \to g \to h \to i \to j \to k \to l \to m \to (,,,,,,,,,,,,,,) a b c d e f g h i j k l m\)

data \((,,,,,,,,,,,,,,,)\)

Exported constructors:

• \((,,,,,,,,,,,,,,,) :: a \to b \to c \to d \to e \to f \to g \to h \to i \to j \to k \to l \to m \to n \to (,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n\)

data \((,,,,,,,,,,,,,,,)\)

Exported constructors:

• \((,,,,,,,,,,,,,,,,) :: a \to b \to c \to d \to e \to f \to g \to h \to i \to j \to k \to l \to m \to n \to o \to (,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o\)

data \((->)\)

Exported constructors:

data Bool
Exported constructors:

- False :: Bool
- True :: Bool

data Char

Exported constructors:

data Either

Exported constructors:

- Left :: a → Either a b
- Right :: b → Either a b

data FilePath = String

data Float

Exported constructors:

data IO

Exported constructors:

data IOError

The (abstract) type of error values. Currently, it distinguishes between general I/O errors, user-generated errors (see `code`userError/code), failures and non-determinism errors during I/O computations. These errors can be caught by `code`catch/code. Each error contains a string shortly explaining the error. This type might be extended in the future to distinguish further error situations.

Exported constructors:

- IOError :: String → IOError
- UserError :: String → IOError
• FailError :: String → IOError
• NondetError :: String → IOError

data Int

Exported constructors:

data Maybe

Exported constructors:
• Nothing :: Maybe a
• Just :: a → Maybe a

data Ordering

Exported constructors:
• LT :: Ordering
• EQ :: Ordering
• GT :: Ordering

type ReadS a = String → [(a,String)]

type ShowS = String → String

type String = [Char]

type Success = Bool

data []

Exported constructors:
• [] :: [] a
• :: :: a → [a] → [] a
Exported functions:

shows :: Show a ⇒ a → String → String

showChar :: Char → String → String

showString :: String → String → String

showParen :: Bool → (String → String) → String → String

reads :: Read a ⇒ String → [(a,String)]

readParen :: Bool → (String → [(a,String)]) → String → [(a,String)]

read :: Read a ⇒ String → a

lex :: String → [(String,String)]

even :: Integral a ⇒ a → Bool

    Returns whether an integer is even.

odd :: Integral a ⇒ a → Bool

    Returns whether an integer is odd.

fromIntegral :: Integral a ⇒ Num b ⇒ a → b

    General coercion from integral types.

realToFrac :: Real a ⇒ Fractional b ⇒ a → b

    General coercion to fractional types.

(^) :: Num b ⇒ Integral a ⇒ b → a → b

(<$>) :: Functor c ⇒ (b → a) → Apply c b → Apply c a
liftM2 :: Monad b ⇒ (d → c → a) → Apply b d → Apply b c → Apply b a

sequence :: Monad a ⇒ [Apply a b] → Apply a [b]

Executes a sequence of monadic actions and collects all results in a list.

sequence_ :: Monad a ⇒ [Apply a b] → Apply a ()

Executes a sequence of monadic actions and ignores the results.

mapM :: Monad c ⇒ (a → Apply c b) → [a] → Apply c [b]

Maps a monadic action function on a list of elements. The results of all monadic actions are collected in a list.

mapM_ :: Monad c ⇒ (a → Apply c b) → [a] → Apply c ()

Maps an monadic action function on a list of elements. The results of all monadic actions are ignored.

isUpper :: Char → Bool

Returns true if the argument is an uppercase letter.

isLower :: Char → Bool

Returns true if the argument is a lowercase letter.

isAlpha :: Char → Bool

Returns true if the argument is a letter.

isDigit :: Char → Bool

Returns true if the argument is a decimal digit.

isAlphaNum :: Char → Bool

Returns true if the argument is a letter or digit.

isBinDigit :: Char → Bool

Returns true if the argument is a binary digit.

isOctDigit :: Char → Bool

Returns true if the argument is an octal digit.

isHexDigit :: Char → Bool

Returns true if the argument is a hexadecimal digit.
isSpace :: Char → Bool

  Returns true if the argument is a white space.

ord :: Char → Int

  Converts a character into its ASCII value.

chr :: Int → Char

  Converts a Unicode value into a character. The conversion is total, i.e., for out-of-bound values, the smallest or largest character is generated.

lines :: String → [String]

  Breaks a string into a list of lines where a line is terminated at a newline character. The resulting lines do not contain newline characters.

unlines :: [String] → String

  Concatenates a list of strings with terminating newlines.

words :: String → [String]

  Breaks a string into a list of words where the words are delimited by white spaces.

unwords :: [String] → String

  Concatenates a list of strings with a blank between two strings.

($) :: (b → a) → b → a

  Right-associative application.

($)! :: (a → b) → a → b

  Right-associative application with strict evaluation of its argument to head normal form.

($)!! :: (a → b) → a → b

  Right-associative application with strict evaluation of its argument to normal form.

($)# :: (b → a) → b → a

  Right-associative application with strict evaluation of its argument to a non-variable term.

($)## :: (a → b) → a → b

  Right-associative application with strict evaluation of its argument to ground normal form.

seq :: b → a → a
Evaluates the first argument to head normal form (which could also be a free variable) and returns the second argument.

```haskell
ensureNotFree :: a → a
```
Evaluates the argument to head normal form and returns it. Suspends until the result is bound to a non-variable term.

```haskell
ensureSpine :: [a] → [a]
```
Evaluates the argument to spine form and returns it. Suspends until the result is bound to a non-variable spine.

```haskell
normalForm :: a → a
```
Evaluates the argument to normal form and returns it.

```haskell
groundNormalForm :: a → a
```
Evaluates the argument to ground normal form and returns it. Suspends as long as the normal form of the argument is not ground.

```haskell
(.) :: (a → b) → (c → a) → c → b
```
Function composition.

```haskell
id :: a → a
```
Identity function.

```haskell
const :: b → a → b
```
Constant function.

```haskell
asTypeOf :: a → a → a
```
asTypeOf is a type-restricted version of `const`. It is usually used as an infix operator, and its typing forces its first argument (which is usually overloaded) to have the same type as the second.

```haskell
curry :: ((c,b) → a) → c → b → a
```
Converts an uncurried function to a curried function.

```haskell
uncurry :: (c → b → a) → (c,b) → a
```
Converts an curried function to a function on pairs.

```haskell
flip :: (b → c → a) → c → b → a
```
flip f is identical to f, but with the order of arguments reversed.

```haskell
until :: (a → Bool) → (a → a) → a → a
```
Repeats application of a function until a predicate holds.

(&&) :: Bool → Bool → Bool

Sequential conjunction on Booleans.

(||) :: Bool → Bool → Bool

Sequential disjunction on Booleans.

not :: Bool → Bool

Negation on Booleans.

otherwise :: Bool

Useful name for the last condition in a sequence of conditional equations.

ifThenElse :: Bool → a → a → a

The standard conditional. It suspends if the condition is a free variable.

fst :: (b,a) → b

Selects the first component of a pair.

snd :: (b,a) → a

Selects the second component of a pair.

head :: [a] → a

Computes the first element of a list.

tail :: [a] → [a]

Computes the remaining elements of a list.

null :: [a] → Bool

Is a list empty?

(++) :: [a] → [a] → [a]

Concatenates two lists. Since it is flexible, it could be also used to split a list into two sublists etc.

length :: [a] → Int

Computes the length of a list.

(!!) :: [a] → Int → a

List index (subscript) operator, head has index 0.

map :: (b → a) → [b] → [a]
Maps a function on all elements of a list.

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

Accumulates all list elements by applying a binary operator from left to right.

\[ \text{foldl1} :: (a \to a \to a) \to [a] \to a \]

Accumulates a non-empty list from left to right.

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

Accumulates all list elements by applying a binary operator from right to left.

\[ \text{foldr1} :: (a \to a \to a) \to [a] \to a \]

Accumulates a non-empty list from right to left:

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

Filters all elements satisfying a given predicate in a list.

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

Joins two lists into one list of pairs. If one input list is shorter than the other, the additional elements of the longer list are discarded.

\[ \text{zip3} :: [c] \to [b] \to [a] \to [(c,b,a)] \]

Joins three lists into one list of triples. If one input list is shorter than the other, the additional elements of the longer lists are discarded.

\[ \text{zipWith} :: (c \to b \to a) \to [c] \to [b] \to [a] \]

Joins two lists into one list by applying a combination function to corresponding pairs of elements. Thus \( \text{zip} = \text{zipWith} (,) \)

\[ \text{zipWith3} :: (d \to c \to b \to a) \to [d] \to [c] \to [b] \to [a] \]

Joins three lists into one list by applying a combination function to corresponding triples of elements. Thus \( \text{zip3} = \text{zipWith3} (,,) \)

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

Transforms a list of pairs into a pair of lists.

\[ \text{unzip3} :: [(c,b,a)] \to ([c],[b],[a]) \]

Transforms a list of triples into a triple of lists.

\[ \text{concat} :: [[a]] \to [a] \]

Concatenates a list of lists into one list.

\[ \text{concatMap} :: (a \to [b]) \to [a] \to [b] \]
Maps a function from elements to lists and merges the result into one list.

\[ \text{iterate} :: (a \to a) \to a \to [a] \]

Infinite list of repeated applications of a function \( f \) to an element \( x \). Thus, \( \text{iterate} f \ x = [x, f \ x, f (f \ x), \ldots] \).

\[ \text{repeat} :: a \to [a] \]

Infinite list where all elements have the same value. Thus, \( \text{repeat} \ x = [x, x, x, \ldots] \).

\[ \text{replicate} :: \text{Int} \to a \to [a] \]

List of length \( n \) where all elements have the same value.

\[ \text{take} :: \text{Int} \to [a] \to [a] \]

Returns prefix of length \( n \).

\[ \text{drop} :: \text{Int} \to [a] \to [a] \]

Returns suffix without first \( n \) elements.

\[ \text{splitAt} :: \text{Int} \to [a] \to ([a],[a]) \]

\( \text{splitAt} \ n \ \text{xs} \) is equivalent to \( (\text{take} \ n \ \text{xs}, \text{drop} \ n \ \text{xs}) \)

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

Returns longest prefix with elements satisfying a predicate.

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

Returns suffix without takeWhile prefix.

\[ \text{span} :: (a \to \text{Bool}) \to [a] \to ([a],[a]) \]

\( \text{span} \ p \ \text{xs} \) is equivalent to \( (\text{takeWhile} \ p \ \text{xs}, \text{dropWhile} \ p \ \text{xs}) \)

\[ \text{break} :: (a \to \text{Bool}) \to [a] \to ([a],[a]) \]

\( \text{break} \ p \ \text{xs} \) is equivalent to \( (\text{takeWhile} \ (\text{not} \ . \ p) \ \text{xs}, \text{dropWhile} \ (\text{not} \ . \ p) \ \text{xs}) \). Thus, it breaks a list at the first occurrence of an element satisfying \( p \).

\[ \text{reverse} :: [a] \to [a] \]

Reverses the order of all elements in a list.

\[ \text{and} :: [\text{Bool}] \to \text{Bool} \]

Computes the conjunction of a Boolean list.

\[ \text{or} :: [\text{Bool}] \to \text{Bool} \]
Computes the disjunction of a Boolean list.

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

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

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

Is a given predicate satisfied by all elements in a list?

\textbf{elem} :: \text{Eq }a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}

Element of a list?

\textbf{notElem} :: \text{Eq }a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}

Not element of a list?

\textbf{lookup} :: \text{Eq }b \Rightarrow b \rightarrow [(b,a)] \rightarrow \text{Maybe }a

Looks up a key in an association list.

\textbf{maybe} :: b \rightarrow (a \rightarrow b) \rightarrow \text{Maybe }a \rightarrow b

\textbf{either} :: (b \rightarrow a) \rightarrow (c \rightarrow a) \rightarrow \text{Either }b\ c\rightarrow a

\textbf{getChar} :: \text{IO }\text{Char}

An action that reads a character from standard output and returns it.

\textbf{getLine} :: \text{IO }\text{String}

An action that reads a line from standard input and returns it.

\textbf{putChar} :: \text{Char} \rightarrow \text{IO }()\n
An action that puts its character argument on standard output.

\textbf{putStr} :: \text{String} \rightarrow \text{IO }()

Action to print a string on standard output.

\textbf{putStrLn} :: \text{String} \rightarrow \text{IO }()

Action to print a string with a newline on standard output.

\textbf{print} :: \text{Show }a \Rightarrow a \rightarrow \text{IO }()

Converts a term into a string and print it.

\textbf{readFile} :: \text{String} \rightarrow \text{IO }\text{String}
An action that (lazily) reads a file and returns its contents.

\[ \text{writeFile} :: \text{String} \to \text{String} \to \text{IO ()} \]

An action that writes a file.

\[ \text{appendFile} :: \text{String} \to \text{String} \to \text{IO ()} \]

An action that appends a string to a file. It behaves like \text{writeFile} if the file does not exist.

\[ \text{userError} :: \text{String} \to \text{IOError} \]

A user error value is created by providing a description of the error situation as a string.

\[ \text{ioError} :: \text{IOError} \to \text{IO a} \]

Raises an I/O exception with a given error value.

\[ \text{catch} :: \text{IO a} \to (\text{IOError} \to \text{IO a}) \to \text{IO a} \]

Catches a possible error or failure during the execution of an I/O action. \text{catch act errfun} executes the I/O action \text{act}. If an exception or failure occurs during this I/O action, the function \text{errfun} is applied to the error value.

\[ \text{success} :: \text{Bool} \]

The always satisfiable constraint.

\[ \text{solve} :: \text{Bool} \to \text{Bool} \]

Enforce a Boolean condition to be true. The computation fails if the argument evaluates to \text{False}.

\[ \text{doSolve} :: \text{Bool} \to \text{IO ()} \]

Solves a constraint as an I/O action. Note: The constraint should be always solvable in a deterministic way.

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

The equational constraint. \((e1 =:= e2)\) is satisfiable if both sides \(e1\) and \(e2\) can be reduced to a unifiable data term (i.e., a term without defined function symbols).

\[ (=:<=) :: \text{a} \to \text{a} \to \text{Bool} \]

Non-strict equational constraint. Used to implement functional patterns.

\[ (=:<=<) :: \text{Data a} \Rightarrow \text{a} \to \text{a} \to \text{Bool} \]

Non-strict equational constraint for linear functional patterns. Thus, it must be ensured that the first argument is always (after evaluation by narrowing) a linear pattern. Experimental.
(\&) :: Bool \to Bool \to Bool

Concurrent conjunction. An expression like \((c_1 \& c_2)\) is evaluated by evaluating the \(c_1\) and \(c_2\) in a concurrent manner.

(\&>) :: Bool \to a \to a

Conditional expression. An expression like \((c \&> e)\) is evaluated by evaluating the first argument to \(\text{True}\) and then evaluating \(e\). The expression has no value if the condition does not evaluate to \(\text{True}\).

(?) :: a \to a \to a

Non-deterministic choice \textit{par excellence}. The value of \(x ? y\) is either \(x\) or \(y\).

anyOf :: [a] \to a

Returns non-deterministically any element of a list.

unknown :: Data a \Rightarrow a

Evaluates to a fresh free variable.

failed :: a

A non-reducible polymorphic function. It is useful to express a failure in a search branch of the execution.

error :: String \to a

Aborts the execution with an error message.

apply :: (a \to b) \to a \to b

cond :: Bool \to a \to a

letrec :: a \to a \to Bool

failure :: a \to b \to c

(===) :: Data a \Rightarrow a \to a \to Bool

aValue :: Data a \Rightarrow () \to a
(==) :: Eq a ⇒ a → a → Bool

(/=) :: Eq a ⇒ a → a → Bool

compare :: Ord a ⇒ a → a → Ordering

(<) :: Ord a ⇒ a → a → Bool

(>) :: Ord a ⇒ a → a → Bool

(<=) :: Ord a ⇒ a → a → Bool

(>=) :: Ord a ⇒ a → a → Bool

min :: Ord a ⇒ a → a → a

max :: Ord a ⇒ a → a → a

show :: Show a ⇒ a → String

showsPrec :: Show a ⇒ Int → a → String → String

showList :: Show a ⇒ [a] → String → String

readsPrec :: Read a ⇒ Int → String → [(a,String)]

readList :: Read a ⇒ String → [[(a, String)]]
minBound :: Bounded a ⇒ () → a

maxBound :: Bounded a ⇒ () → a

succ :: Enum a ⇒ a → a

pred :: Enum a ⇒ a → a

toEnum :: Enum a ⇒ Int → a

fromEnum :: Enum a ⇒ a → Int

toEnum :: Enum a ⇒ Int → a

enumFrom :: Enum a ⇒ a → [a]

enumFromThen :: Enum a ⇒ a → a → [a]

enumFromTo :: Enum a ⇒ a → a → [a]

enumFromThenTo :: Enum a ⇒ a → a → [a]

(+) :: Num a ⇒ a → a → a

(−) :: Num a ⇒ a → a → a

(*) :: Num a ⇒ a → a → a

negate :: Num a ⇒ a → a
abs :: Num a ⇒ a → a

signum :: Num a ⇒ a → a

fromInt :: Num a ⇒ Int → a

(/) :: Fractional a ⇒ a → a → a

recip :: Fractional a ⇒ a → a

fromFloat :: Fractional a ⇒ Float → a

toFloat :: Real a ⇒ a → Float

div :: Integral a ⇒ a → a → a

mod :: Integral a ⇒ a → a → a

quot :: Integral a ⇒ a → a → a

rem :: Integral a ⇒ a → a → a

divMod :: Integral a ⇒ a → a → (a,a)

quotRem :: Integral a ⇒ a → a → (a,a)

toInt :: Integral a ⇒ a → Int
properFraction :: RealFrac a ⇒ forall b.Integral b ⇒ a → (b,a)

truncate :: RealFrac a ⇒ forall b.Integral b ⇒ a → b

round :: RealFrac a ⇒ forall b.Integral b ⇒ a → b

ceiling :: RealFrac a ⇒ forall b.Integral b ⇒ a → b

floor :: RealFrac a ⇒ forall b.Integral b ⇒ a → b

pi :: Floating a ⇒ () → a

exp :: Floating a ⇒ a → a

log :: Floating a ⇒ a → a

sqrt :: Floating a ⇒ a → a

(**) :: Floating a ⇒ a → a → a

logBase :: Floating a ⇒ a → a → a

sin :: Floating a ⇒ a → a

cos :: Floating a ⇒ a → a

tan :: Floating a ⇒ a → a
asin :: Floating a ⇒ a → a
acos :: Floating a ⇒ a → a
atan :: Floating a ⇒ a → a
sinh :: Floating a ⇒ a → a
cosh :: Floating a ⇒ a → a
tanh :: Floating a ⇒ a → a
asinh :: Floating a ⇒ a → a
acosh :: Floating a ⇒ a → a
atanh :: Floating a ⇒ a → a
mempty :: Monoid a ⇒ () → a
mappend :: Monoid a ⇒ a → a → a
mconcat :: Monoid a ⇒ [a] → a
fmap :: Functor a ⇒ forall b c.(c → b) → Apply a c → Apply a b
(<$) :: Functor a ⇒ forall b c.c → Apply a b → Apply a c
pure :: Applicative a ⇒ forall b. b → Apply a b

(<*>): Applicative a ⇒ forall b c. Apply a (c → b) → Apply a c → Apply a b

(*>): Applicative a ⇒ forall b c. Apply a c → Apply a b → Apply a b

(<*): Applicative a ⇒ forall b c. Apply a c → Apply a b → Apply a c

liftA2 :: Applicative a ⇒ forall b c d. (d → c → b) → Apply a d → Apply a c → Apply a b

empty :: Alternative a ⇒ forall b. () → Apply a b

(<|>): Alternative a ⇒ forall b. Apply a b → Apply a b → Apply a b

some :: Alternative a ⇒ forall b. Apply a b → Apply a [b]

many :: Alternative a ⇒ forall b. Apply a b → Apply a [b]

(>>=) :: Monad a ⇒ forall b c. Apply a c → (c → Apply a b) → Apply a b

(>>) :: Monad a ⇒ forall b c. Apply a c → Apply a b → Apply a b

return :: Monad a ⇒ forall b. b → Apply a b

fail :: MonadFail a ⇒ forall b. String → Apply a b

This Module is a modified version of the Module System.Console.GetOpt by Sven Panne from the ghc-base package it has been adapted for Curry by Bjoern Peemoeller
(c) Sven Panne 2002-2005 The Glasgow Haskell Compiler License
Copyright 2004, The University Court of the University of Glasgow. All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted
provided that the following conditions are met:
this list of conditions and the following disclaimer.
this list of conditions and the following disclaimer in the documentation and/or other materials
provided with the distribution.
used to endorse or promote products derived from this software without specific prior written
permission.

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

Exported types:

data ArgOrder

  
  \textit{Exported constructors:}
  
  \begin{itemize}
  \item RequireOrder :: ArgOrder a
  \item Permute :: ArgOrder a
  \item ReturnInOrder :: (String $\to$ a) $\to$ ArgOrder a
  \end{itemize}

data OptDescr

  \textit{Exported constructors:}
  
  \begin{itemize}
  \item Option :: String $\to$ [String] $\to$ (ArgDescr a) $\to$ String $\to$ OptDescr a
  \end{itemize}

data ArgDescr
Exported constructors:

- NoArg :: a → ArgDescr a
- ReqArg :: (String → a) → String → ArgDescr a
- OptArg :: (Maybe String → a) → String → ArgDescr a

Exported functions:

usageInfo :: String → [OptDescr a] → String

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

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

A.2.15 Library System.CPUTime

Exported functions:

gCPUTime :: IO Int
Returns the current cpu time of the process in milliseconds.

gElapsedTime :: IO Int
Returns the current elapsed time of the process in milliseconds. This operation is not supported in KiCS2 (there it always returns 0), but only included for compatibility reasons.

A.2.16 Library System.Environment

Library to access parts of the system environment.

Exported functions:

gArgs :: IO [String]
Returns the list of the program’s command line arguments. The program name is not included.

gEnv :: String → IO String
Returns the value of an environment variable. The empty string is returned for undefined environment variables.
setEnv :: String → String → IO ()

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

unsetEnv :: String → IO ()

Removes an environment variable that has been set by `codesetEnv/code`.

getHostname :: IO String

Returns the hostname of the machine running this process.

getProgName :: IO String

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

isPosix :: Bool

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

isWindows :: Bool

Is the underlying operating system a Windows system?

A.2.17 Library System.IO

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

Exported types:

data Handle

  `Exported constructors:`

data IOMode

  The modes for opening a file.

  `Exported constructors:`

  • ReadMode :: IOMode

  • WriteMode :: IOMode

  • AppendMode :: IOMode

data SeekMode
The modes for positioning with \texttt{hSeek} in a file.

\textit{Exported constructors:}

- \texttt{AbsoluteSeek :: SeekMode}
- \texttt{RelativeSeek :: SeekMode}
- \texttt{SeekFromEnd :: SeekMode}

\textit{Exported functions:}

\texttt{stdin :: Handle}

Standard input stream.

\texttt{stdout :: Handle}

Standard output stream.

\texttt{stderr :: Handle}

Standard error stream.

\texttt{openFile :: String \to IOMode \to IO Handle}

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

\texttt{hClose :: Handle \to IO ()}

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

\texttt{hFlush :: Handle \to IO ()}

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

\texttt{hIsEOF :: Handle \to IO Bool}

Is handle at end of file?

\texttt{isEOF :: IO Bool}

Is standard input at end of file?

\texttt{hSeek :: Handle \to SeekMode \to Int \to IO ()}

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

\texttt{hWaitForInput :: Handle \to Int \to 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 is treated as a line terminator and the (partial) line is returned.

hGetContents :: Handle → IO String

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

getContents :: IO String

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

hPutChar :: Handle → Char → IO ()

Puts a character to an output handle.
hPutStr :: Handle → String → IO ()
   Puts a string to an output handle.

hPutStrLn :: Handle → String → IO ()
   Puts a string with a newline to an output handle.

hPrint :: Show a ⇒ Handle → a → IO ()
   Converts a term into a string and puts it to an output handle.

hIsReadable :: Handle → IO Bool
   Is the handle readable?

hIsWritable :: Handle → IO Bool
   Is the handle writable?

hIsTerminalDevice :: Handle → IO Bool
   Is the handle connected to a terminal?

A.2.18 Library System.IO.Unsafe

Library containing unsafe operations. These operations should be carefully used (e.g., for testing or debugging). These operations should not be used in application programs!

Exported functions:

unsafePerformIO :: IO a → a
   Performs and hides an I/O action in a computation (use with care!).

trace :: String → a → a
   Prints the first argument as a side effect and behaves as identity on the second argument.

spawnConstraint :: Bool → a → a
   Spawns a constraint and returns the second argument. This function can be considered as defined by spawnConstraint c x | c = x. However, the evaluation of the constraint and the right-hand side are performed concurrently, i.e., a suspension of the constraint does not imply a blocking of the right-hand side and the right-hand side might be evaluated before the constraint is successfully solved. Thus, a computation might return a result even if some of the spawned constraints are suspended (use the PAKCS option +suspend to show such suspended goals).

isVar :: a → Bool
   Tests whether the first argument evaluates to a currently unbound variable (use with care!).

133
identicalVar :: a → a → Bool

Tests whether both arguments evaluate to the identical currently unbound variable (use with care!). For instance, identicalVar (id x) (fst (x,1)) evaluates to True whereas identicalVar x y and let x=1 in identicalVar x x evaluate to False

isGround :: a → Bool

Tests whether the argument evaluates to a ground value (use with care!).

cmpareAnyTerm :: a → a → Ordering

Comparison of any data terms, possibly containing variables. Data constructors are compared in the order of their definition in the datatype declarations and recursively in the arguments. Variables are compared in some internal order.

showAnyTerm :: a → String

Transforms the normal form of a term into a string representation in standard prefix notation. Thus, showAnyTerm evaluates its argument to normal form. This function is similar to the function ReadShowTerm.showTerm but it also transforms logic variables into a string representation that can be read back by Unsafe.read(s)AnyUnqualifiedTerm. Thus, the result depends on the evaluation and binding status of logic variables so that it should be used with care!

showAnyQTerm :: a → String

Transforms the normal form of a term into a string representation in standard prefix notation. Thus, showAnyQTerm evaluates its argument to normal form. This function is similar to the function ReadShowTerm.showQTerm but it also transforms logic variables into a string representation that can be read back by Unsafe.read(s)AnyQTerm. Thus, the result depends on the evaluation and binding status of logic variables so that it should be used with care!

readsAnyUnqualifiedTerm :: [String] → String → [(a,String)]

Transform a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The string might contain logical variable encodings produced by showAnyTerm. In case of a successful parse, the result is a one element list containing a pair of the data term and the remaining unparsed string.

readAnyUnqualifiedTerm :: [String] → String → a

Transforms a string containing a term in standard prefix notation without module qualifiers into the corresponding data term. The string might contain logical variable encodings produced by showAnyTerm.

readsAnyQTerm :: String → [(a,String)]
Transforms a string containing a term in standard prefix notation with qualified constructor names into the corresponding data term. The string might contain logical variable encodings produced by \( \text{showAnyQTerm} \). 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.

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

Transforms a string containing a term in standard prefix notation with qualified constructor names into the corresponding data term. The string might contain logical variable encodings produced by \( \text{showAnyQTerm} \).

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

Transforms any expression (even not in normal form) into a string representation in standard prefix notation without module qualifiers. The result depends on the evaluation and binding status of logic variables so that it should be used with care!

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

Transforms any expression (even not in normal form) into a string representation in standard prefix notation with module qualifiers. The result depends on the evaluation and binding status of logic variables so that it should be used with care!

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

Transforms a string containing an expression in standard prefix notation with qualified constructor names into the corresponding expression. The string might contain logical variable and defined function encodings produced by \( \text{showAnyQExpression} \). In case of a successful parse, the result is a one element list containing a pair of the expression and the remaining unparsed string.

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

Transforms a string containing an expression in standard prefix notation with qualified constructor names into the corresponding expression. The string might contain logical variable and defined function encodings produced by \( \text{showAnyQExpression} \).

A.2.19 Library Test.Prop

This module defines the interface of properties that can be checked with the CurryCheck tool, an automatic property-based test tool based on the EasyCheck library. The ideas behind EasyCheck are described in this paper. CurryCheck automatically tests properties defined with this library. CurryCheck supports the definition of unit tests (also for I/O operations) and property tests parameterized over some arguments. CurryCheck is described in more detail in this paper. Basically, this module is a stub clone of the EasyCheck library which contains only the interface of the operations used to specify properties. Hence, this library does not import any other library. This supports the definition of properties in any other module (except for the prelude).
Exported functions:

returns :: Eq a ⇒ Show a ⇒ IO a → a → PropIO

The property returns a x is satisfied if the execution of the I/O action a returns the value x.

dameReturns :: Eq a ⇒ Show a ⇒ IO a → IO a → PropIO

The property sameReturns a1 a2 is satisfied if the execution of the I/O actions a1 and a2 return identical values.

toError :: a → PropIO

The property toError a is satisfied if the evaluation of the argument to normal form yields an exception.

toI0Error :: IO a → PropIO

The property toI0Error a is satisfied if the execution of the I/O action a causes an exception.

(=-=-) :: Eq a ⇒ Show a ⇒ a → a → Prop

The property x -=- y is satisfied if x and y have deterministic values that are equal.

(<~>) :: Eq a ⇒ Show a ⇒ a → a → Prop

The property x <~&gt; y is satisfied if the sets of the values of x and y are equal.

(~>) :: Eq a ⇒ Show a ⇒ a → a → Prop

The property x ~> y is satisfied if x evaluates to every value of y. Thus, the set of values of y must be a subset of the set of values of x.

(<~) :: Eq a ⇒ Show a ⇒ a → a → Prop

The property x <~ y is satisfied if y evaluates to every value of x. Thus, the set of values of x must be a subset of the set of values of y.

(<~>) :: Eq a ⇒ Show a ⇒ a → a → Prop

The property x <~&gt; y is satisfied if the multisets of the values of x and y are equal.

(==>) :: Bool → Prop → Prop

A conditional property is tested if the condition evaluates to True.

solutionOf :: Data a ⇒ (a → Bool) → a

solutionOf p returns (non-deterministically) a solution of predicate p. This operation is useful to test solutions of predicates.
is :: Show a ⇒ a → (a → Bool) → Prop

The property is x p is satisfied if x has a deterministic value which satisfies p.

isAlways :: Show a ⇒ a → (a → Bool) → Prop

The property isAlways x p is satisfied if all values of x satisfy p.

isEventually :: Show a ⇒ a → (a → Bool) → Prop

The property isEventually x p is satisfied if some value of x satisfies p.

uniquely :: Bool → Prop

The property uniquely x is satisfied if x has a deterministic value which is true.

always :: Bool → Prop

The property always x is satisfied if all values of x are true.

eventually :: Bool → Prop

The property eventually x is satisfied if some value of x is true.

failing :: Show a ⇒ a → Prop

The property failing x is satisfied if x has no value.

successful :: Show a ⇒ a → Prop

The property successful x is satisfied if x has at least one value.

deterministic :: Show a ⇒ a → Prop

The property deterministic x is satisfied if x has exactly one value.

(#) :: Eq a ⇒ Show a ⇒ a → Int → Prop

The property x # n is satisfied if x has n values.

(#<) :: Eq a ⇒ Show a ⇒ a → Int → Prop

The property x #< n is satisfied if x has less than n values.

(#>) :: Eq a ⇒ Show a ⇒ a → Int → Prop

The property x #> n is satisfied if x has more than n values.

for :: Show a ⇒ a → (a → Prop) → Prop

The property for x p is satisfied if all values y of x satisfy property p y.

forAll :: Show a ⇒ [a] → (a → Prop) → Prop

The property forAll xs p is satisfied if all values x of the list xs satisfy property p x.
The property \( f \Leftrightarrow g \) is satisfied if \( f \) and \( g \) are equivalent operations, i.e., they can be replaced in any context without changing the computed results.

**label :: String \to Prop \to Prop**

Assign a label to a property. All labeled tests are counted and shown at the end.

**classify :: Bool \to String \to Prop \to Prop**

Assign a label to a property if the first argument is \texttt{True}. All labeled tests are counted and shown at the end. Hence, this combinator can be used to classify tests:

\[
multIsComm x y = classify (x<0 \lor y<0) \text{"Negative"} \$ x*y \ldots y*x
\]

**trivial :: Bool \to Prop \to Prop**

Assign the label "trivial" to a property if the first argument is \texttt{True}. All labeled tests are counted and shown at the end.

**collect :: Show a \Rightarrow a \to Prop \to Prop**

Assign a label showing the given argument to a property. All labeled tests are counted and shown at the end.

**collectAs :: Show a \Rightarrow String \to a \to Prop \to Prop**

Assign a label showing a given name and the given argument to a property. All labeled tests are counted and shown at the end.

**valuesOf :: a \to [a]**

Computes the list of all values of the given argument according to a given strategy (here: randomized diagonalization of levels with flattening).

### A.2.20 Library Test.Prop.Types

This module defines some types used by the EasyCheck libraries.

**Exported types:**

**data PropIO**

Abstract type to represent properties involving IO actions.

**Exported constructors:**

- **PropIO :: (Bool \to String \to IO (Maybe String)) \to PropIO**

**data Prop**
Abstract type to represent standard properties to be checked. Basically, it contains all tests to be executed to check the property.

**Exported constructors:**

- Prop :: [Test] → Prop

**data Test**

Abstract type to represent a single test for a property to be checked. A test consists of the result computed for this test, the arguments used for this test, and the labels possibly assigned to this test by annotating properties.

**Exported constructors:**

- Test :: Result → [String] → [String] → Test

**data Result**

Data type to represent the result of checking a property.

**Exported constructors:**

- Undef :: Result
- Ok :: Result
- Falsified :: [String] → Result
- Ambigious :: [Bool] → [String] → Result

**A.2.21 Library Text.Show**

This library provides a type and combinators for show functions using functional lists.

**Exported types:**

\[
type \text{ShowS} = \text{String} \rightarrow \text{String}
\]

**Exported functions:**

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

Prepend a string

\[
\text{showChar} :: \text{Char} \rightarrow \text{String} \rightarrow \text{String}
\]

Prepend a single character

\[
\text{showParen} :: \text{Bool} \rightarrow (\text{String} \rightarrow \text{String}) \rightarrow \text{String} \rightarrow \text{String}
\]
Surround the inner show function with parentheses if the first argument evaluates to True.

\[
\text{shows} :: \text{Show } a \Rightarrow a \rightarrow \text{String} \rightarrow \text{String}
\]

Convert a value to ShowS using the standard show function.
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:

```markdown
_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
```
B.2 Lists and Block Formatting

An unordered list (i.e., without numbering) is introduced by putting a star in front of the list elements (where the star can be preceded by blanks). The individual list elements must contain the same indentation, as in

* First list element
  with two lines

* Next list element.

  It contains two paragraphs.

* Final list element.

This is formatted as follows:

- First list element with two lines
- Next list element.
  It contains two paragraphs.
- Final list element.

Instead of a star, one can also put dashes or plus to mark unordered list items. Furthermore, one could nest lists. Thus, the input text

- Color:
  + Yellow
  + Read
  + Blue
- BW:
  + Black
  + White

is formatted as

- Color:
Similarly, ordered lists (i.e., with numbering each item) are introduced by a number followed by a dot and at least one blank. All following lines belonging to the same numbered item must have the same indent as the first line. The actual value of the number is not important. Thus, the input

1. First element

99. Second
   element

is formatted as

1. First element

2. Second element

A quotation block is marked by putting a right angle followed by a blank in front of each line:

> This is
> a quotation.

It will be formatted as a quote element:

This is a quotation.

A block containing program code starts with a blank line and is marked by intending each input line by at least four spaces where all following lines must have at least the same indentation as the first non-blank character of the first line:

```haskell
f x y = let z = (x,y)
       in (z,z)
```

The indentation is removed in the output:

```haskell
f x y = let z = (x,y)
       in (z,z)
```

To visualize the structure of a document, one can also put a line containing only blanks and at least three dashes (stars would also work) in the source text:

```
--------------------------------------------------
```

This is formatted as a horizontal line:
B.3 Headers

There are two forms to mark headers. In the first form, one can "underline" the main header in the source text by equal signs and the second-level header by dashes:

First-level header
==================

Second-level header
-------------------

Alternatively (and for more levels), one can prefix the header line by up to six hash characters, where the number of characters corresponds to the header level (where level 1 is the main header):

# Main header

## Level 2 header

### Level 3

#### Level 4

##### Level 5

###### Level 6

_________________________________
C  SQL Syntax Supported by CurryPP

This section contains a grammar in EBNF which specifies the SQL syntax recognized by the Curry preprocessor in integrated SQL code (see Sect. 12.4). The grammar satisfies the LL(1) property and is influenced by the SQLite dialect.\(^{14}\)

--- type of statements -----------------------------------

\[
\text{statement ::= queryStatement | transactionStatement} \\
\text{queryStatement ::= ( deleteStatement \\
| insertStatement \\
| selectStatement \\
| updateStatement )} \\
','
\]

--- transaction -------------------------------------

\[
\text{transactionStatement ::= (BEGIN \\
| IN TRANSACTION '(' queryStatement \\
| queryStatement { queryStatement }')' \\
| COMMIT \\
| ROLLBACK ) ';'}
\]

--- delete ------------------------------------------

\[
\text{deleteStatement ::= DELETE FROM tableSpecification} \\
[ WHERE condition ]
\]

--- insert ------------------------------------------

\[
\text{insertStatement ::= INSERT INTO tableSpecification} \\
\text{insertSpecification} \\
\text{insertSpecification ::= [ '(' columnNameList ')' ] valuesClause} \\
\text{valuesClause ::= VALUES valueList}
\]

--- update ----------------------------------------

\[
\text{updateStatement ::= UPDATE tableSpecification} \\
\text{SET columnAssignment {',' columnAssignment} \\
[ WHERE condition ] \\
| embeddedCurryExpression }) \\
\text{columnAssignment ::= columnName '=' literal}
\]

--- select statement ---------------------------------

\(^{14}\)https://sqlite.org/lang.html
selectStatement ::= selectHead { setOperator selectHead }  
             [ orderByClause ]  
             [ limitClause ]  
selectHead ::= selectClause fromClause  
             [ WHERE condition ]  
             [ groupByClause [ havingClause ]]  
setOperator ::= UNION | INTERSECT | EXCEPT  
selectClause ::= SELECT [( DISTINCT | ALL )]  
             ( selectElementList | '*' )  
selectElementList ::= selectElement { ',' selectElement }  
selectElement ::= [ tableIdentifier'.' ] columnName  
             | aggregation  
             | caseExpression  
aggregation ::= function '(' [ DISTINCT ] columnReference ')'  
caseExpression ::= CASE WHEN condition THEN operand  
                 ELSE operand END  
function ::= COUNT | MIN | MAX | AVG | SUM  
fromClause ::= FROM tableReference { ',' tableReference }  
groupByClause ::= GROUP BY columnList  
havingClause ::= HAVING conditionWithAggregation  
orderByClause ::= ORDER BY columnReference [ sortDirection ]  
             { ',' columnReference  
              [ sortDirection ] }  
sortDirection ::= ASC | DESC  
limitClause = LIMIT integerExpression  
-------------common elements-----------------------------  
columnList ::= columnReference { ',' columnReference }  
columnReference ::= [ tableIdentifier'.' ] columnName  
columnNameList ::= columnName { ',' columnName}  
tableReference ::= tableSpecification [ AS tablePseudonym ]
tableSpecification ::= tableName

condition ::= operand operatorExpression

| [logicalOperator condition]
| EXISTS subquery [logicalOperator condition]
| NOT condition
| '(' condition ')'
| satConstraint [logicalOperator condition]

operand ::= columnReference
| literal

subquery ::= '(' selectStatement ')

operatorExpression ::= IS NULL
| NOT NULL
| binaryOperator operand
| IN setSpecification
| BETWEEN operand operand
| LIKE quotes pattern quotes

setSpecification ::= literalList

binaryOperator ::= '>' | '<' | '>= ' | '<= ' | '=' | '!='

logicalOperator ::= AND | OR

conditionWithAggregation ::= aggregation [logicalOperator disaggregation]
| '(' conditionWithAggregation ')
| operand operatorExpression

| [logicalOperator conditionWithAggregation]
| NOT conditionWithAggregation
| EXISTS subquery
| [logicalOperator conditionWithAggregation]
| satConstraint

aggregation ::= function '('ALL | DISTINCT) columnReference')
| binaryOperator
| operand

satConstraint ::= SATISFIES tablePseudonym
| relation
| tablePseudonym

joinSpecification ::= joinType tableSpecification
\[ \text{AS tablePseudonym} \]
\[ \text{joinCondition} \]
\[ \text{joinSpecification} \]

joinType ::= CROSS JOIN | INNER JOIN

joinCondition ::= ON condition

-------------identifier and datatypes-------------------------

valueList ::= ( embeddedCurryExpression | literalList )
\{',', ( embeddedCurryExpression | literalList )\}

literalList ::= '\(' literal \{',' literal \} '\')'

literal ::= numericalLiteral
| quotes alphaNumericalLiteral quotes
| dateLiteral
| booleanLiteral
| embeddedCurryExpression
| NULL

numericalLiteral ::= integerExpression
| floatExpression

integerExpression ::= \[ [ - ] digit \{ digit \}\]

floatExpression ::= \[ [ - ] digit \{ digit \} \'.', digit \{ digit \}\]

alphaNumericalLiteral ::= character \{ character \}
character ::= digit | letter

dateLiteral ::= year ':.' month ':.' day ':.'
| hours ':.' minutes ':.' seconds

month ::= digit digit

day ::= digit digit

hours ::= digit digit

minutes ::= digit digit

seconds ::= digit digit

year ::= digit digit digit digit

booleanLiteral ::= TRUE | FALSE

embeddedCurryExpression ::= '{' curryExpression '}'

pattern ::= ( character | specialCharacter )
\{( character | specialCharacter )\}

specialCharacter ::= '%' | '_'
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

letter ::= (a...z) | (A...Z)

tableIdentifier ::= tablePseudonym | tableName
columnName ::= letter [alphanumericalLiteral]
tableName ::= letter [alphanumericalLiteral]
tablePseudonym ::= letter
relation ::= letter [[alphanumericalLiteral] | '_']
quotes ::= ("" | ''')
D  Overview of the PAKCS Distribution

A schematic overview of the various components contained in the distribution of PAKCS and the translation process of programs inside PAKCS is shown in Figure 7 on page 151. In this figure, boxes denote different components of PAKCS and names in boldface denote files containing various intermediate representations during the translation process (see Section E below). The PAKCS distribution contains a front end for reading (parsing and type checking) Curry programs that can be also used by other Curry implementations. The back end (formerly known as “Curry2Prolog”) compiles Curry programs into Prolog programs. It also supports packages with constraint solvers for arithmetic constraints over real numbers and finite domain constraints, and further libraries for GUI programming, meta-programming etc. It does not implement encapsulated search in full generality (only a strict version of findall is supported by the library Control.Findall which is part of the package searchtree), and concurrent threads are not executed in a fair manner.
Figure 7: Overview of PAKCS
E Auxiliary Files

During the translation and execution of a Curry program with PAKCS, various intermediate representations of the source program are created and stored in different files which are shortly explained in this section. If you use PAKCS, 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 1.1) to clean up your directories.

The various components of PAKCS 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.1.4). This representation might be useful for other back ends and compilers for Curry and is the basis doing metaprogramming in Curry. This file is implicitly generated when a program is compiled with PAKCS. It can be also explicitly generated by the front end of PAKCS:

```
pakcs frontend --flat -ipakcs.home/lib prog
```

The FlatCurry representation of a Curry program is usually generated by the front-end after parsing, type checking and eliminating local declarations.

If the Curry module $M$ is stored in the directory $dir$, the corresponding FlatCurry program is stored in the directory “$dir/.curry”’. This is also the case for hierarchical module names: if the module $D1.D2.M$ is stored in the directory $dir$ (i.e., the module is actually stored in $dir/D1/D2/M.curry$), then the corresponding FlatCurry program is stored in “$dir/.curry/D1/D2/M.fcy$”.

**prog.fint**: This file contains the interface of the program in the so-called “FlatCurry” representation, i.e., it is similar to **prog.fcy** but contains only exported entities and the bodies of all functions omitted (i.e., “external”). This representation is useful for providing a fast access to module interfaces. This file is implicitly generated when a program is compiled with PAKCS and stored in the same directory as **prog.fcy**.

**prog.pl**: This file contains a Prolog program as the result of translating the Curry program with PAKCS.

If the Curry module $M$ is stored in the directory $dir$, the corresponding Prolog program is stored in the directory “$dir/.curry/pakcs”’. This is also the case for hierarchical module names: if the module $D1.D2.M$ is stored in the directory $dir$ (i.e., the module is actually stored in $dir/D1/D2/M.curry$), then the corresponding Prolog program is stored in “$dir/.curry/pakcs/D1/D2/ prog.pl$”.

**prog.po**: This file contains the Prolog program **prog.pl** in an intermediate format for faster loading. This file is stored in the same directory as **prog.pl**.

**prog**: This file contains the executable after compiling and saving a program with PAKCS (see Section 2.2).
External Functions

Currently, PAKCS has no general interface to external functions. Therefore, if a new external function should be added to the system, this function must be declared as `external` in the Curry source code and then an implementation for this external function must be inserted in the corresponding back end. An external function is defined as follows in the Curry source code:

1. Add a type declaration for the external function somewhere in the body of the appropriate file (usually, the prelude or some system module).

2. For external functions 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

   ```curry
   f external
   ```

   somewhere in the file containing the type declaration for the external function \( f \).

For instance, the addition on integers can be declared as an external function as follows:

```curry
(+) :: Int → Int → Int
(+) external
```

The further modifications to be done for an inclusion of an external function has to be done in the back end. A new external function is added to the back end of PAKCS by informing the compiler about the existence of an external function and adding an implementation of this function in the run-time system. Therefore, the following items must be added in the PAKCS compiler system:

1. If the Curry module \( \text{Mod} \) contains external functions, there must be a file named \( \text{Mod.pakcs} \) containing the specification of these external functions. The contents of this file is in XML format and has the following general structure:\(^{15}\)

   ```xml
   <primitives>
   
   specification of external function \( f_1 \)
   ...
   
   specification of external function \( f_n \)
   </primitives>
   ```

   The specification of an external function \( f \) with arity \( n \) has the form

   ```xml
   <primitive name="f" arity="n">
   
   <library>lib</library>
   <entry>pred</entry>
   </primitive>
   ```

   where `lib` is the Prolog library (stored in the directory of the Curry module or in the global directory `pakcs/home/src/lib_src`) containing the code implementing this function and `pred` is a predicate name in this library implementing this function. Note that the function \( f \) must be declared in module \( \text{Mod} \): either as an external function or defined in Curry by equations.

\(^{15}\)http://www.informatik.uni-kiel.de/~pakcs/primitives.dtd contains a DTD describing the exact structure of these files.
the latter case, the Curry definition is not translated but calls to this function are redirected to the Prolog code specified above.

Furthermore, the list of specifications can also contain entries of the form

\[ \text{<ignore name="f" arity="n" />} \]

for functions \( f \) with arity \( n \) that are declared in module \( \text{Mod} \) but should be ignored for code generation, e.g., since they are never called w.r.t. to the current implementation of external functions. For instance, this is useful when functions that can be defined in Curry should be (usually more efficiently) implemented as external functions.

Note that the arguments are passed in their current (possibly unevaluated) form. Thus, if the external function requires the arguments to be evaluated in a particular form, this must be done before calling the external function. For instance, the external function for adding two integers requires that both arguments must be evaluated to non-variable head normal form (which is identical to the ground constructor normal form). Therefore, the function \( "+" \) is specified in the prelude by

\[
\begin{align*}
(+) & : : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
    x + y &= (\text{prim\_Int\_plus}$\$ y)$\$ x
\end{align*}
\]

\[
\begin{align*}
\text{prim\_Int\_plus} & : : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{prim\_Int\_plus} & \text{ external}
\end{align*}
\]

where \( \text{prim\_Int\_plus} \) is the actual external function implementing the addition on integers. Consequently, the specification file \( \text{Prelude.pakcs} \) has an entry of the form

\[
\begin{align*}
\text{<primitive name="prim\_Int\_plus" arity="2"}> \\
\text{<library>prim\_standard</library>} \\
\text{<entry>prim\_Int\_plus</entry>} \\
\text{</primitive>}
\end{align*}
\]

where the Prolog library \( \text{prim\_standard.pl} \) contains the Prolog code implementing this function.

2. For most external functions, a standard interface is generated by the compiler so that an \( n \)-ary function can be implemented by an \( (n + 1) \)-ary predicate where the last argument must be instantiated to the result of evaluating the function. The standard interface can be used if all arguments are ensured to be fully evaluated (e.g., see definition of \( (+) \) above) and no suspension control is necessary, i.e., it is ensured that the external function call does not suspend for all arguments. Otherwise, the raw interface (see below) must be used. For instance, the Prolog code implementing \( \text{prim\_Int\_plus} \) contained in the Prolog library \( \text{prim\_standard.pl} \) is as follows (note that the arguments of \( (+) \) are passed in reverse order to \( \text{prim\_Int\_plus} \) in order to ensure a left-to-right evaluation of the original arguments by the calls to \( ($\#$) \)):

\[
\text{prim\_Int\_plus}(Y,X,R) :- R \text{ is } X+Y.
\]

3. The standard interface for I/O actions, i.e., external functions with result type \( \text{IO } a \), assumes that the I/O action is implemented as a predicate (with a possible side effect) that instantiates
the last argument to the returned value of type “a”. For instance, the primitive predicate `prim_getChar` implementing prelude I/O action `getChar` can be implemented by the Prolog code

```prolog
prim_getChar(C) :- get_code(N), char_int(C,N).
```

where `char_int` is a predicate relating the internal Curry representation of a character with its ASCII value.

4. If some arguments passed to the external functions are not fully evaluated or the external function might suspend, the implementation must follow the structure of the PAKCS runtime system by using the raw interface. In this case, the name of the external entry must be suffixed by “[raw]” in the pakcs file. For instance, if we want to use the raw interface for the external function `prim_Int_plus`, the specification file `Prelude.pakcs` must have an entry of the form

```xml
<primitive name="prim_Int_plus" arity="2">
  <library>prim_standard</library>
  <entry>prim_Int_plus[raw]</entry>
</primitive>
```

In the raw interface, the actual implementation of an n-ary external function consists of the definition of an (n+3)-ary predicate `pred`. The first n arguments are the corresponding actual arguments. The (n+1)-th argument is a free variable which must be instantiated to the result of the function call after successful execution. The last two arguments control the suspension behavior of the function (see [5] for more details): The code for the predicate `pred` should only be executed when the (n+2)-th argument is not free, i.e., this predicate has always the SICStus-Prolog block declaration

```prolog
?- block pred(?,...,?,+,-,?).
```

In addition, typical external functions should suspend until the actual arguments are instantiated. This can be ensured by a call to `ensureNotFree` or `$#$` before calling the external function. Finally, the last argument (which is a free variable at call time) must be unified with the (n+2)-th argument after the function call is successfully evaluated (and does not suspend). Additionally, the actual (evaluated) arguments must be dereferenced before they are accessed. Thus, an implementation of the external function for adding integers is as follows in the raw interface:

```prolog
?- block prim_Int_plus(?,...,?,+,-,?).
prim_Int_plus(RY,RX,Result,E0,E) :-
  deref(RX,X), deref(RY,Y), Result is X+Y, E0=E.
```

Here, `deref` is a predefined predicate for dereferencing the actual argument into a constant (and `derefAll` for dereferencing complex structures).

The Prolog code implementing the external functions must be accessible to the run-time system of PAKCS by putting it into the directory containing the corresponding Curry module or into the
system directory `pakcshome/src/lib_src`. Then it will be automatically loaded into the run-time environment of each compiled Curry program.

Note that arbitrary functions implemented in C or Java can be connected to PAKCS by using the corresponding interfaces of the underlying Prolog system.
Index

(), 107
(,), 107
(,,), 107
(,,,), 108
(,,,,), 108
(,,,,,), 108
(,,,,,,), 108
(,,,,,,,), 109
(,,,,,,,,), 109
(,,,,,,,,,), 109
(,,,,,,,,,,), 109
(,,,,,,,,,,,), 109
(...), 109
->, 120
*:, 120
**:, 122
>=:<<, 120
==:, 122
===>, 136
?, 121
@, 17
@author, 44
@cons, 44
@param, 44
@return, 45
@version, 44
[], 111
#, 137
#<, 137
#>., 137
$, 114
$#, 114
$##, 114
&., 121
&&, 116
&>, 121
PAKCS, 8
<?, 122
<*, 127
<*>., 127
:interface, 10
:load, 9
:modules, 10
:peval, 11
:programs, 10
:quit, 10
:reload, 9
:save, 11
:set, 10
:set path, 7
:show, 10
:source, 11
:type, 10
:usedimports, 10
-=, 120
=:=, 120
=<, 120
==<, 120
==, 122
===, 121
==>>, 136
?, 121
@, 17
@author, 44
@cons, 44
@param, 44
@return, 45
@version, 44
[]:, 111
#:?, 137
#:>, 137
$, 114
$#, 114
$##, 114
&., 121
&&, 116
&>, 121
PAKCS, 8
<?, 122
<*, 127
<*>., 127
:interface, 10
:load, 9
:modules, 10
:peval, 11
:programs, 10
:quit, 10
:reload, 9
:save, 11
:set, 10
:set path, 7
:show, 10
:source, 11
:type, 10
:usedimports, 10
-=, 120
=:=, 120
=<, 120
==<, 120
==, 122
===, 121
==>>, 136
?, 121
@, 17
@author, 44
@cons, 44
@param, 44
@return, 45
@version, 44
[]:, 111
#:?, 137
#:>, 137
$, 114
$#, 114
$##, 114
&., 121
&&, 116
&>, 121
PAKCS, 8
<?, 122
<*, 127
<*>., 127
:interface, 10
:load, 9
:modules, 10
:peval, 11
:programs, 10
:quit, 10
:reload, 9
:save, 11
:set, 10
:set path, 7
:show, 10
:source, 11
:type, 10
:usedimports, 10
-=, 120
=:=, 120
=<, 120
==<, 120
==, 122
===, 121
==>>, 136
?, 121
@, 17
@author, 44
@cons, 44
@param, 44
@return, 45
@version, 44
[]:, 111
#:?, 137
#:>, 137
$, 114
$#, 114
$##, 114
&., 121
&&, 116
&>, 121
PAKCS, 8
<?, 122
<*, 127
<*>., 127
abs, 124
AbstractCurry, 94
acos, 126
acosh, 126
all, 119
allfails, 12
always, 137
analyzing programs, 68
and, 118
any, 119
anyOf, 121
appendFile, 120
apply, 121
ArgDescr, 128
ArgOrder, 128
args, 14
as-pattern, 17
asin, 126
asinh, 126
assert, 106
assertIO, 106
asTypeOf, 115
atan, 126
atanh, 126
aValue, 121
baseVersion, 98
Bool, 109
break, 118
CASC, 47
CASS, 68
catch, 120
catMaybes, 105
ceiling, 125
Char, 110
chr, 114
classify, 138
cleancurry, 6
collect, 138
collectAs, 138
comment
documentation, 44
compact, 12
compare, 122
compareAnyTerm, 134
concat, 117
concatMap, 117
cond, 121
connectPort, 93
consfail, 12
cost, 115
construct, 38
cos, 125
cosh, 126
curry, 8, 115
curry erd2curry, 75
Curry mode, 15
Curry preprocessor, 52
curry-doc, 46
curry-peval, 79
curry-style, 47
curry-verify, 48
Curry2Prolog, 150
CurryCheck, 29
curryCompiler, 97
curryCompilerMajorVersion, 97
curryCompilerMinorVersion, 97
curryCompilerRevisionVersion, 97
CurryDoc, 44
CURRYPATH, 7, 13
function
    external, 153
functional pattern, 16

getArgs, 129
getChar, 119
getContents, 132
getCPUTime, 129
getElapsedTime, 129
getEnv, 129
getHostname, 130
getLine, 119
getOpt, 129
getOpt', 129
getProgName, 130
groundNormalForm, 115
group, 102
groupBy, 102

Handle, 130
hClose, 131
head, 116
hFlush, 131
hGetChar, 132
hGetContents, 132
hGetLine, 132
hIsEOF, 131
hIsReadable, 133
hIsTerminalDevice, 133
hIsWritable, 133
hPrint, 133
hPutChar, 132
hPutStr, 133
hPutStrLn, 133
hReady, 132
hSeek, 131
hWaitForInput, 131
hWaitForInputOrMsg, 132
hWaitForInputs, 132
hWaitForInputsOrMsg, 132

id, 115
identicalVar, 134
Identity, 100
ifThenElse, 116
init, 103
inits, 103
insertBy, 103
installDir, 98
Int, 111
interactive, 12
intercalate, 102
intersect, 101
intersectBy, 102
intersperse, 102
intToDigit, 98
IO, 110
IOError, 110
ioError, 120
IOMode, 130
IORef, 100
is, 137
isAlpha, 113
isAlphaNum, 113
isAlways, 137
isAscii, 98
isAsciiLower, 98
isAsciiUpper, 98
isBinDigit, 113
isControl, 98
isDigit, 113
isEOF, 131
isEventually, 137
isGround, 134
isHexDigit, 113
isInfixOf, 103
isJust, 105
isLatin1, 98
isLeft, 99
isLower, 113
isNothing, 105
isOctDigit, 113
isPosix, 130
isPrefixOf, 103
isRight, 99
isSpace, 114
isSuffixOf, 103
isUpper, 113
isVar, 133
postcondition, 38
precondition, 38
pred, 123
preprocessor, 52
print, 119
printdepth, 13
printfail, 12
product, 104
profile, 12
program
    analysis, 68
    documentation, 44
    testing, 29
    verification, 48
Prop, 138
properFraction, 125
PropIO, 138
pure, 127
putChar, 119
putStr, 119
putStrLn, 119
quot, 124
quotRem, 124
read, 112
readAnyQExpression, 135
readAnyQTerm, 135
readAnyUnqualifiedTerm, 134
readBin, 107
readCurry, 95
readFile, 119
readFlatCurry, 95
readHex, 106
readInt, 106
readIORef, 100
readList, 122
readNat, 106
readOct, 106
readParen, 112
ReadS, 111
reads, 112
readAnyQExpression, 135
readAnyQTerm, 134
readsAnyUnqualifiedTerm, 134
readsPrec, 122
realToFrac, 112
recip, 124
rem, 124
repeat, 118
replace, 103
replicate, 118
replicateM, 97
replicateM_, 97
Result, 139
return, 127
returns, 136
reverse, 118
rights, 99
round, 125
runcurry, 65
runIdentity, 100
safe, 13
sameReturns, 136
scanl, 104
scanl1, 104
scanr, 104
scanr1, 104
SeekMode, 130
send, 93
seq, 114
sequence, 113
sequence_, 113
sequenceA, 96
sequenceA_, 96
set functions, 7
setEnv, 130
show, 122
showAnyExpression, 135
showAnyQExpression, 135
showAnyQTerm, 134
showAnyTerm, 134
showChar, 112, 139
showList, 122
showParen, 112, 139
ShowS, 111, 139
shows, 112, 140
showsPrec, 122
showString, 112, 139
signum, 124
sin, 125
single, 14
singleton variables, 6
sinh, 126
snd, 116
solutionOf, 136
solve, 120
some, 127
sort, 103
sortBy, 103
span, 118
spawnConstraint, 133
specification, 38
spiceup, 77
Spicey, 77
split, 102
splitAt, 118
splitOn, 102
spy, 14
sqrt, 125
stderr, 131
stdin, 131
stdout, 131
String, 111
style, 47
style checking, 47
succ, 123
Success, 111
success, 120
successful, 137
sum, 103
tabulator stops, 6
tail, 116
tails, 103
take, 118
takeWhile, 118
tan, 125
tanh, 126
Test, 139
Test.EasyCheck, 29, 33
Test.Prop, 29
testing programs, 29
time, 13
toEnum, 123
toError, 136
toFloat, 124
toInt, 124
toIOError, 136
toLower, 98
toUpper, 98
trace, 14, 105, 133
traceId, 105
traceIO, 106
traceShow, 106
traceShowId, 106
transpose, 102
trivial, 138
truncate, 125
uncurry, 115
unfoldr, 104
union, 101
unionBy, 101
uniquely, 137
unknown, 121
unless, 97
unlines, 114
unsafePerformIO, 133
unsetEnv, 130
until, 115
unwords, 114
unzip, 117
unzip3, 117
usageInfo, 129
userError, 120
v, 13
valuesOf, 138
variables
singleton, 6
verbosity, 13
verify, 48
verifying programs, 48
warn, 13
when, 96
where, 16
words, 114
writeFile, 120
writeIORef, 100

zip, 117
zip3, 117
zipWith, 117
zipWith3, 117
zipWithM, 96
zipWithM_, 96