PAKCS 1.7.3

The Portland Aachen Kiel Curry System

User Manual

Version of September 4, 2006

Michael Hanus[1] [editor]

Additional Contributors:

Sergio Antoy[2]
Bernd Braßel[3]
Martin Engelke[4]
Klaus Höppner[5]
Johannes Koj[6]
Philipp Niederau[7]
Ramin Sadre[8]
Frank Steiner[9]

(1) University of Kiel, Germany, mh@informatik.uni-kiel.de
(2) Portland State University, USA, antoy@cs.pdx.edu
(3) University of Kiel, Germany, bbr@informatik.uni-kiel.de
(4) University of Kiel, Germany, men@informatik.uni-kiel.de
(5) University of Kiel, Germany, klh@informatik.uni-kiel.de
(6) RWTH Aachen, Germany, johannes.koj@sdm.de
(7) RWTH Aachen, Germany, philipp@navigium.de
(8) RWTH Aachen, Germany, ramin@lvs.informatik.rwth-aachen.de
(9) LMU Munich, Germany, fst@bio.informatik.uni-muenchen.de

Contents

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 the high-level implementation of distributed applications, graphical user interfaces, and web services (as described in more detail in [10, 11, 12]).

We assume familiarity with the ideas and features of Curry as described in the Curry language definition [19]. 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 0.8.2).

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 and Ha 2457/5-1.

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`". They are automatically converted into corresponding "`.curry`" files by deleting all lines not starting with "`>`" and removing the prefix "`> `" of the remaining lines.

Since the translation of Curry programs with PAKCS creates some auxiliary files (see Section B 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 on Curry Programs

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

- *Singleton 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 various run-time systems, the definition of functions with local declarations look different from their original definition (in order to see the result of this transformation, you can use the Curry-Browser, see Section 5).
- Tabulator stops instead of blank spaces in source files are interpreted as stops at columns 9, 17, 25, 33, and so on.
- 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`. In contrast to the general definition of encapsulated search [18], 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. It is recommended to use the system module `AllSolutions` for encapsulating search.

- There is currently no general connection to external constraint solvers. However, the Curry2Prolog compiler provides constraint solvers for arithmetic and finite domain constraints (see Sections 2.4.1 and 2.4.2).

1.3 Modules in PAKCS

The current implementation of PAKCS supports only flat module names, i.e., the notation `Dir.Mod.f` is not supported. In order to allow the structuring of modules in different directories, 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 directories "*pakcshome*`/lib`" and "*pakcshome*`/lib/meta`". This search path can be extended by setting the environment variable `CURRYPATH` (or by the "`:set path`" command in Curry2Prolog, see below) to a list of directory names separated by colons ("`:`"). In this case, the directory of the main program is added as the first and the system module directories "*pakcshome*`/lib:`*pakcshome*`/lib/meta`" as the last directories to `CURRYPATH`. Then, each imported module will be searched relative to this search path, i.e., the first occurrence of a module file in the search path is imported. Note that the standard prelude (*pakcshome*`/lib/Prelude.curry`) will be always implicitly imported to all modules.

2 PAKCS/Curry2Prolog: An Interactive Curry Development System

PAKCS/Curry2Prolog, in the following just called "PAKCS", is an interactive system to develop applications written in Curry.[1] 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 How to Use PAKCS

To start PAKCS, execute the command "`pakcs`" (this is a shell script stored in *pakcshome*`/bin` where *pakcshome* is the installation directory of PAKCS). When the system is ready, the prelude (*pakcshome*`/lib/Prelude.curry`) is already loaded, i.e., all definitions in the prelude are accessible. Now you can type in various commands. The **most important commands** 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`. If this file does not exist, the system looks for a FlatCurry file *prog*`.fcy` and compiles from this intermediate representation. If the file *prog*`.fcy` does not exists, too, the system looks for a file *prog*`_flat.xml` containing a FlatCurry program in XML representation (compare command "`:xml`"), translates this into a FlatCurry file *prog*`.fcy` and compiles from this intermediate representation.

`:reload` Repeat the last load command.

expr Evaluate the expression *expr* to normal form and show the computed results. Since the PAKCS compiles Curry programs into Prolog programs, non-deterministic computations are implemented by backtracking. Therefore, computed results are shown one after the other. After each computed result, you can proceed the computation of the next alternative result by typing "`;`" (followed by a CR) or stop the search for alternatives by just typing CR.

Free variables in initial expressions must be declared as in Curry programs (if the free variable mode is not turned on, see option "`+free`" below), i.e., either by a "`let...free in`" or by a "`where...free`" declaration. For instance, one can write

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

or

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

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

Note that lambda abstractions, `let`s and list comprehensions in top-level expressions are not yet supported in initial expressions typed in the top-level of PAKCS.

[1] There are also two other implementations of Curry contained in the PAKCS distribution (Curry2Java and TasteCurry, see Appendix C and D for more details). Since the other implementations are no longer actively supported and Curry2Prolog is the most advanced implementation, we recommend the use of the Curry2Prolog compiler system.

`let` x `=` $expr$ Define the identifier x as an abbreviation for the expression $expr$ which can be used in subsequent expressions. The identifier x is visible until the next `load` or `reload` command.

`:quit` Exit the system.

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

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

`:browse` Start the CurryBrowser to analyze the currently loaded module together with all its imported modules (see Section 5 for more details).

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

`:interface` $prog$ 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 Section 2.4.4).

`:analyze` Analyze the currently loaded program for some properties. Currently, there are the following analysis options:

`functions` Check properties of all functions defined in the currently loaded Curry program (i.e., without the functions defined in the prelude and imported modules). Currently, the following properties are checked:

1. Which functions are defined by overlapping left-hand sides?
2. Which functions are indeterministic, i.e., contains an indirect/implicit call to a `send` constraint on ports (see Section 2.4.3, which includes an implicit committed choice)?

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

`igraph` Visualize the module dependencies of the currently loaded module (without the prelude which is used everywhere) as a graph with the daVinci graph drawing tool (see also the system library `DaVinci`).

`:set` $option$ Set or turn on/off a specific option of the PAKCS environment. 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. The following options 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).

`+/-free` Free variable mode. If the free variable mode is off (default), then free variables occurring in initial expressions entered in the PAKCS environment must always be declared by a "`let...free in`" or "`where...free`" declaration (as in Curry programs). This avoids the introduction of free variables in initial expressions by typos (which might lead to the exploration of infinite search spaces). If the free variable mode is on, each undefined symbol in an initial expression is considered as a free variable.

`+/-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.

`+/-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` 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` 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` Verbose mode (initially, it is off). If the verbose mode is on, the initial expression of a computation (together with its type) is printed before this expression is evaluated.

`+/-warn` 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.

`path` *path* Set the additional search path for loading modules to *path*. Note that this search path is only used for loading modules inside this invocation of PAKCS and not in application programs (e.g., Curry programs using the operation `System.getLoadPathForFile`).

`printdepth` *n* Set the depth for printing terms to the value `n` (initially: 10). 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` Show a help text on the "`:set` *option*" command together with the current values of all options.

`:show` Show the source text of the currently loaded Curry program. If the source text is not available (since the program has been directly compiled from a FlatCurry or XML file), the loaded program is decompiled and the decompiled Curry program text is shown.

`: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 current state of the system (together with the compiled program `prog.curry`) in the file `prog.state`, i.e., you can later start the program again by typing "`prog.state`" as a Unix command.

`:save` *expr* Similar as "`:save`" but the expression *expr* (typically: a call to the main function) will be executed after restoring the state and the execution of the restored state terminates when the evaluation of the expression *expr* terminates.

`: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 Section 2.4.3) 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 [7].

`:xml` Translate the currently loaded program module into an XML representation according to the format described in http://www.informatik.uni-kiel.de/~curry/flat/. Actually, this yields an implementation-independent representation of the corresponding FlatCurry program (see Section 2.4.4 for a description of FlatCurry). If *prog* is the name of the currently loaded program, the XML representation will be written into the file "*prog*`_flat.xml`".

`: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].

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 for the "`:set`" command:**

`+/-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.

2.2 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.

2.3 Emacs Interface

Emacs is a powerful programmable editor suitable for program development. It is freely available for many platforms (see http://www.emacs.org or http://www.xemacs.org). The distribution of PAKCS contains also a special *Curry mode* that supports the development of Curry programs in the (X)Emacs environment. This mode includes support for syntax highlighting, finding declarations in the current buffer, and loading Curry programs into the PAKCS/Curry2Prolog compiler system 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.

2.4 Libraries for Application Programming

The PAKCS/Curry2Prolog compiler system provides a number of system libraries for application programming. In particular, it has libraries for

- arithmetic constraints over real numbers,
- finite domain constraints,
- ports for concurrent and distributed programming,
- meta-programming by representing Curry programs in Curry,

and many more. These libraries are sketched in the following sections. For a more detailed online documentation of all libraries of PAKCS, see http://www.informatik.uni-kiel.de/~pakcs/lib/index.html.

2.4.1 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

```
import CLPR
```

Floating point arithmetic is supported in Curry2Prolog 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:

`(+.) :: Float -> Float -> Float`
Addition on floating point numbers.

`(-.) :: Float -> Float -> Float`
Subtraction on floating point numbers.

`(*.) :: Float -> Float -> Float`
Multiplication on floating point numbers.

`(/.) :: Float -> Float -> Float`
Division on floating point numbers.

`(<.) :: Float -> Float -> Success`
Comparing two floating point numbers with the "less than" relation.

`(>.) :: Float -> Float -> Success`
Comparing two floating point numbers with the "greater than" relation.

`(<=.) :: Float -> Float -> Success`
Comparing two floating point numbers with the "less than or equal" relation.

`(>=.) :: Float -> Float -> Success`
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):

```
import CLPR

mortgage p t ir r b | t >. 0.0 & t <=. 1.0  --lifetime not more than 1 month?
                    =  b =:= p *. (1.0 +. t *. ir) -. t*.r

mortgage p t ir r b | t >. 1.0              --lifetime more than 1 month?
                    =  mortgage (p *. (1.0+.ir)-.r) (t-.1.0) 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

```
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.

2.4.2 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:[2]

`domain :: [Int] -> Int -> Int -> Success`
: The constraint "`domain` $[x_1,\ldots,x_n]$ l u" is satisfied if the domain of all variables x_i 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 -> Success`
: Equality of finite domain values.

`(/=#) :: Int -> Int -> Success`
: Disequality of finite domain values.

`(<#) :: Int -> Int -> Success`
: "less than" relation on finite domain values.

`(<=#) :: Int -> Int -> Success`
: "less than or equal" relation on finite domain values.

`(>#) :: Int -> Int -> Success`
: "greater than" relation on finite domain values.

`(>=#) :: Int -> Int -> Success`
: "greater than or equal" relation on finite domain values.

`sum :: [Int] -> (Int -> Int -> Success) -> Int -> Success`
: The constraint "`sum` $[x_1,\ldots,x_n]$ op x" is satisfied if all $x_1+\cdots+x_n$ op x is satisfied, where op is one of the above finite domain constraint relations (e.g., "`=#`").

`scalar_product :: [Int] -> [Int] -> (Int -> Int -> Success) -> Int -> Success`
: The constraint "`scalar_product` $[c_1,\ldots,c_n]$ $[x_1,\ldots,x_n]$ op x" is satisfied if all $c_1x_1+\cdots+c_nx_n$ op x is satisfied, where op is one of the above finite domain constraint relations.

`count :: Int -> [Int] -> (Int -> Int -> Success) -> Int -> Success`
: The constraint "`count` k $[x_1,\ldots,x_n]$ 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.

[2] 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.

`all_different :: [Int] -> Success`

The constraint "`all_different` $[x_1, \ldots, x_n]$" is satisfied if all x_i have pairwise different values.

`labeling :: [LabelingOption] -> [Int] -> Success`

The constraint "`labeling` *os* $[x_1, \ldots, x_n]$" non-deterministically instantiates all x_i to the values 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 &
        all_different 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`}.

2.4.3 Ports: Distributed Programming in Curry

To support the development of concurrent and distributed applications, PAKCS supports internal and external ports as described in [10].[3] Since [10] 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.

[3] Ports are also supported by the TasteCurry interpreter, see Appendix D, and by the Curry2Java compiler, see Appendix C. However, the TasteCurry interpreter allows only to send strings over external ports and the Curry2Java compiler does not yet support the sending of logical variables over external ports.

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] -> Success`
: 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 -> Success`
: 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. However, as discussed in [10], 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 system 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 the PAKCS system 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 the PAKCS system is started*, then all connections to external ports and all messages sent and received on external ports are printed on the standard error stream.

2.4.4 AbstractCurry and FlatCurry: Meta-Programming in Curry

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

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

Since all datatypes are explained in detail in these modules, we refer to the online documentation[4] of these modules.

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

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

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

```
(Prog "test"
 ["Prelude"]
```

[4] http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/FlatCurry.html and http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/AbstractCurry.html

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

2.4.5 Further System Modules

There are a number of other system modules supported by PAKCS which are only listed but not described in detail here. Look into the system module directories "*pakcshome*`/lib`" (and subdirectories) for their definitions.

`AbstractCurryPrinter`**:** This library contains a pretty printer for `AbstractCurry` programs in order to show an `AbstractCurry` program (or part of it) in standard Curry syntax.

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

`Array`**:** An implementation of arrays with Braun Trees.

`Char`**:** Some useful functions on characters.

`CSV`**:** This library supports reading/writing files in CSV (comma separated values) format that can be imported and exported by most spreadsheed and database applications.

`Combinatorial`**:** This library contains a collection of common non-deterministic and/or combinatorial operations.

`DaVinci`**:** This library supports the visualization of graphs by a binding to the *daVinci graph drawing tool*[5].

`Dequeue`**:** This library contains an implementation of double-ended queues supporting access at both ends in constant amortized time.

[5] http://www.tzi.de/daVinci/

`Directory`: This library supports the access to the directory structure of the underlying operating system.

`Dynamic`: This library supports the manipulation of dynamic predicates, i.e., predicates that are defined by facts than can change over time and can be persistently stored. The ideas of this library are described in [13].

`FileGoodies`: A collection of useful operations when dealing with files.

`FiniteMap`: An implementation of finite maps. A finite map is an efficient purely functional data structure to store a mapping from keys to values.

`FlatCurryGoodies`: Library containing selector functions, test and update operations as well as some useful auxiliary functions for manipulating FlatCurry data terms.

`FlatCurryShow`: Library containing some functions to transform FlatCurry programs into string representations, either in a FlatCurry format or in a Curry-like syntax.

`FlatCurryXML`: This library contains functions to convert FlatCurry programs into XML terms and vice versa. A detailed specification of the XML format used here can be found in `http://www.informatik.uni-kiel.de/~curry/flat/`.

`Float`: A collection of operations on floating point numbers.

`GUI`: A library for GUI programming in Curry. The concept behind this library exploits the functional and logical features of Curry. It is described in [11]. However, this library is an improved and updated version of the older library `Tk`. The latter might not be supported in the future.

`HTML`: Library for HTML and CGI programming. A detailed description of this library and its basic ideas can be found in [12].

`HTML_Parser`: Library for parsing HTML documents. It mainly exports a function containing a file name or URL containing an HTML document and returns an HTML term (as defined in module `HTML`) representing this document.

`Integer`: A collection of common operations on integer numbers.

`IO`: This library contains some I/O operations, like reading and writing files, that are not already contained in the prelude.

`IOExts`: This library contains some useful extensions to the IO monad, in particular, the implementation of a global state.

`List`: Some useful operations on lists that are not contained in the prelude.

`Maybe`: This library contains some useful operations on the `Maybe` type which are not contained in the prelude.

`Parser`: A library of functional logic parser combinators. This has been adapted from [8] where you can find a detailed description of the ideas behind these parser combinators.

`Random`: A library for generating pseudo-random numbers sequences.

`ReadNumeric`: A library with functions for reading and converting numeric tokens in strings, like natural or integer numbers.

`Read`: A library with functions for reading special tokens in strings, e.g., converting strings into natural or integer numbers.

`ReadShowTerm`: A library with functions for converting ground data terms to strings and vice versa. This is useful for storing data terms in files and reading them back.

`SetRBT`: This library provides an efficient implementation of sets as red-black trees. The implementation is generic in the types of elements and the set operations require an ordering predicate on elements. The library also contains a generic sort function with complexity $O(n \log(n))$ based on insertion into red-black trees.

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

`System`: A library to access parts of the system environment (like date, environment variables, system calls etc).

`TableRBT`: This library provides an efficient implementation of tables (i.e., finite mappings from keys to values) as red-black trees. The implementation is generic in the types of keys and values and the table operations require an ordering predicate on keys.

`Time`: This library contains definitions and functions to handle date and time information.

`Tk`: A (deprecated) library for GUI programming in Curry. This library might not be supported in the future (see library `GUI` for the current library for GUI programming).

`Unsafe`: This library contains unsafe operations which should not be used.

`URL`: This library contains functions related to URLs, in particular, downloading of documents accessible by a URL.

`WUI`: This library supports the type-oriented construction of Web User Interfaces (WUIs). The ideas behind the application and implementation of WUIs are described in [16].

`XML`: Library for processing data in XML format. It contains a definition of a datatype for representing XML terms and a parser and pretty printing for converting strings into such XML terms and vice versa.

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

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

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

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

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

3.2 Function Patterns

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

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

```
last xs | ys++[y] =:= xs  = y   where y,ys 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.

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

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

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

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

which shows that the evaluation of the list elements (except for the last one) is not demanded.

In general, a pattern of the form `(`$f\ t_1 \ldots t_n$`)` $(n > 0)$ is interpreted as a function pattern if f is not a visible constructor but a defined function that is visible in the scope of the pattern.

Optimization of programs containing function 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 function patterns that are guaranteed to evaluate to linear terms, there is an optimizer for function patterns that checks for occurrences of function 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

  ```
  export FCYPP="-fpopt"
  ```

  Then the function 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 function pattern optimization is applied if this program is compiled and loaded in PAKCS.

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

3.3 Records

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

Unlike labeled data fields in Haskell, records are not syntactic sugar but a real extension of the language[6]. The basic concept is described in [21] but the current version does not yet provide all features mentioned there. The restrictions are explained in Section 3.3.7.

3.3.1 Record Type Declaration

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

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

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

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

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

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

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

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

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

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

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

getName :: Person -> String
getName ...
```

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

Like in type synonym declarations, recursive or mutually dependent record declarations are not allowed. Records can only be declared at the top level. Further restrictions are described in section 3.3.7.

3.3.2 Record Construction

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

$\{\ l_1 = v_1, \ \ldots, \ l_m = v_m\ \}$

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

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

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

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

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

```
{person = {name = "Smith", age = 20},   -- generates a record of
```

```
street = "Main Street",                -- type 'Address'
city   = "Springfield"}
```

3.3.3 Field Selection

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

r `->` l

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

```
pers -> name
```

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

```
(addr -> person) -> age
```

The value of the label `age` is extracted from a record which itself is the value of the label `person` in the record `addr` (which has the type `Address`). When a field selection is used in expressions, it has to be parenthesized.

3.3.4 Field Update

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

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

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

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

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

3.3.5 Records in Pattern Matching

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

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

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

When trying to match a record against a record pattern, the patterns of the specified labels are matched against the corresponding values in the record expression. On success, all pattern

variables occurring in the patterns are replaced by their actual expression. If none of the patterns matches, the computation fails.

Here are some examples of pattern matching with records:

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

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

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

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

3.3.6 Export of Records

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

- `module` M `(...,` R`, ...) where`
  exports the record R without any of its labels.
- `module` M `(...,` R`(..), ...) where`
  exports the record R together with all its labels.
- `module` M `(...,` R`(`l_1`,...,`l_k`), ...) where`
  exports the record R together with the labels l_1, ..., l_k.

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

3.3.7 Restrictions in the Usage of Records

In contrast to the basic concept in [21], PAKCS/Curry provides a simpler version of records. Some of the features described there are currently not supported or even restricted.

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

- Labels cannot be sequentially associated with multiple values (record fields do not behave like stacks).

4 CurryDoc: 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.

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.[7] The documentation comments for the complete module occur before the first "module" or "import" line in the module. The comments can also contain several special tags. These tags must be the first thing on its line (in the documentation comment) and continues until the next tag is encountered or until the end of the comment. The following tags are recognized:

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

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

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

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

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

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

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

module Example where

--- The function conc concatenates two lists. It is defined
--- as flexible so that it can also be used to split a given list.
```

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

```
--- @param xs - the first list
--- @param ys - the second list
--- @return a list containing all elements of xs and ys
conc eval flex
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.
--- @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 datatype defines polymorphic trees.
--- @cons Leaf - a leaf of the tree
--- @cons Node - an inner node of the tree
data Tree a = Leaf a | Node [Tree a]
```

To generate the documentation, execute the command

```
currydoc Example
```

(`currydoc` is a command usually stored in *pakcshome*`/bin` where *pakcshome* is the installation directory of PAKCS; see Section 1.1). This command creates the directory `DOC_Example` (if it does not exist) and puts all HTML documentation files for the main program module `Example` and all its imported modules in this directory together with a main index file `index.html`. If one prefers another directory for the documentation files, one can also execute the command

```
currydoc docdir Example
```

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

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

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

`currydoc onlyindex docdir Mod1 Mod2 ...Mod`n `:` This command generates only the index pages (i.e., a main index page and index pages for all functions and constructors defined in the modules `Mod1`, `M2`,...,`Mod`n and their imported modules) in the directory `docdir`.

5 CurryBrowser: A Tool for Analyzing and Browsing Curry Programs

CurryBrowser is a tool to browse through the modules and functions of a Curry application, show them in various formats, and analyze their properties.[8] Moreover, it is constructed in a way so that new analyzers can be easily connected to CurryBrowser. A detailed description of the ideas behind this tool can be found in [14, 15].

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

- In the command shell via the command: *pakcshome*`/bin/currybrowser mod`
- In the PAKCS/Curry2Prolog environment after loading the module `mod` and typing the command "`:browse`".

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

To get an impression of the use of CurryBrowser, Figure 1 shows a snapshot of its use on a particular application (here: the implementation of CurryBrowser). The upper list box in the left column shows the modules and their imports in order to browse through the modules of an application. Similarly to directory browsers, the list of imported modules of a module can be opened or closed by clicking. After selecting a module in the list of modules, its source code, interface, or various other formats of the module can be shown in the main (right) text area. For instance, one can show pretty-printed versions of the intermediate flat programs (see below) in order to see how local function definitions are translated by lambda lifting [20] or pattern matching is translated into case expressions [9, 23]. Since Curry is a language with parametric polymorphism and type inference, programmers often omit the type signatures when defining functions. Therefore, one can also view (and store) the selected module as source code where missing type signatures are added.

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

The function list box can be also filled with functions via the menu "Select functions". For instance, all functions or only the exported functions defined in the currently selected module can be shown there, or all functions from different modules that are directly or indirectly called from a currently selected function. This list box is central to focus on a function in the source code of some module or to analyze some function, i.e., showing their properties. In order to focus on a function, it is sufficient to check the "focus on code" button. To analyze an individually selected function, one can select an analysis from the list of available program analyses (through the menu "Select analysis"). In this case, the analysis results are either shown in the text box below the main text

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

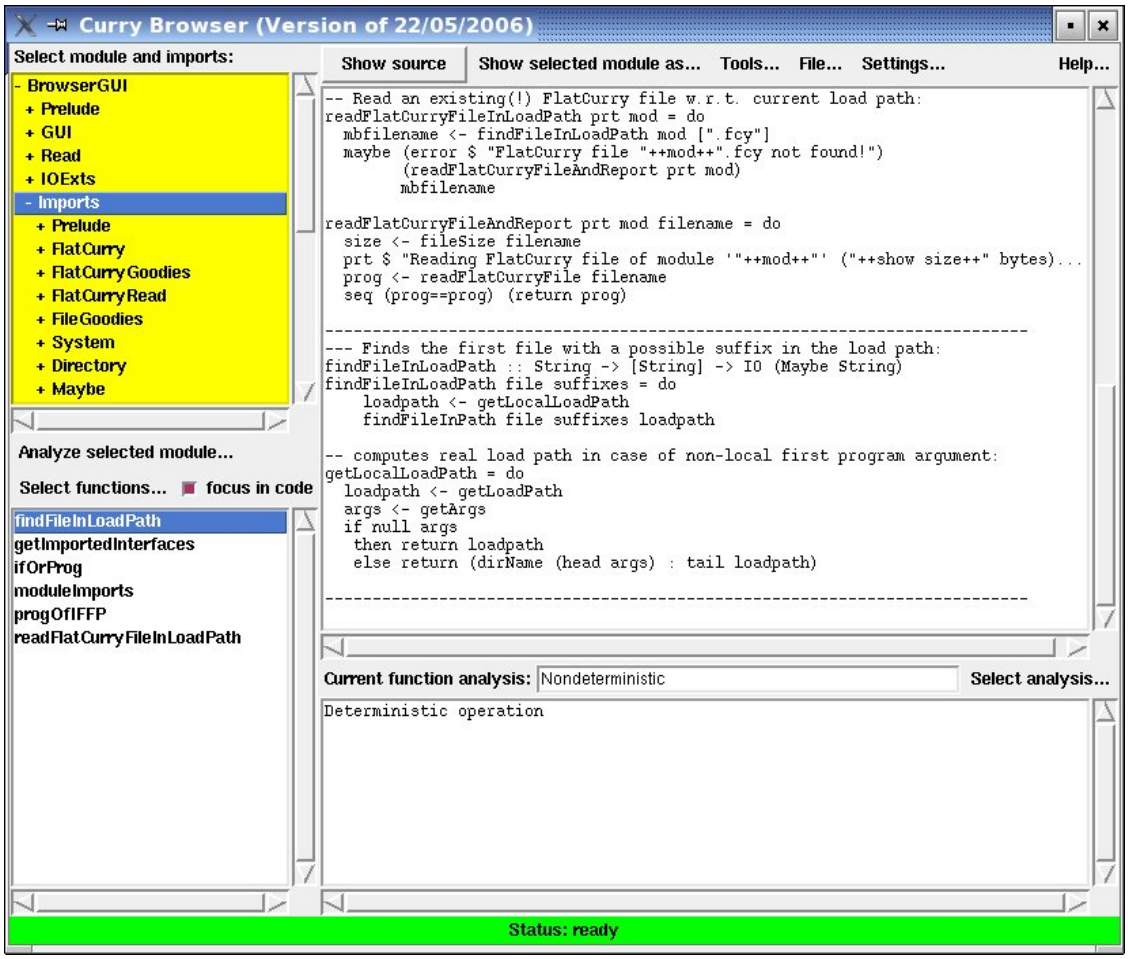

Figure 1: Snapshot of the main window of CurryBorwser

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.

6 CurryTest: A Tool for Testing Curry Programs

CurryTest is a simple tool in the PAKCS distribution to write and run repeatable tests. CurryTest simplifies the task of writing test cases for a module and executing them. The tool is easy to use. Assume one has implemented a module `MyMod` and wants to write some test cases to test its functionality, making regression tests in future versions, etc. For this purpose, there is a system library `Assertion` which contains the necessary definitions for writing tests. In particular, it exports the following datatype:

```
data Assertion a = AssertTrue      String Bool
                 | AssertEqual     String a a
                 | AssertValues    String a [a]
                 | AssertSolutions String (a->Success) [a]
                 | AssertIO        String (IO a) a
                 | AssertEqualIO   String (IO a) (IO a)
```

The expression "`AssertTrue` s b" is an assertion (named s) that the expression b has the value `True`. Similarly, the expression "`AssertEqual` s e_1 e_2" asserts that the expressions e_1 and e_2 must be equal (i.e., e_1==e_2 must hold), the expression "`AssertValues` s e vs" asserts that vs is the multiset of all values of e, and the expression "`AssertSolutions` s c vs" asserts that the constraint abstraction c has the multiset of solutions vs. Furthermore, the expression "`AssertIO` s a v" asserts that the I/O action a yields the value v whenever it is executed, and the expression "`AssertEqualIO` s a_1 a_2" asserts that the I/O actions a_1 and a_2 yields equal values. The name of each assertion is used in the protocol of the test tool.

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

```
import List
import Assertion

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

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

test3 = AssertSolutions "prefix" (\x -> let y free in  x++y =:= [1,2])
                                 [[],[1],[1,2]]
```

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

We can execute a test suite by the command

```
currytest testList
```

(`currytest` is a program stored in *pakcshome*`/bin` where *pakcshome* is the installation directory of PAKCS; see Section 1.1). In our example, "`testList.curry`" is the program containing the definition of all assertions. This has the effect that all exported top-level functions of type `Assertion` are tested (i.e., the corresponding assertions are checked) and the results ("`OK`" or failure) are reported together with the name of each assertion. For our example above, we obtain the following successful protocol:

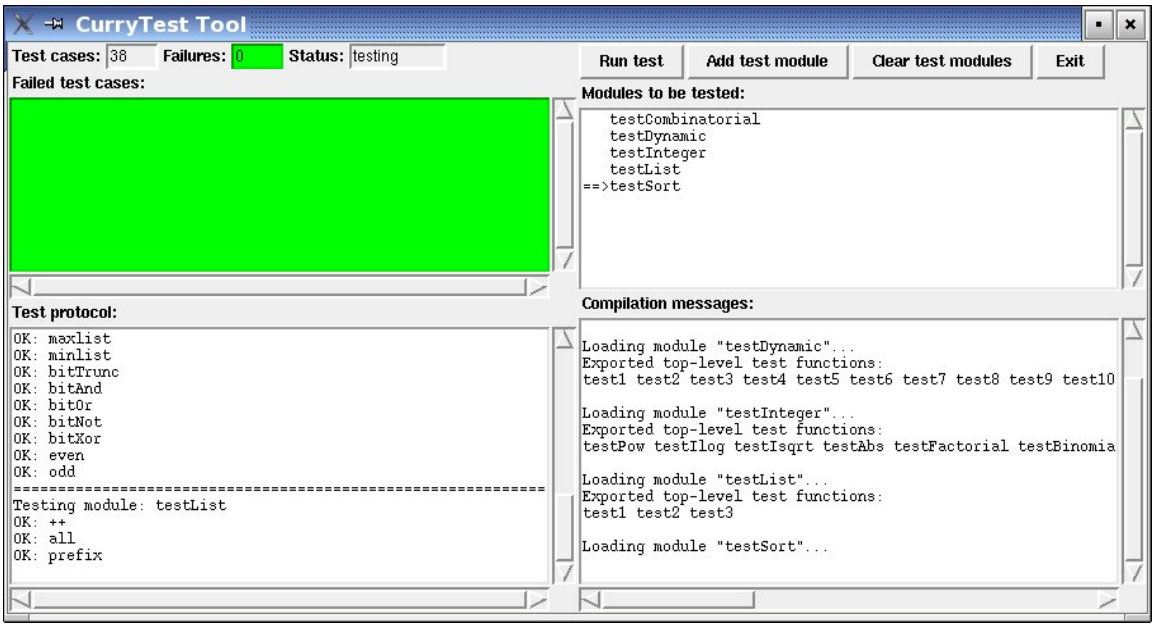

Figure 2: Snapshot of CurryTest's graphical interface

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

There is also a graphical interface that summarizes the results more nicely.[9] In order to start this interface, one has to add the parameter "`-window`", e.g., executing a test suite by

```
currytest -window testList
```

A snapshot of the interface is shown in Figure 2.

[9]Due to a bug in older versions of SICStus-Prolog, it works only with SICStus-Prolog version 3.8.5 (or newer).

7 Technical Problems

Due to the fact that Curry is intended to implement distributed systems (see Section 2.4.3), 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 8766: This port is used by the **Curry Port Name Server** (CPNS) to implement symbolic names for ports in Curry (see Section 2.4.3). If some other process uses this port on the machine, the distribution facilities defined in the module `Ports` (psee Section 2.4.3) 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 "`netstat -a | fgrep 8766`" (or similar).

The CPNS is implemented as a demon listening on its port 8766 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 scripts *pakcshome*`/cpns/start` and *pakcshome*`/cpns/stop`. If the demon is already running, the command *pakcshome*`/cpns/start` does nothing (so it can be always executed before invoking a Curry program using ports).

If you detect any further technical problem, please write to

`mh@informatik.uni-kiel.de`

References

[1] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A partial evaluation framework for Curry programs. In *Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, pages 376–395. Springer LNCS 1705, 1999.

[2] E. Albert, M. Hanus, and G. Vidal. Using an abstract representation to specialize functional logic programs. In *Proc. of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR 2000)*, pages 381–398. Springer LNCS 1955, 2000.

[3] E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. In *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pages 326–342. Springer LNCS 2024, 2001.

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

[5] S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

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

[7] B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing functional logic computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 193–208. Springer LNCS 3057, 2004.

[8] R. Caballero and F.J. López-Fraguas. A functional-logic perspective of parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 85–99. Springer LNCS 1722, 1999.

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

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

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

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

[13] M. Hanus. Dynamic predicates in functional logic programs. In *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)*, pages 62–73, Aachen (Germany), 2004. Technical Report AIB-2004-05, RWTH Aachen.

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

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

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

[17] M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(6), 1999.

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

[19] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.

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

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

[22] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 146–159, 1997.

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

A 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 3 on page 36. 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 B below). The PAKCS distribution contains a common front end for reading (parsing and type checking) Curry programs and three different back ends for executing them:[10]

1. The **Curry2Prolog** compiler is currently the most efficient implementation of Curry inside PAKCS. Due to its simple user interface (e.g., it can be used without any knowledge about PAKCS and its translation process) and its advanced debugging features, we recommend the use of the Curry2Prolog compiler system for most applications. Therefore, the programming environment with the integrated Curry2Prolog compiler is available by the executable "`pakcs`" (see Section 1.1 for the general use of PAKCS). Moreover, it also contains constraint solvers for arithmetic constraints over real numbers and finite domain constraints, and further libraries for GUI programming, meta-programming etc. Currently, it does not implement encapsulated search in full generality (only a strict version of `findall` is supported), and concurrent threads are not executed in a fair manner.

2. The **Curry2Java** compiler [17] translates Curry programs into Java classes (to be precise, the Pizza extension [22] of Java is used). The most distinctive feature of this implementation is the use of Java threads to implement disjunctive computations at the top-level and concurrent conjunctions of constraints (i.e., it implements OR- and AND-parallelism via Java threads). These threads are executed in a fair manner in contrast to the Curry2Prolog compiler. Although the execution speed of the generated programs are acceptable for many applications, this implementation inherits the lack of efficiency of current Java implementations. In particular, the Java compiler needs a lot of time to translate Curry programs into JVM code.

3. The **TasteCurry Interpreter** is a slow but fairly complete implementation of Curry. It is an interpreter written in Prolog and does not implement sharing but uses pure term rewriting for executing programs. It should only be used to run smaller programs involving advanced language constructs like committed choice or encapsulated search. Since this interpreter is a non-sharing implementation, it often evaluates complex terms in a very inefficient way and computes, in the presence of non-deterministic functions, sometimes results which are not conform with the language definition.

B 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 only use the Curry2Prolog compiler system, the Curry2Java compiler, or

[10] Note that only the Curry2Prolog compiler will be installed in the standard installation. See Appendix C and D how to install the other back ends.

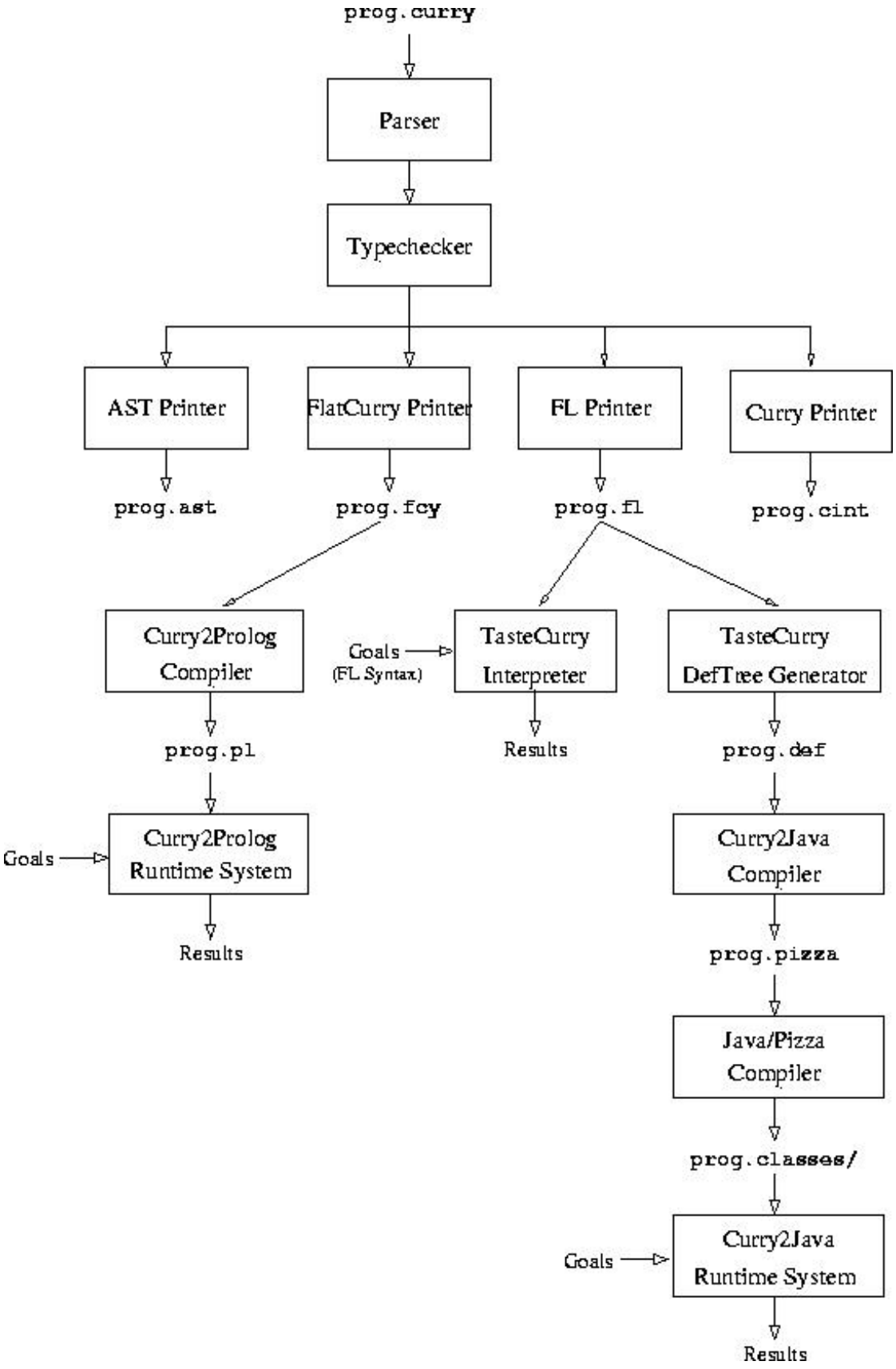

Figure 3: Overview of PAKCS

the TasteCurry interpreter, 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.fl`: This file contains the Curry program translated into the internal TasteCurry syntax (see Section D.3). It is implicitly generated when the TasteCurry interpreter or the Curry2Java compiler is used. It can be also explicitly generated by the command

```
parsecurry -fl prog
```

`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 Section 2.4.4). This representation might be useful for other back ends and compilers for Curry and is the basis doing meta-programming in Curry. This file is implicitly generated when a program is read by the Curry2Prolog compiler. It can be also explicitly generated by the command

```
parsecurry -fcy prog
```

The FlatCurry representation of a Curry program is usually generated by the front-end after parsing, type checking and eliminating local declarations.

`prog.fint`: This file contains the interface of the program in the so-called "FlatCurry" representation, i.e., it is similar to `prog.fcy` but contains only exported entities and the bodies of all functions omitted (i.e., "external"). This representation is useful for providing a fast access to module interfaces. This file is implicitly generated by the command

```
parsecurry -fcy prog
```

`prog.pl`: This file contains a Prolog program as the result of translating the Curry program with the Curry2Prolog compiler.

`prog.po`: This file contains the Prolog program `prog.pl` in an intermediate format for faster loading.

`prog.state`: This file contains the saved state after compiling and saving a program in the Curry2Prolog compiler (see Section 2.1).

`prog.def`: This file contains an intermediate representation of the Curry program which will be used by the Curry2Java compiler. This file is implicitly generated when a program is compiled with this compiler. It can be also explicitly generated by the command

```
parsecurry -def prog
```

`prog.pizza`: This implicitly generated file contains a Java (more precisely, Pizza) program as the result of translating the Curry program with the Curry2Java compiler.

`prog.classes`: This directory contains the JVM code of the compiled `prog.pizza` file.

C Curry2Java: A Compiler from Curry into Java

The Curry2Java compiler translates Curry programs into Java programs[11] as described in [17] and contains a runtime system to execute the translated programs with different expressions. This compiler translates each defined Curry function into a Java class containing instructions of an abstract machine which is interpreted by the runtime system. Although this indirect execution is not highly efficient due to the current implementations of Java systems, it has several interesting features. The most distinctive one is the use of Java threads to implement disjunctive computations at the top-level and concurrent conjunctions of constraints (i.e., it implements OR- and AND-parallelism via Java threads). In particular, an infinite derivation branch at the top-level will not inhibit the computation of solutions by other alternative branches.

The Curry2Java can be installed by executing "`make curry2java`" in the installation directory of PAKCS. To start the Curry2Java system, go into the directory where you have stored your Curry program and execute the command

```
curry2java prog
```

(`curry2java` is a shell script usually stored in *pakcshome*`/bin` where *pakcshome* is the installation directory of PAKCS; see Section 1.1). This command reads the file `prog.curry` which must contain a Curry program (or, if `prog.curry` does not exist, the file `prog.fl` which must contain a program in internal TasteCurry syntax) and performs the following compilation steps:

1. Parse the program in `prog.curry` and translate it into a corresponding program in internal TasteCurry syntax which will be stored in `prog.fl`.

2. Read and check the program file `prog.fl` and generate an intermediate representation of all functions in `prog.def`.

3. Read the function definitions stored in `prog.def` and translate them into a Java (more precisely, Pizza) program `prog.pizza`.

4. Compile the program `prog.pizza` into Java bytecode (which is stored in the directory `prog.classes`) and start the runtime system.

After the successful compilation, you can type in an expression to be evaluated. Expressions have the usual Curry syntax but there are some restrictions for initial expressions in the Curry2Java runtime shell:

1. All *applications* must be written in the prefix notation "$f\ arg_1 \ldots arg_n$", i.e., there are no infix operators. For instance, an arithmetic expression must be written in prefix notation like "`+ 3 (* 5 6)`".

2. *Lists* can be written in the standard notation `[`e_1`,`e_2`,...,`e_n`]`. Thus, to increment all elements in a list, one can write "`map (+ 1) [3,4,5]`". One can also use the constructor "`:`" for lists, i.e., the list `[3,4,5]` can be also written as "`: 1 (: 2 (: 3 []))`". Since the character `]` can also occur in identifiers, *a separator must be inserted if the last element in a list is an identifier*, e.g., one must write "`[True,False ]`".

[11] More precisely, Curry2Java uses the Java extension Pizza.

3. The *concurrent conjunction of constraints* is written with the operator `/\` (and not with `&`). Furthermore, the symbol `=` is used instead of `=:=` for equational constraints. For instance, the constraint `x+x=:=y & x=:=3` is written in the Curry2Java runtime shell as "`/\ (= (+ x x) y) (= x 3)`".

To leave the Curry2Java runtime system, type the end-of-file character (Ctrl-D).

Quiet mode. You can also execute the Curry2Java system in a "quiet" mode by

```
curry2java -q prog
```

If the program `prog` was already compiled in a previous session, then no system output is produced (except for the output computed in the Curry program). This option is useful if you want to write Curry programs which should act as a filter or which should only generate some textual output (e.g., in cgi scripts for WWW applications). For instance, if the file `hello.curry` contains the simple program

```
main = putStrLn "Hello world."
```

which was compiled by a previous `curry2java` command, then the Unix command "`echo main | curry2java -q hello`" echos the string "`Hello world.`" on the standard output. If the file `hello.curry` contains the program

```
main =
 putStrLn "Content-type: text/html" >>
 putStrLn "" >>
 putStrLn "<HTML>" >>
 putStrLn "Hello <B>world</B>.<P>" >>
 putStrLn "This web page is generated by a Curry program." >>
 putStrLn "</HTML>"
```

and you execute this program via your web browser (by loading a cgi script containing the shell commands "`echo main | curry2java -q hello`") then the corresponding HTML page is produced by the Curry program.

Restrictions: Since the development of Curry2Java is no longer actively supported, the implemented subset of Curry is largely restricted.

D The TasteCurry Interpreter

D.1 How to Use the TasteCurry Interpreter

The TasteCurry interpreter can be installed by executing "`make tastecurry`" in the installation directory of PAKCS. To start the TasteCurry interpreter, go into the directory where you have stored your Curry program and execute the command "`tastecurry`" (it is a shell script stored in *pakcshome*`/bin` where *pakcshome* is the installation directory of PAKCS). When the interpreter is ready, you can type in the following commands:

`read prog.` Load the file `prog.curry` which must contain a valid Curry program. If the name `prog` contains other characters than only lower case letters, it must be enclosed in single

quotes (e.g., `read 'a2b'.`). After successful loading and checking, all functions and types defined in this file (plus the functions defined in the prelude) are known to the interpreter, i.e., now you can evaluate expressions containing these functions and constructors.

If there is no file `prog.curry`, then the system searches for the file `prog.fl` which must contain a Curry program in the internal TasteCurry syntax (see Section D.3). If there exists a file `prog.curry`, it is translated into internal TasteCurry syntax which is subsequently stored in the file `prog.fl`.

`<`*expression*`>.` Evaluate the <*expression*> w.r.t. the functions defined in the current program. The <*expression*> must be written in the internal TasteCurry syntax (see Section D.3) and terminated by a dot. Before the expression is evaluated, it is checked whether it is well typed. The result (in general, a disjunctive expression which is not further reducible) is printed on the terminal.

`trace.` Show each reduction step, i.e., show all intermediate expressions occurring during the evaluation of an expression.

`notrace.` Turn off the `trace` mode.

`single.` Turn on the single step execution mode. In this mode, the evaluation of an expression is stopped after each reduction step and the user is asked how to proceed (see the options there).

`nosingle.` Turn off the `single` mode.

`time.` After the evaluation of an expression, show the time needed to evaluate this expression.

`notime.` Turn off the `time` mode.

`opt.` Generate optimal definitional trees when reading the next Curry program.

`noopt.` Turn off the `opt` mode.

`type` <*expression*>`.` Show the type of the expression <*expression*> (which must be written in the internal TasteCurry syntax, see Section D.3).

`eval` f`.` Show the definitional tree of the function f.

`writeflat file.` Write the FlatCurry representation (see Section 2.4.4) of the Curry program read in before to the file `file.flat`.

`writeprelude file.` Write the FlatCurry representation of the prelude to the file `file.flat`.

`exit.` Leave the TasteCurry interpreter.

D.2 Restrictions on Curry Programs in the TasteCurry Interpreter

There is one additional minor restriction on Curry programs which are loaded into the TasteCurry interpreter.

The difference between *lowercase* and *uppercase letters* is significant in Curry. However, since the TasteCurry interpreter uses internally a Prolog like syntax (see below), the first character of function and constructor names is automatically transformed into a lowercase letter (this is important to know if you use the interactive TasteCurry interpreter, see below). Therefore, two different objects should not only differ in the case of their first letter. For instance, the following program produces a type error since the names `fun` and `Fun` are both converted into `fun` which causes a name clash in the TasteCurry interpreter:

```
fun = 0
Fun = True
```

D.3 Internal TasteCurry Syntax

Since the TasteCurry interpreter is implemented in Prolog and uses the Prolog parser for reading programs and expressions, Curry programs are parsed and translated into a Prolog-like syntax which is called *internal TasteCurry syntax* throughout this document. Since one can also write programs directly in this syntax (then the files must have the suffix "`.fl`"), we describe in the following the differences between Curry and the internal TasteCurry syntax:

- Every declaration (datatype, function type, and rule) must be terminated by a dot ("`.`") followed by a blank or newline.

- The *names of functions, constructors and type constructors* must start with a lowercase letter followed by a sequence of letters and digits. The following predefined function names in Curry are different in TasteCurry:

  `=` instead of `=:=`
  `/\` instead of `&`
  {} instead of `success`
  `constraint` instead of `Success`

  There are some predefined names consisting of special characters which can be used as infix operators similarly to Curry (e.g., the type constructor `->`, and the functions `==`, `=`, `/\`, `&&`).

- The *names of extra variables*, i.e., variables which do not occur in arguments of the left-hand side of a rule, should start with an underscore ("`_`") followed by a sequence of letters and digits. Otherwise, the TasteCurry interpreter will print a warning since this is a typical source of programming errors (typos in function names).

- The *application* of an object φ (type constructor, function, or data constructor) to n arguments $a_1, \ldots, a_n$ is written as $\varphi(a_1, \ldots, a_n)$ (which is denoted in Curry by $\varphi\ a_1 \ldots a_n$). If the first argument is not a simple name starting with a lowercase letter, the infix symbol `@` must be used to denote the application. For instance, if the function variable `F` should be applied

to some argument a, it *must* be written as `F@`a. `@` associates to the left, i.e., an application of a variable `F` to two arguments a_1, a_2 can be written as `F@`a_1`@`a_2.

- A *datatype declaration* is written in the form

  `data` t`(`A_1`,...,`A_n`) = `c_1`(`τ_{11}`,...,`τ_{1n_1}`) ; ... ; `c_k`(`τ_{k1}`,...,`τ_{kn_k}`).`

  where each τ_{ij} is a type expression built from the type variables $A_1, \ldots, A_n$ and some type constructors. In contrast to Curry, the single constructors are separated by "`;`" instead of "`|`".

- The *type of lists* with elements of type t is denoted by `list(`t`)`. The data constructor of a non-empty list is the dot "`.`" (instead of "`:`"). This data constructor is not defined as an infix operator. `[X|Xs]` is the notation for a non-empty list consisting of the head `X` and the tail `Xs`. Note that `[X|Xs]` is equivalent to the expressions "`.(X,Xs)`" and "`(.)@X@Xs`".

- *Characters* are identified with their ASCII values. Thus, the string `"Hello"` is identical to the integer list `[72,101,108,108,111]`. In particular, the standard monadic I/O actions for reading and writing characters or strings have in TasteCurry the types

```
getChar  :: io(int).
getLine  :: io(list(int)).
putChar  :: int       -> io(unit).
putStr   :: list(int) -> io(unit).
putStrLn :: list(int) -> io(unit).
```

- *Tuples* are not yet implemented. However, there is a data type `pair` which is predefined by the declaration

```
data pair(A,B) = (A,B).
```

  Thus, `(1,2)` denotes a pair of integers, and `(1,2,3)` has type `pair(int,pair(int,int))` (i.e., the comma is a right-associative infix operator).

- *Constraints* must be always enclosed in curly brackets (for an example, see the definition of `member` in the next paragraph).

- In a *conditional rule*, the symbol "`|`" introducing the condition is replaced by "`if`". For instance, the membership predicate based on list concatenation is defined in the internal TasteCurry syntax by

```
member :: T -> list(T) -> bool.
member(E,L)  if {append(_,[E|_])=L}  = true.
```

- A *lambda abstraction* always abstracts a single variable. For instance, the anonymous function with two arguments that adds its arguments must be written in the form

```
\X -> (\Y -> X+Y)
```

  In the initial expression (which is typed in after loading the program into the interpreter, see Section D.1), the use of lambda abstractions is even more restricted: in the initial expression, every subexpression of the form `\X->`e must satisfy:

1. e must be of type `constraint`.
2. e does not contain any lambda abstraction.

This is enough to allow the use of search operators in initial expressions. Other uses of lambda abstractions must always be written into the program.

- *Local variables in constraints* are introduced by the keywords `local...in` *inside* the constraint. Thus, the Curry expression

  ```
  let l1,l2 free in append l1 l2 =:= [0,1]
  ```

  is written in the internal TasteCurry syntax in the form

  ```
  {local [_l1,_l2] in append(_l1,_l2) = [0,1]}
  ```

  The square brackets around the local variables are only necessary if there is more than one variable. Therefore, the Curry expression

  ```
  let l free in append [0] l =:= [0,1]
  ```

  can be written in TasteCurry as

  ```
  {local _l in append([0],_l) = [0,1]}
  ```

- Instead of `where`-*clauses with free variables*, one has to introduce such free variables with the keyword `localIn` *before* the constraint. Thus, the Curry rule

  ```
  last l | append xs [e] =:= l  = e  where xs,e free
  ```

  is written in the internal TasteCurry syntax as

  ```
  last(L) if [Xs,E] localIn {append(Xs,[E])=L} = E.
  ```

  and the indeterministic merge function is written in the internal TasteCurry syntax as

  ```
  merge :: list(A) -> list(A) -> list(A).
  merge(L1,L2) = choice {L1=[]}  -> L2;
                        {L2=[]}  -> L1;
                        [E,R] localIn {L1=[E|R]} -> [E|merge(R,L2)];
                        [E,R] localIn {L2=[E|R]} -> [E|merge(L1,R)].
  ```

- *Positions in evaluation annotations* contain the separator "`#`" instead of the dot, e.g., the position `1.3.2` is denoted in the internal TasteCurry syntax by `1#3#2`.

The following function definitions (concatenation of lists and application of a function to all elements of a list) show further examples for the internal TasteCurry syntax.

```
append :: list(T) -> list(T) -> list(T).
append([],X)      = X.
append([X|Xs],Ys) = [X|append(Xs,Ys)].

map :: (T1->T2) -> list(T1) -> list(T2).
map(F,[])       = [].
map(F,[X|Xs])   = [F@X|map(F,Xs)].
```

Local Declarations

At the end of each defining equation for a function, local value and function declarations can be added by a *where clause*. A where clause is introduced by the keyword `where` followed by a semicolon-separated list of equations. Each equation defines either a local function (similar to top-level equations) or local variables (in this case the left-hand side must be a pattern). The newly introduced functions and variables can be used in the right-hand side of the equation where the where clause is added. Thus, a quicksort function by splitting the given list can be defined as follows:

```
split(E,[]) = ([],[]).
split(E,[X|Xs]) if E>=X = ([X|L],R)
                if E<X  = (L,[X|R])
                        where (L,R) = split(E,Xs).

qsort([])     = [].
qsort([X|Xs]) = qsort(L) ++ [X|qsort(R)]  where (L,R) = split(X,Xs).
```

Nested where clauses are not allowed. Furthermore, local declarations with patterns in the left-hand side should only contain in its right-hand side argument variables from the globally defined function and other global functions. The TasteCurry interpreter automatically translates all local declarations into global functions with additional arguments. Thus, the evaluation annotations for functions with local declarations look different from the original definition.

D.4 Modules in the TasteCurry Interpreter

In the current implementation of PAKCS, modules are only supported in the internal TasteCurry syntax. Moreover, the module system slightly differs from the module system described in the Curry report. Therefore, we give here a complete description of this module system in this section.

A *module* defines a collection of datatypes, constructors and functions which we call *entities* in the following. A module exports some of its entities which can be imported and used by other modules. An entity which is not exported is not accessible from other modules.

A Curry *program* is a collection of modules. There is one main module which is loaded into a Curry system. The modules imported by the main module are implicitly loaded but not visible to the user. After loading the main module, the user can evaluate expressions which contain entities exported by the main module.

There is one distinguished module, named `prelude`, which is implicitly imported into all programs. Thus, the entities defined in the prelude (basic functions for arithmetic, list processing etc.) can be always used.

A module always starts with the head which contains at least the name of the module, like

```
module stack.
```

If a program does not contain a module head, the *standard module head* "`module main.`" is implicitly inserted.

Without any further restrictions in the module head, all entities defined or imported in the module are exported. In order to restrict the exported entities of a module, an *export list* can be added to the module head. For instance, a module with the head

```
module stack(stackType, push, pop, newStack).
```

exports the entities `stackType`, `push`, `pop`, and `newStack`. An export list can contain the following entries:

1. Names of datatypes: This exports only the datatype defined in this module *but not* the constructors of the datatype. The export of a datatype without its constructors allows the definition of abstract datatypes.

2. Datatypes with constructors: If the export list contains the entry `t(`c_1`,...,`c_n`)`, then `t` must be a datatype defined in the module and $c_1,\ldots,c_n$ are constructors of this datatype. In this case, the datatype `t` and the constructors $c_1,\ldots,c_n$ are exported by this module.

3. Datatypes with all constructors: If the export list contains the entry `t(..)`, then `t` must be a datatype defined in the module. In this case, the datatype `t` and all constructors of this datatype are exported.

4. Names of functions: This exports the corresponding functions defined in this module. The types occurring in the argument and result type of this function are implicitly exported, otherwise the function may not be applicable outside this module.

5. Modules: The set of all entities imported from a module m into the current module (see below) can be exported by a single entry "`(module` m`)`" in the export list. For instance, if the head of the module `stack` is defined as above, the module head

   ```
   module queue((module stack), enqueue, dequeue).
   ```

   specifies that the module `queue` exports the entities `stackType`, `push`, `pop`, `newStack`, `enqueue`, and `dequeue`.

   If the exported entities from imported modules should be further restricted, one can also add an export list to the exported module. This list can contain names of datatypes and functions imported from this module. If a datatype which is imported from another module is exported, the datatype is exported in the same way (i.e., with or without constructors) how it is imported into the current module. Thus, a further specification for the exported constructors is not necessary. For instance, the module head

   ```
   module queue((module stack(stackType,newStack)), enqueue, dequeue).
   ```

   specifies that the module `queue` exports the entities `stackType` and `newStack`, which are imported from `stack`, and `enqueue` and `dequeue`, which are defined in `queue`.

The entities exported by a module can be brought into the scope of another module by an `import` declaration. An import declaration consists of the name of the imported module and (optionally) a list of entities imported from that module. If the list of imported entities is omitted, all entities exported by that module are imported. For instance, the import declaration

```
import stack.
```

imports all entities exported by the module `stack`, whereas the declaration

```
import family(father, grandfather).
```

imports only the entities `father` and `grandfather` from the module `family`, provided that they are exported by `family`.

The names of all imported entities are available in the current module, i.e., they are equivalent to top-level declarations. It is not allowed to write new top-level declarations for an imported entity, but the names can be shadowed by local declarations inside a function definition. As a consequence, several imports can only import different names. For instance, the imports

```
module main.
import m1.
import m2.
```

are only allowed if the entities exported by `m1` and `m2` have different names. In case of conflicting names of imported entities, one can rename imported entities to solve the name conflicts. For instance, if both `m1` and `m2` exports functions named `f` and `g`, then the conflict can be resolved by the following imports:

```
module main.
import m1.
import m2 renaming f to m2_f.
          renaming g to m2_g.
```

In the subsequent body of this module, the name `f` refers to the entity exported by module `m1` and the name `m2_f` refers to the entity `f` exported by module `m2`. Only imported entities can be renamed, i.e., the import declaration

```
import m(f) renaming g to mg.
```

will cause an error. Only entities which are also exported can be renamed.

The import dependencies between modules must be *non-circular*, i.e., it is not allowed that module m_1 imports module m_2 and module m_2 also imports (directly or indirectly) module m_1.

The explicit import of the prelude as a module is not allowed. For each module m, an interface stored in the file m`.int` is automatically generated. This interface describes all entities which are exported by the module, i.e., the datatypes with their exported constructors and the functions with their type declarations.

E Changing the Prelude or System Modules

The standard prelude, which is automatically imported into each Curry program, and all system modules containing datatypes and functions useful for application programming (cf. Section 2.4) are stored in the system module directory "*pakcshome*`/lib`" (and its subdirectories). If you change any of these modules, you have to recompile the complete system by executing `make` in the directory *pakcshome*.

F 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

   ```
   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:

```
(+) :: Int -> Int -> Int
(+) external
```

The further modifications to be done for an inclusion of an external function depend on the corresponding back end. In the following we describe the insertion of new external functions in Curry2Prolog and in the TasteCurry interpreter.

F.1 External Functions in Curry2Prolog

A new external function is added to the Curry2Prolog compiler system 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 Curry2Prolog compiler system:

1. If the Curry module `Mod` contains external functions, there must be a file named `Mod.prim_c2p` containing the specification of these external functions. The contents of this file is in XML format and has the following general structure:[12]

   ```
   <primitives>
     specification of external function f1
     ...
     specification of external function fn
   </primitives>
   ```

   The specification of an external function f with arity n[13] has the form

   ```
   <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 *pakcshome*`/curry2prolog/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 `Mod`: either as an external function or defined in Curry by equations. In the latter case, the Curry definition is not translated but calls to this function are redirected to the Prolog code specified above.

[12] http://www.informatik.uni-kiel.de/~pakcs/primitives.dtd contains a DTD describing the exact structure of these files.

[13] Note that I/O actions have a virtual "world" argument as the last argument which is not explicitly declared, e.g., `putChar` is a function with *two* arguments: the character to be printed and the "current world."

Furthermore, the list of specifications can also contain entries of the form

```
<ignore name="f" arity="n" />
```

for functions f with arity n that are declared in module `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) are 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

```
(+)   :: Int -> Int -> Int
x + y = (prim_Int_plus $# y) $# x

prim_Int_plus :: Int -> Int -> Int
prim_Int_plus external
```

where `prim_Int_plus` is the actual external function implementing the addition on integers. Consequently, the specification file `Prelude.prim_c2p` has an entry of the form

```
<primitive name="prim_Int_plus" arity="2">
  <library>prim_standard</library>
  <entry>prim_Int_plus</entry>
</primitive>
```

where the Prolog library `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 `prim_Int_plus` contained in the Prolog library `prim_standard.pl` is as follows (note that the arguments of `(+)` are passed in reverse order to `prim_Int_plus` in order to ensure a left-to-right evaluation of the original arguments by the calls to `($#)`):

   ```
   prim_Int_plus(Y,X,R) :- R is X+Y.
   ```

3. 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 Curry2Prolog run-time system by using the *raw interface*. In this case, the name of the external entry must be suffixed by "`[raw]`" in the `prim_c2p` file. For instance, if we want to use the raw interface for the external function `prim_Int_plus`, the specification file `Prelude.prim_c2p` must have an entry of the form

```
<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

```
?- 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:

```
?- 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 Curry2Prolog by putting it into the directory containing the corresponding Curry module or into the system directory *pakcshome*`/curry2prolog/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 the Curry2Prolog compiler system by using the corresponding interfaces of underlying Prolog system.

F.2 External Functions in TasteCurry

In the TasteCurry interpreter, you can add external functions only in the prelude. In addition to the declarations in the source code of the prelude as described above, you must also add the following in order to include a new external function in the TasteCurry interpreter:

1. The file *pakcshome*`/tastecurry/prelude.flpreface` contains standard declarations in the internal TasteCurry syntax which are added in front of the prelude. Provide a type declaration in the body of this module prefixed with the keyword `external` in front of the type declaration. For instance, the primitive addition on integers is declared in *pakcshome*`/tastecurry/prelude.flpreface` by

   ```
   external (+)   :: int -> int -> int.
   ```

Since the defining rules for this implementation are unknown, a call to an external function is delayed until all arguments are known (i.e., in head normal form). Thus, for each external function an evaluation annotation with a rigid annotation for each argument is automatically generated if no other evaluation annotation is provided. For instance, for the function `+` the annotation

```
(+) eval 1:rigid(_=>2:rigid(_=>rule))
```

is generated (the anonymous variable `_` denotes that the first argument must be matched against an arbitrary constant). Thus, in order to evaluate a call t_1`+`t_2, first t_1 is evaluated to a head normal form, and if this is not a variable, t_2 is evaluated to a head normal form, followed by a call to the external implementation of `+` provided that t_2 was not evaluated to a variable.

2. The connection of the implementation of an external function to the TasteCurry interpreter is done by adding a special clause in the module `external.pl` of the interpreter's sources (stored in the directory *pakcshome*`/tastecurry`). To implement an n-ary external function f, `external.pl` must contain the following Prolog clause:

   `external_call(`f`(X`$_1$`,...,X`$_n$`), Result) :-` <code computing the `Result`>

   For instance, the external function `+` is implemented by the following clause:

   ```
   external_call(+(X,Y),Result) :- Result is X+Y.
   ```

   By using the Prolog/C interface of SICStus-Prolog, arbitrary C functions can be connected to the TasteCurry interpreter.

3. After adding all these declarations, recompile the TasteCurry interpreter by executing `make` in the directory *pakcshome*.

Index