# CIDER: An Integrated Development Environment for Curry

Michael Hanus[1]* Johannes Koj[2]

[1] Institut für Informatik, Christian-Albrechts-Universität Kiel
D-24098 Kiel, Germany, mh@informatik.uni-kiel.de
[2] Lehrstuhl für Informatik II, RWTH Aachen, Germany, johannes.koj@sdm.de

**Abstract.** In this system demonstration we present CIDER (Curry Integrated Development EnviRonment), an analysis and programming environment for the declarative multi-paradigm language Curry. CIDER is a graphical environment to support the development of Curry programs by providing integrated tools for the analysis and visualization of programs. CIDER is completely implemented in Curry using libraries for GUI programming (based on Tcl/Tk) and meta-programming. An important aspect is the extensibility of the development environment w.r.t. new analysis methods, since a new program analysis can be easily added to our environment. The lazy evaluation strategy of the underlying implementation language Curry becomes quite useful for this feature.

## 1 Overview

CIDER is a graphical programming and development environment for the declarative multi-paradigm language Curry [2]. Since the development of analysis and debugging tools for declarative languages is still an ongoing research, CIDER is not designed as a closed system but it is intended as an open platform to integrate various tools for analyzing and debugging Curry programs. Currently, CIDER consists of

- a program editor with the usual functionality,
- various tools for analyzing properties of functions in Curry programs (types, overlapping definitions, complete definitions, dependencies etc),
- a graphical debugger, i.e., a visualization of the evaluation of expressions,
- a tool for drawing dependency graphs.

CIDER is completely implemented in Curry so that it is fairly easy to extend CIDER by new analysis tools: a new analysis can be integrated by implementing a function with type

```
Prog -> [(String,AnaRes)]
```

which takes a program as input (`Prog`) and produces a list of analysis results, where each analysis result is a pair consisting of a function name and the associated result for analyzing this function (currently, a string or a graph). If an

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD under the PROCOPE programme.

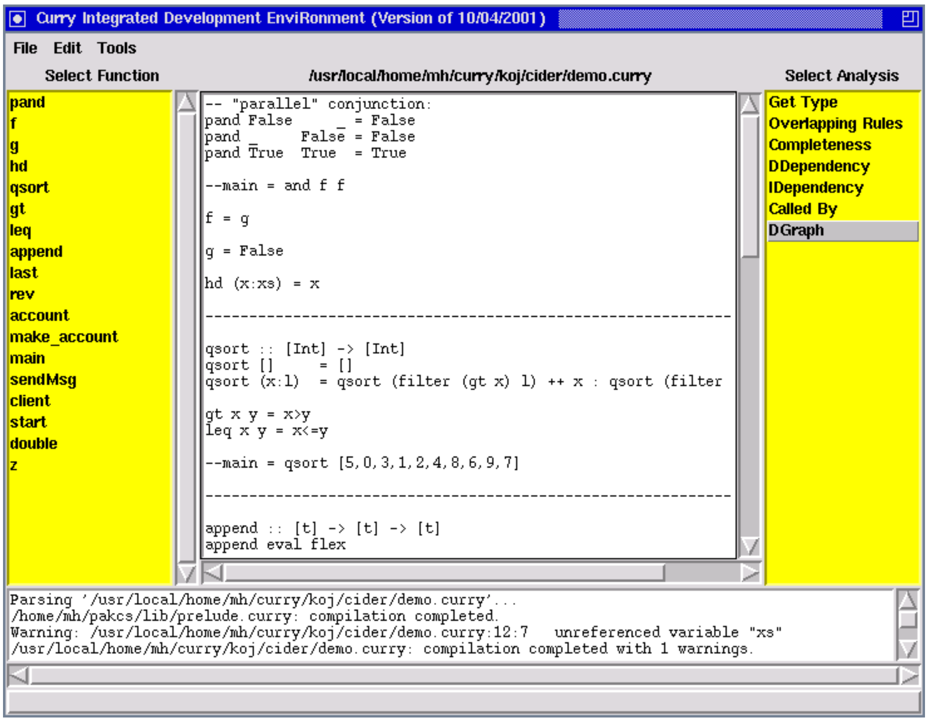


**Fig. 1.** The main window of CIDER

analysis is implemented in this way, only one line of code must be added in the implementation of CIDER to provide a graphical user interface (GUI) for this new analysis. Note that Curry's lazy evaluation is quite handy here: although an analysis is defined on an entire program, only those parts are actually analyzed that are relevant for showing the currently requested result.

To get an impression of the use of CIDER, Fig. 1 shows a snapshot window after starting CIDER and loading a program. The main window in the middle is an editor window for the current program. On the left- and right-hand side, there is a list of the top-level functions in the current file and a list of the currently available analysis tools, respectively. After selecting a function and an analysis in the corresponding list boxes, the function is analyzed and the analysis result is either shown in the bottom window (if it is a textual result) or, if it is a graph, it is visualized with the graph visualization tool daVinci[1]. Currently, CIDER contains the following analysis tools (which are not very difficult and mainly included for demonstration issues):

`Get Type`: Compute the function's type.

`Overlapping Rules`: Is the function defined by overlapping rules that might cause non-deterministic evaluations even for ground expressions?

[1] `http://www.tzi.de/daVinci/`

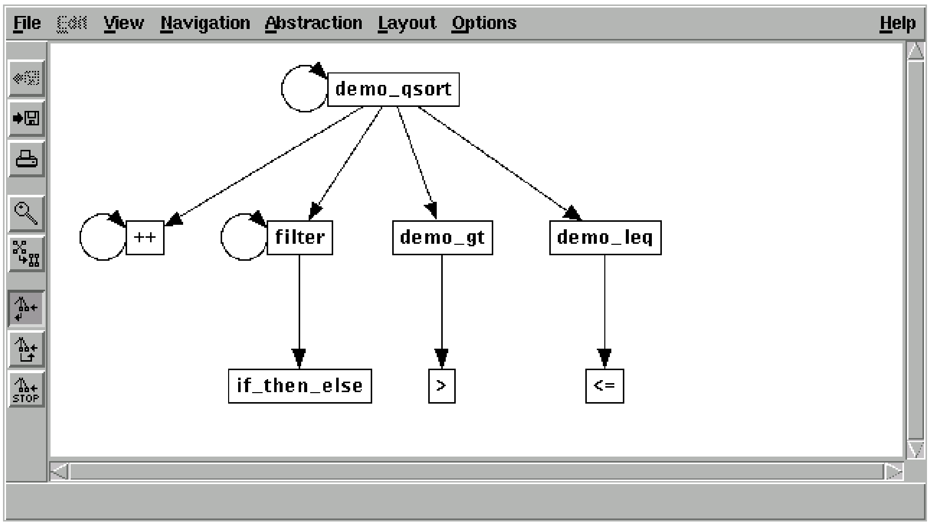


**Fig. 2.** Visualization of a program dependency graph

`Completeness`: Is the function completely defined, i.e., reducible on all ground constructor terms? Due to possible overlapping rules, the current implementation is based only on a sufficient criterion, i.e., the analysis results are "complete" or "might be incomplete".

`(D/I)Dependency`: Direct/indirect dependency, i.e., all functions that are directly or indirectly called in the rules defining this function.

`Called By`: Compute the list of all functions that call this function in their defining rules.

`DGraph`: Show the dependency graph for this function.

For instance, the visualization computed by the analysis `DGraph` for the function `qsort` in the program `demo.curry` is shown in Fig. 2.

Finally, CIDER contains also a *graphical debugger/tracer* to visualize the evaluation of expressions. The debugger always shows the expressions as trees although some parts of the expressions are actually shared. If a subexpression is reduced, all identical subexpressions shared with this subexpression are also reduced in the same step in order to be conform with Curry's operational semantics (which is based on sharing to support laziness, cf. [1]). The subexpression reduced in the next step (the next `redex`) is always emphasized in red (similarly, a variable is emphasized in red if it will be bound in the next step). This is quite useful to visualize the execution of concurrent computations which are synchronized in Curry by the instantiation of logical variables, as in concurrent constraint languages [7]. One can trace forward and backward through all evaluation steps. Furthermore, one can also set a breakpoint to skip larger parts of a computation. A snapshot of the debugger is shown in the Fig. 3.

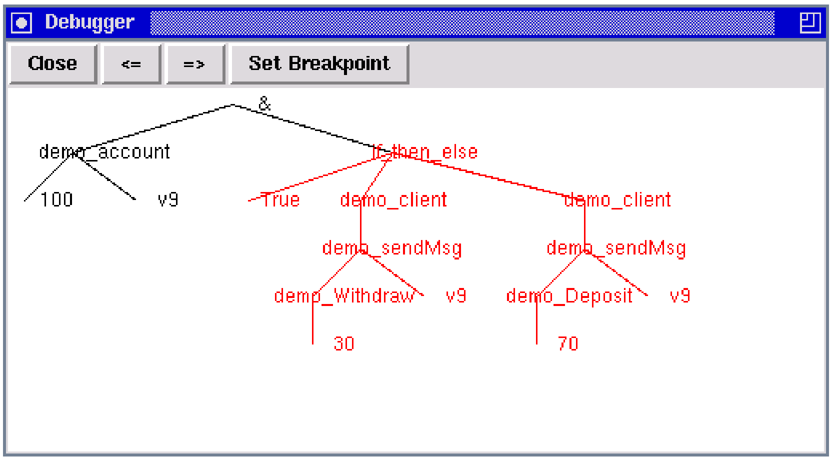


**Fig. 3.** Visualization of a (concurrent) computation

## 2 Implementation

As mentioned above, CIDER is completely implemented in Curry using various libraries available in the PAKCS implementation [4]. For instance, in order to analyze programs, a representation of Curry programs as algebraic datatypes is used which is provided by the library `Flat` for meta-programming in Curry[2]. The graphical user interface is implemented with the library `Tk` which supports a high-level implementation of GUIs in Curry by exploiting the integrated functional and logic features of the language [3]. Furthermore, the handling of internal states, which is necessary in complex GUI applications, is modeled in an object-oriented manner following the ideas of object-oriented functional logic programming proposed in [5].

The code size of the complete implementation of CIDER is approximately 1400 lines of Curry code. This includes the implementation of the graphical user interface, the various analysis tools, and the meta-interpreter and graphical debugger. In addition, the total size of all imported system libraries is approximately 1500 lines of Curry code. These numbers indicate the advantage of the use of declarative high-level programming languages in the implementation of complex systems.

The implementation has one configuration module which contains the definition of a constant

```
anaList :: [(String, Prog->[(String,AnaRes)])]
```

This constant specifies the list of all currently available analyses, i.e., the first component of each entry is the name of the analysis (shown in the right column of the main window) and the second component is the implementation of the

[2] `http://www.informatik.uni-kiel.de/~curry/flat/`

analysis as a function as described above. Thus, in order to integrate a new analysis into CIDER, one only needs to add a new element to this list and recompile the CIDER system.

The implementation of CIDER is freely available from the web page `http://www.informatik.uni-kiel.de/~pakcs/cider/`. Details about the design and implementation of CIDER can be found in [6].

## References


1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
2. 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.
3. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
4. M. Hanus, S. Antoy, 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/`, 2000.
5. M. Hanus, F. Huch, and P. Niederau. An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry. In *Proc. 12th Int. Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 89–106. Springer LNCS 2011, 2001.
6. J. Koj. A graphical programming environment for declarative programming languages (in german). Master's thesis, RWTH Aachen, 2000.
7. V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.