A Generic Analysis Environment for Declarative Programs [*]

Michael Hanus

Institut für Informatik, Christian-Albrechts-Universität Kiel
D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract

In this paper we present CurryBrowser, a generic analysis environment for the declarative multi-paradigm language Curry. Curry-Browser supports browsing through the implementation of an application written in Curry, i.e., the main module and all directly or indirectly imported modules. Each module can be shown in different formats (e.g., source code, interface, intermediate code) and, inside each module, various properties of functions defined in this module can be analyzed. In order to support the integration of various program analyses, CurryBrowser has a generic interface to connect local and global analyses implemented in Curry. Curry-Browser is completely implemented in Curry using libraries for GUI programming and meta-programming.

Categories and Subject Descriptors   D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming;   D.1.6 [*Programming Techniques*]: Logic Programming;   D.2.6 [*Software Engineering*]: Programming Environments;   F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms   Languages

Keywords   Functional Logic Programming, Programming Environments, Program Analysis

1. Overview

CurryBrowser is intended as a tool to support the analysis of declarative programs. It can be used to browse through an implementation written in the declarative multi-paradigm language Curry [6, 12], analyze properties of individual or all functions defined in a module, or visualize dependencies between modules or functions. It can be also used as a testbed for program analyses (the analyses of functional logic programs is still ongoing research) since it supports the easy integration of new program analyses by a generic interface. The implementation of CurryBrowser is based on an intermediate language to which functional, logic, and also integrated functional logic programs can be compiled (e.g., see [2, 4, 13]). Thus, it is also adaptable to other declarative languages.

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 [14] or pattern matching is translated into case expressions [6, 16]. 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.

The lower list box of the left column shows the list of exported or all functions defined in the selected module. This list box is central for the analysis of functions, i.e., showing their properties. For this purpose, the lower list box of the middle column offers a list of available program analyses that can be applied to individual functions. Their results are either shown in the text box below the main text area or visualized by separate tools, e.g., by a graph drawing tool for visualizing call graphs. Some analyses are local, i.e., they need only to consider the local definition of this function (e.g., "Calls directly," "Overlapping rules," "Pattern completeness"), where other analyses are global, i.e., they consider the definitions of all functions directly or indirectly called by this function (e.g., "Depends on," "Solution complete," "Set-valued"). The middle list box of the middle column allows to analyze all functions of the current module at once (in order to see some "critical" functions, like impure or incompletely defined ones). Finally, the upper list box is useful to focus on a function in the source code of some module. For this purpose, one can put there the list of all functions of the current module or all imported modules, or the list of all functions directly or indirectly called from the currently selected function. Selecting a function in this list shows its source code in the main text area. Furthermore, there are a few additional tools integrated into CurryBrowser, for instance, to visualize the import relation between all modules as a dependency graph.

In the next section, we review some features of Curry in order to show a few details of the implementation of CurryBrowser in Section 3. The currently available analyses and tools are sketched in Section 4 before we conclude in Section 5.

2. Curry Programs

Since CurryBrowser is implemented in Curry and intended to be applied to Curry programs, we review in this section some aspects of Curry programs that are necessary to understand the functionality and implementation of our programming environment. More details about Curry's computation model and a complete description of all language features can be found in [6, 12].
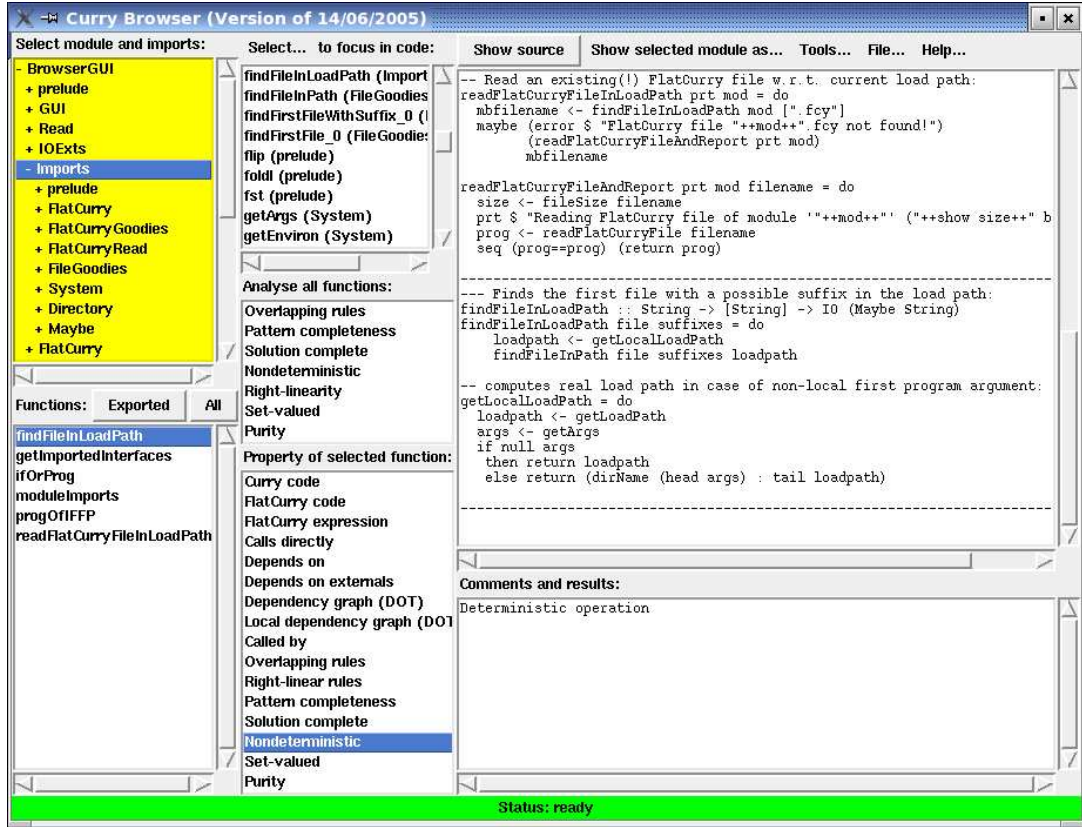
Figure 1. The main window of CurryBrowser

Curry is a declarative multi-paradigm language combining in a seamless way features from functional, logic, and concurrent programming and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [15], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of a function f to an argument e is denoted by juxtaposition ("$f\ e$").

A *Curry program* consists of the definition of functions and the data types on which the functions operate. Functions are defined by conditional equations with constraints in the conditions. They are evaluated lazily and can be called with partially instantiated arguments.

EXAMPLE 1. The following program defines the types of Boolean values and polymorphic lists and functions to concatenate lists and to compute the last element of a list:

```
data Bool   = True | False
data List a = []    | a : List a

conc :: [a] -> [a] -> [a]
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last xs | conc ys [x] =:= xs  = x  where x,ys free
```

The data type declarations define `True` and `False` as the Boolean constants and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over

all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell). The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[1]

The operational semantics of Curry [1, 6] is a conservative extension of lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. To describe this semantics, compile programs, or implement analyzers and similar tools, an intermediate representation of Curry programs has been shown to be useful. Programs of this intermediate language, also called *flat programs*, contain a single rule for each function where the pattern matching strategy is represented by case/or expressions. The basic structure of flat programs is defined as follows:[2]

$$
\begin{array}{llll}
P & ::= & D_1 \ldots D_m & \qquad e & ::= & v \\
D & ::= & f\ v_1 \ldots v_n = e & & | & c\ e_1 \ldots e_n \\
& & & & | & f\ e_1 \ldots e_n \\
p & ::= & c\ v_1 \ldots v_n & & | & case\ e_0\ of\ \{\overline{p_k \to e_k}\} \\
& & & & | & fcase\ e_0\ of\ \{\overline{p_k \to e_k}\} \\
& & & & | & e_1\ or\ e_2
\end{array}
$$

A program P consists of a sequence of function definitions D with pairwise different variables in the left-hand sides. The right-hand sides are expressions e composed by variables, constructor and

---

[1] Curry uses curried function types where $\alpha\texttt{->}\beta$ denotes the type of all functions mapping elements of type α into elements of type β.

[2] $\overline{o_k}$ denotes a sequence of objects $o_1, \ldots, o_k$.

function calls, case expressions, and disjunctions. A case expression has the form $(f)\,case\ e\ of\ \{c_1\ \overline{x_{n_1}} \to e_1, \ldots, c_k\ \overline{x_{n_k}} \to e_k\}$, where e is an expression, $c_1, \ldots, c_k$ are different constructors of the type of e, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are local variables which occur only in the corresponding subexpression e_i. The difference between *case* and *fcase* shows up when the argument e is a free variable: *case* suspends (which corresponds to residuation) whereas *fcase* nondeterministically binds this variable to the pattern in a branch of the case expression (which corresponds to narrowing).

The PAKCS implementation of Curry [9] provides a library for meta-programming which contains the data types for representing flat programs (i.e., the data types and functions defined in a module) and an I/O action for reading a module and translating its contents into the corresponding data term. For instance, a module of a Curry program is represented as an expression of the type

```
data Prog = Prog String
                 [String]
                 [TypeDecl]
                 [FuncDecl]
                 [OpDecl]
```

where the arguments of the data constructor `Prog` are the module name, the names of all imported modules, the list of all type, function, and infix operator declarations. Furthermore, a function declaration is represented as

```
data FuncDecl = Func QName Int Visibility
                     TypeExpr Rule
```

where the arguments are the qualified name (i.e., a pair of module and function name), arity, visibility (`Public` or `Private`), type, and rule (of the form "`Rule` *arguments expr*") of the function. The remaining data type declarations for representing Curry programs are similar but we omit them due to lack of space.

3. Implementation

CurryBrowser is implemented in Curry using libraries for GUI programming [8] and meta-programming sketched above. In order to ensure a fast startup time, only the interface files of all modules (they have the same structure as flat programs but contain only the type signature of exported functions and data types) are read at the beginning. This information is sufficient to show the import structure of all modules in the initial main window. Complete flat programs are only read as demanded by the analyses.

As discussed in Section 1, CurryBrowser offers a generic interface to integrate various analysis tools for declarative programs. Since flat programs are an appropriate abstraction level for implementing such tools [10, 11], this interface is based on them. To be more concrete, CurryBrowser provides the following type definition to connect program analyzers (where `a` is the type of the analysis result):

```
data FunctionAnalysis a =
    LocalAnalysis     (FuncDecl -> a)
  | LocalDataAnalysis ([TypeDecl] -> FuncDecl -> a)
  | GlobalAnalysis    ([FuncDecl] -> [(QName,a)])
  | GlobalDataAnalysis ([TypeDecl] -> [FuncDecl]
                                   -> [(QName,a)])
```

A local analysis associates results to single function definitions (e.g., "Calls directly"), a local data analysis requires in addition the type declarations (e.g., "Pattern completeness"), and global analyses require all defined functions and yield lists containing for each function name (`QName`) an associated result. Thanks to laziness, the results for all functions are not computed at once but only as demanded by the user.

As a simple example, consider the "Overlapping rules" analysis. This analysis is intended to indicate whether a function is defined by rules with overlapping left-hand sides. This can be easily spotted in the flat representation of Curry programs as an occurrence of a disjunction (*or*) in the right-hand side of the rule defining this function. Thus, this analysis is a local one that can be implemented as follows:

```
isOverlappingFunction :: FuncDecl -> Bool
isOverlappingFunction (Func _ _ _ _ (Rule _ e))
    = orInExpr e
```

where the operation `orInExpr` checks for occurrences of disjunctions in an expression.

For the simple addition of new analyzers, the implementation of CurryBrowser has a configuration module containing definitions of the currently available tools. For instance, it contains a constant `functionAnalyses` of type

```
[(String, FunctionAnalysis AnalysisResult)]
```

where each element in this list consists of the name and the function implementing the analysis. The result type of the concrete function analyses, defined as

```
data AnalysisResult =
          MsgResult String | ActionResult (IO ())
```

indicates whether the analysis result is a string or an I/O action to visualize the result via an external tool (e.g., a graph drawing tool). Thus, to test a new analysis by integrating it into CurryBrowser, one has to connect its implementation by adding a new element to the list `functionAnalyses` and recompile the system. Then, the CurryBrowser environment provides the analysis with the appropriate data whenever the user selects the analysis.

Note that the possible result types of analyses to be integrated into CurryBrowser are fixed since the implementation needs to know what to do with the analysis result. Therefore, the current implementation supports

- string results that are shown inside the main window, or
- I/O actions that calls some external tool for visualization.

For instance, to show the dependency graph of a function, the corresponding global analysis computes a graph structure and calls an external program, the DOT graph drawing tool[3], to visualize this graph structure (see Figure 2 for an example). Since the main GUI of CurryBrowser is executed in the I/O monad, the event handlers that implement reactions to user events are I/O actions [8]. Thus, analyses with a result type `IO ()` can be executed by the event handlers responsible for analyzing programs. In order to avoid the crash of the CurryBrowser environment if some analyses fails with run-time errors, the execution of an analyses is wrapped into an exception handler.

The restriction to a fixed set of analysis result types requires the transformation of arbitrary program analyses when they are integrated into CurryBrowser. For instance, the "Overlapping rules" analysis sketched above delivers Boolean results that must be converted into appropriate strings shown to the user. For this purpose, one can define a simple conversion operation to show the result of the overlapping analysis:

```
showOverlap :: Bool -> String
showOverlap True  = "Overlapping"
showOverlap False = "Not Overlapping"
```

In order to support a simple conversion of arbitrary analyses into the analyses with string results as required by the interface of

---

[3] http://www.graphviz.org/
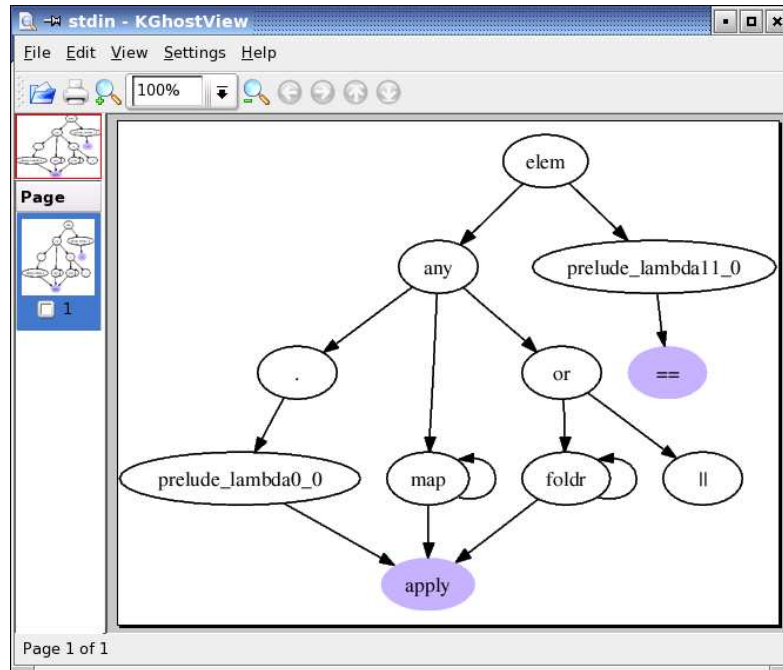
Figure 2. Visualization of the dependency graph of the prelude function `elem`

CurryBrowser, the implementation of CurryBrowser contains the following conversion operation:

```
showWithMsg :: FunctionAnalysis a
               -> (a->String)
               -> FunctionAnalysis AnalysisResult
showWithMsg (LocalAnalysis ana) showresult =
   LocalAnalysis
     (\f -> MsgResult (showresult(ana f)))
showWithMsg (LocalDataAnalysis ana) showresult =
   LocalDataAnalysis
     (\tds f -> MsgResult (showresult(ana tds f)))
showWithMsg (GlobalAnalysis ana) ...
showWithMsg (GlobalDataAnalysis ana) ...
```

Based on this conversion operation (which is also defined as an infix operator), it is easy to integrate an analysis like `isOverlappingFunction` into CurryBrowser by adding an element to the configuration list `functionAnalyses` as follows:

```
functionAnalyses =
 [...,
  ("Overlapping rules",
   LocalAnalysis isOverlappingFunction
                 'showWithMsg' showOverlap),
  ...]
```

Similarly to the list constant `functionAnalyses`, the configuration module of CurryBrowser contains two further list constants that specify the available analyses:

- a list constant `allFunctionAnalyses` that contains analyses that are applied to *all* functions of the selected module (e.g., applying the analysis "Pattern completeness" to all functions of a module is useful to spot those functions with incomplete pattern definitions): the results of these analyses are shown as prefixes in the column showing the list of all functions of the currently selected module;

- a list constant `moduleAnalyses` that contains analyses that are applied to complete modules (e.g., to generate the interface, the flat representation of a module, or a source code representation where missing type signatures are added); similarly to a function analysis, the result of a module analysis is either a (program) text to be shown in the main window or an I/O action that visualizes the result by an external tool.

Thus, it is easy to integrate existing tools (implemented in Curry) into CurryBrowser.

4. Available Analyses and Tools

This section shortly surveys the analyses and tools that are currently available in CurryBrowser. Due to the simple integration of further analyses and tools, this set is likely to be extended in future releases of CurryBrowser.

In the flat representation of Curry, pattern matching is made explicit by case expressions and disjunctions, and local definitions are "globalized" by lambda lifting [14]. Thus, it is sometimes interesting to show these transformations performed by the front end of a Curry implementation. For this purpose, CurryBrowser can show the flat representation of each function or module as well as a source-like representation where case expressions and disjunctions are translated back into pattern-based definitions but local definitions are globalized. Furthermore, the following analyses are available for individual functions:

`Calls directly` (local analysis): Shows all functions that are directly called by this function.

`Depends on` (global analysis): Shows all functions that might be directly or indirectly called in the rules defining this function.

`Dependency graph` (global analysis): Shows the dependency graph of the selected function. This is a combination as well as a graphical visualization of `Calls directly` and `Depends on`, i.e., an arc is drawn from each function symbol to all functions directly called in the rules defining this function

and all reachable function nodes are included in the graph (see Figure 2 for an example).

`Local dependency graph` (global analysis): Shows the dependency graph of the selected function restricted to all rules occurring in the current module. This is useful for complex functions, e.g., depending on other non-trivial library functions, where the complete dependency graph becomes unreadable due to its size.

`Called by` (global analysis): Shows the list of all functions in the current module that call this function in their defining rules.

`Overlapping rules` (local analysis): Shows whether the function is defined by overlapping rules (which might cause non-deterministic evaluations even for ground expressions). This is interesting for logic programming but might be also useful for purely functional programs.

`Right-linear rules` (local analysis): Shows whether the function is defined by right-linear rules, i.e., rules where each variable has at most one occurrence in the right-hand side.

`Right-linearity` (global analysis): Shows whether the function is defined by right-linear rules and depends only on functions defined by right-linear rules. This information is useful for some program optimizations (e.g., [3]).

`Pattern completeness` (local data analysis): Shows   whether the pattern matching is exhaustive, i.e., if the function is reducible on any combination of (well-typed) ground constructor argument terms.

`Solution completeness` (global analysis): Shows whether the function is operationally complete, i.e., if it is ensured that the execution of the function does not suspend for any arguments.

`Nondeterministic` (global analysis): Shows whether the function is possibly nondeterministic, i.e., if it directly or indirectly depends on an operation defined by overlapping rules so that it might deliver (nondeterministically) two values for the same ground constructor arguments.

`Set-valued` (global analysis): Shows whether the function is possibly set-valued, i.e., if it directly or indirectly depends on an operation defined by overlapping rules or rules containing extra variables (variables occurring in the right-hand side but not in the left-hand side) so that an application to some ground constructor arguments is equal to a set of more than one value.

`Purity` (global analysis): Shows whether the function is pure (referentially transparent), i.e., if it is ensured that it delivers always the same values for the same ground constructor arguments at each time and all schedulings of the evaluation. This might not be the case if committed choice or sending via ports is executed during its evaluation [7].

Finally, there are also useful tools to process complete modules or collections of modules. For instance, beyond showing the source code, interface, and flat representation of a module, one can also show a version of the source code where type signatures are added to functions where the programmer has omitted them.

To get more information about the import structure between modules, one can show all modules directly imported by the current one together with their exported functions that are accessed in the current module. This is useful to spot superfluously imported modules. Finally, one can also visualize the entire import relation between all modules of the currently loaded application as a module dependency graph (see Figure 3 for an example).

5. Conclusions

We have presented CurryBrowser, a generic analysis environment for Curry programs. CurryBrowser supports browsing through the modules of an application and offers a wide range of analysis tools in an integrated manner. The currently available set of analyses and tools can be easily extended due to the generic interface offered by the CurryBrowser implementation.

The mostly related system (and, in some sense, its predecessor) is CIDER [10], an environment to analyze single Curry modules. In contrast to CIDER, CurryBrowser can be applied to complete applications consisting of several modules and supports the browsing through the module structure. Furthermore, CurryBrowser provides a better structure to integrate analyzers: CIDER assumes that every analysis takes the complete program as input, whereas CurryBrowser distinguishes between different kinds of analyses (local, global, data-dependent) and provides them with the appropriate information from the modules and functions selected by the user.

Another related system is $\mathcal{IDE}$ [5], a graphical development environment for the functional logic languages Toy and Curry. $\mathcal{IDE}$ supports the writing of programs in a standard text editor window and the compilation and execution of programs. However, $\mathcal{IDE}$ does not offer further tools, e.g., for program analysis.

CurryBrowser is completely implemented in Curry. The advanced programming techniques offered by Curry (e.g., higher-order functions, demand-driven evaluation, meta-programming, high-level abstractions with logic variables for GUI programming [8]) has supported the fast and maintainable implementation of this environment. The size of the complete implementation of CurryBrowser is approximately 1400 lines of Curry code. This includes the implementation of the graphical user interface and all currently available analyses and tools. In addition, the total size of all imported system libraries is approximately 2500 lines of Curry code. These numbers provides an indication of the advantages obtained by the use of declarative high-level programming languages for the implementation of complex systems.

The implementation of CurryBrowser is freely available with the latest distribution of PAKCS [9] where it has been integrated for easy use.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 795–829, 2005.

[2] S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pp. 171–185. Springer LNCS 1794, 2000.

[3] 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.

[4] S. Antoy, M. Hanus, B. Massey, and F. Steiner. An Implementation of Narrowing Strategies. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 207–217. ACM Press, 2001.

[5] J. de Dios Castro and J.C. González Moreno. A Graphical Development Environment for Functional Logic Languages. In *Proc. of the Ninth International Workshop on Functional and Logic Programming (WFLP 2000)*, pp. 404–417. Universidad Politécnica de Valencia, 2000.

[6] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.

[7] 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)*, pp. 376–395. Springer LNCS 1702, 1999.

[8] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *Intl. Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
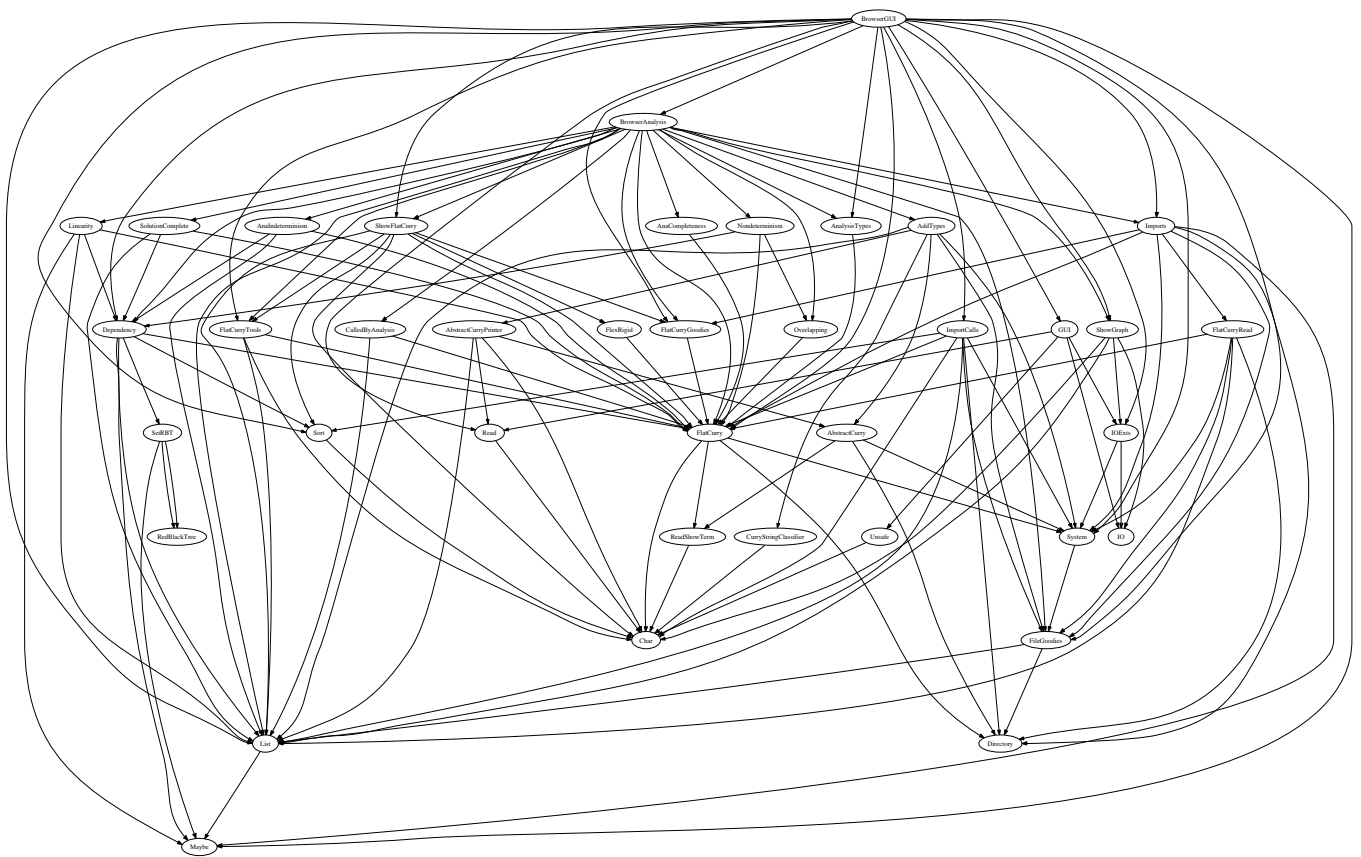
Figure 3. Module dependency graph of the CurryBrowser implementation (visualized by CurryBrowser)

[9] M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/˜pakcs/`, 2005.

[10] M. Hanus and J. Koj. An Integrated Development Environment for Declarative Multi-Paradigm Programming. In *Proc. of the International Workshop on Logic Programming Environments (WLPE'01)*, pp. 1–14, Paphos (Cyprus), 2001. Also available from the Computing Research Repository (CoRR) at `http://arXiv.org/abs/cs.PL/0111039`.

[11] M. Hanus and F. Steiner. Type-based Nondeterminism Checking in Functional Logic Programs. In *Proc. of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pp. 202–213. ACM Press, 2000.

[12] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/˜curry`, 2003.

[13] T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pp. 216–232. Springer LNCS 2024, 2001.

[14] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Functions. In *Functional Programming Languages and Computer Architecture*, pp. 190–203. Springer LNCS 201, 1985.

[15] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[16] P. Wadler. Efficient Compilation of Pattern-Matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pp. 78–103. Prentice Hall, 1987.