An Integrated Development Environment for Declarative Multi-Paradigm Programming*

Michael Hanus† CAU Kiel

Johannes Koj‡ RWTH Aachen

Abstract

In this paper 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 of our environment is the possible adaptation of the development environment to other declarative source languages (e.g., Prolog or Haskell) and the extensibility w.r.t. new analysis methods. To support the latter feature, the lazy evaluation strategy of the underlying implementation language Curry becomes quite useful.

1 Overview

CIDER is a graphical programming and development environment for the construction and debugging of declarative multi-paradigm programs. Although the current implementation of CIDER is targeted at the multi-paradigm programming language Curry [10], the intension is to provide a development platform for both functional and logic languages since Curry integrates the most important 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 expressions with synchronization on logical variables). In particular,

*In A. Kusalik (ed), Proceedings of the Eleventh Workshop on Logic Programming Environments (WLPE'01), December 1, 2001, Paphos, Cyprus. COmputer Research Repository (http://www.acm.org/corr/), cs.PL/0111039; whole proceedings: cs.PL/0111042. 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.

†Institut für Informatik, Christian-Albrechts-Universität Kiel, Olshausenstr. 40, D-24098 Kiel, Germany, mh@informatik.uni-kiel.de

‡Lehrstuhl für Informatik II, RWTH Aachen, Germany, johannes.koj@sdm.de

the implementation of CIDER is based on an intermediate language to which functional, logic, and also integrated functional logic programs can be compiled (e.g., see [1, 5, 6, 17]). Thus, CIDER can be adapted to other declarative languages provided that there exists a front end to compile programs into this implementation-independent format (there exists also an XML representation for this intermediate language, see [6]).

CIDER is an environment where various analysis and debugging tools for declarative multi-paradigm languages are available. Since the development of such tools 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 programs. Currently, CIDER consists of

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

To get an impression of the use of CIDER, Fig. 1 shows a snapshot 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 (see below for a description), 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]. The current version of CIDER contains the following analysis tools (which are useful but very simple and mainly included for demonstration issues; see also Section 4 for a description on how to add new analysis tools):

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

`Overlapping Rules:` 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.

`Completeness:` Shows whether 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".

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

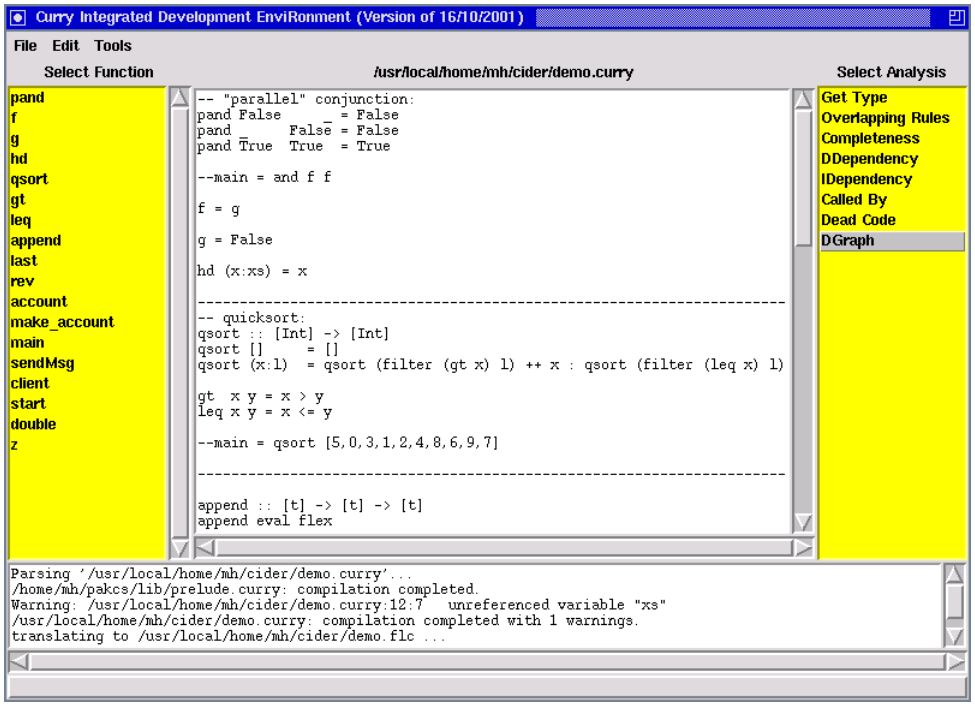

Figure 1: The main window of CIDER

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

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

`Dead Code`**:** Computes the list of all top-level functions in the currently loaded module that are not reachable from the selected function.

`DGraph`**:** Shows the dependency graph for the selected function. This is a mixture as well as a graphical visualization of `(D/I)Dependency`, 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.

For instance, the visualization computed by the analysis `DGraph` for the function `qsort` (compare Fig. 1) is shown in Fig. 2.

Finally, CIDER contains also a graphical debugger/tracer to visualize the evaluation of expressions. Due to the fact that the operational semantics of the considered language is important for this part of the programming environment, we discuss it in Section 3.4.

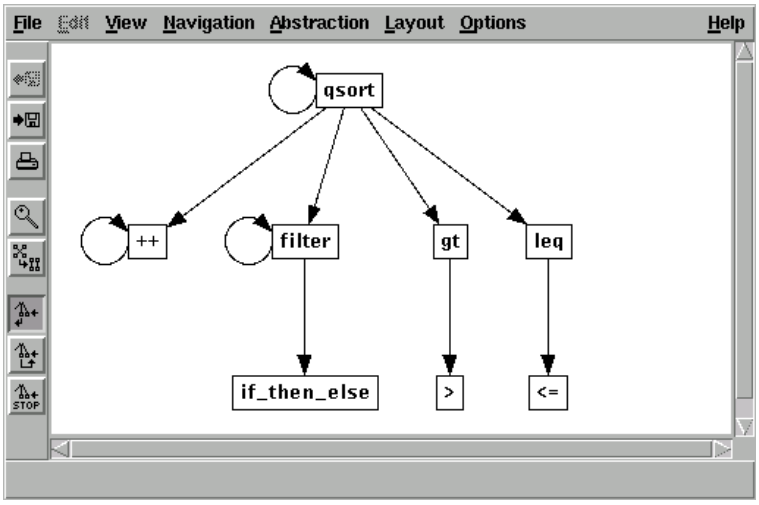

Figure 2: Visualization of a program dependency graph

The rest of this paper is structured as follows. The next section provides a short overview of the main features of Curry as relevant for this paper. Section 3 surveys the implementation of our programming environment. Section 4 sketches the necessary tasks to add new analysis tools to CIDER. Finally, Section 5 contains our conclusions.

2 Basic Elements of Curry

Although we mentioned above that our programming environment can be adapted to other declarative languages than Curry, the main motivation for the development of CIDER (and also Curry itself) is to provide a common platform for declarative programming where the most important declarative paradigms are smoothly integrated. Therefore, Curry is our main target language and we review in this section those elements of Curry which 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 [10, 16].

Curry is a modern multi-paradigm declarative 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 [21], 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 f to 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 evaluated in a lazy manner. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. The behavior of function calls with free variables depends on the evaluation annotations of functions which can be either *flexible* or *rigid*. Calls to rigid functions are suspended if a demanded argument, i.e., an argument whose value is necessary to decide the applicability of a rule, is uninstantiated ("*residuation*"). Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments to the required values in order to apply a rule ("*narrowing*").

Example 1 The following Curry program defines the data types of Boolean values and polymorphic lists (first two lines) and functions for computing the concatenation of lists and the last element of a list:

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

conc :: [a] -> [a] -> [a]
conc eval flex       -- specify evaluation mode of conc as "flexible"

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.[2] Since `conc` is explicitly defined as flexible[3] (by "`eval flex`"), the equation "`conc ys [x] =:= xs`" can be solved by instantiating the first argument `ys` to the list `xs` without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

In general, functions are defined by (*conditional*) *rules* of the form "l `|` c `=` e `where` vs `free`" where l has the form $f\,t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, the *condition* c is a

[2] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[3] As a default, all functions except for constraints are rigid.

constraint, e is a well-formed *expression* which may also contain function calls, lambda abstractions etc, and vs is the list of *free variables* that occur in c and e but not in l (the condition and the `where` parts can be omitted if c and vs are empty, respectively). The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable. A *constraint* is any expression of the built-in type `Success`. Each Curry system provides at least equational constraints of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms (i.e., terms without defined function symbols). However, specific Curry systems can also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [13].

Concurrent programming is supported by the concurrent conjunction operator "`&`" on constraints, i.e., a non-primitive constraint of the form "c_1 `&` c_2" is evaluated by solving both constraints c_1 and c_2 concurrently. Since suspension is controlled by the instantiation of arguments in calls to rigid functions, concurrent computations are synchronized in a high-level manner by logic variables as in concurrent constraint programming [22].

The operational semantics of Curry, precisely described in [10, 16], is a conservative extension of lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Since it is based on an optimal evaluation strategy [4], Curry can be considered as a generalization of concurrent constraint programming [22] with a lazy (optimal) evaluation strategy. Due to this generalization, Curry supports a clear separation between the sequential (functional) parts of a program, which are evaluated with an efficient and optimal evaluation strategy, and the concurrent parts, based on the concurrent evaluation of constraints, to coordinate concurrent program units.

The concurrent conjunction operator "`&`" is a basic combinator to create a fixed network of concurrent activities. However, this primitive is too limited when a dynamically varying number of processes with many-to-one communication structures should be modeled, since this requires the merging of message streams from different processes into a single message stream. Doing that with a merger function causes a set of problems as discussed in [11, 18]. Therefore, Janson et al. [18] proposed the use of ports for the concurrent logic language AKL which are generalized in [11] to support distributed programming in Curry. In principle, a *port* is a constraint between a multiset and a stream which is satisfied if the multiset and the stream contain the same elements (messages). In Curry a port is created by a constraint "`openPort p s`" where `p` and `s` are free logical variables. This constraint creates a multiset and a stream and combines them over a port. Elements can be inserted into the multiset by sending them to `p` by the constraint "`send m p`". When a message is sent to `p`, it will automatically be added to the stream `s` in order to satisfy the port constraint. To support the implementation of distributed systems, where the processes run on different machines, ports can be also made externally accessible by assigning a

symbolic name to them. For instance, the I/O action[4] `(openNamedPort "`*name*`")` opens an external port with symbolic name *name* and returns the (infinite and lazy) stream of incoming messages. If this port has been opened on machine m, clients can access this port by executing the I/O action `(connectPort "`*name*`@`*m*`")`. This returns a port p to which they can send their messages. Note that messages can also contain logical variables which provides for a high-level mechanism to return values by instantiation (rather than creating reply channels, see [11]). The port concept has been used to integrate object-oriented features into Curry [14] which are often necessary in GUI (Graphical User Interface) programming to keep the state of user interfaces [12].

3 Implementation

In this section we provide an overview on the implementation of CIDER. This can be also seen as an example to show that high-level declarative languages provide the appropriate abstractions to implement such advanced application in a modular and extensible way. More details about the design and implementation of CIDER can be found in [19].

3.1 Intermediate Representation Language

In order to provide a high-level language for implementing new analysis tools to be integrated in our programming environment (see also Section 4), CIDER is completely implemented in Curry. In order to implement program analyzers, one needs a representation of programs as data objects. For representing functional logic programs, the direct representation of all program rules is not adequate since the particular pattern-matching strategy is quite important to reduce the search space and the length of derivations [4] and concrete languages often differ in this strategy. An appropriate data structure to describe such strategies are definitional trees, introduced in [3]. More recent approaches to the manipulation of functional logic programs (e.g., [1, 6, 17]) advocate the explicit representation of pattern matching by means of case constructs.

In the context of functional logic languages, it is necessary to distinguish two kinds of case expressions in order to specify the flexible/rigid status of functions. To be more precise, we assume that all functions are defined by one rule whose left-hand side contains only pairwise different variables as parameters and the right-hand side contains case expressions for pattern matching. Thus, the basic syntax of programs in this representation can be summarized as follows:

[4] See [23] for a description of the monadic I/O concept of Haskell which has been adapted without changes to Curry.

P	::=	$D_1 \ldots D_m$	e	::=	v	(variable)
D	::=	$f\ v_1\ \ldots\ v_n = e$		|	$c\ e_1\ \ldots\ e_n$	(constructor)
				|	$f\ e_1\ \ldots\ e_n$	(function call)
p	::=	$c\ v_1\ \ldots\ v_n$		|	*case* e_0 *of* $\{p_1 \to e_1; \ldots; p_n \to e_n\}$	(rigid case)
				|	*fcase* e_0 *of* $\{p_1 \to e_1; \ldots; p_n \to e_n\}$	(flexible case)
				|	e_1 *or* e_2	(disjunction)

where P denotes a program, D a function definition, p a pattern and e an arbitrary expression. A program P consists of a sequence of function definitions D such that the left-hand side has pairwise different variable arguments and the right-hand side is an expression e composed by variables, constructors, 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).

Example 2 *Consider the rules defining the (rigid) function "$\leqslant$":*

```
     0 ⩽ n       =  True
Succ m ⩽ 0       =  False
Succ m ⩽ Succ n  =  m ⩽ n
```

These rules can be represented by the following rule in our representation:

```
x ⩽ y = case x of {0        → True;
                   Succ x1  → case y of {0 → False;
                                         Succ y1 → x1 ⩽ y1} }
```

Based on this representation, some of the analyses discussed in Section 1, like the analysis for overlapping rules or completely defined functions, can be implemented in a straightforward manner by analyzing the structure of case expressions. In order to cover all features of functional logic languages, the definition of expressions can be extended by a few additional constructs, like higher-order applications ("*apply* $e_1\ e_2$"), partial applications, calls to external functions, and existential quantification of variables (e.g., see [2]). Programs in this language, which is also called *FlatCurry*[5], can be simply represented as data objects by a set of data type declarations similarly to Example 1. For instance, an entire program module is represented as an expression of the type

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

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

`[OpDecl]` `[Translation]`

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 and a table to map external into internal names and vice versa. Furthermore, a function declaration is represented as

```
data FuncDecl = Func String Int TypeExpr Rule
```

where the arguments are the name, arity, type, and rule (of the form "`Rule` *arguments expr*") of the function (here we omit the other data type declarations). The PAKCS implementation of Curry [13] provides a library `Flat` for meta-programming which contains the definition of such data types (also for representing data type declarations) and an I/O action for reading a program file and translating its contents into a `Prog` term.

The FlatCurry representation of programs has been used as an intermediate language to compile Curry [5, 6] or similar functional logic programs [17] and to optimize declarative programs by partial evaluation [1, 2]. However, it should be clear that this representation is not restricted to Curry. Purely functional programs can be translated into this intermediate language without the use of *fcase* and *or* constructs. Purely logic languages have only constraints as functions and do not use the *case* construct (although this could be used for the translation of logic languages with coroutining). For instance, the Prolog program

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

can be translated into the Curry program

```
app [] ys zs = (ys =:= zs)
app (x:xs) ys (z:zs) = (x =:= z) &> (app xs ys zs)
```

where "`&>`" denotes the sequential ("left-to-right") conjunction of constraints (replacing "`&>`" by the concurrent conjunction "`&`" corresponds to a logic program with coroutining). Note that the introduction of auxiliary variables and explicit equality constraints is necessary due to the left-linearity requirement in Curry. The latter rules can be translated into the following FlatCurry definition:

```
app xs ys zs =
        fcase xs of { []   → ys =:= zs;
                      x:x1 → fcase zs of
                               { z:z1 → (x =:= z) &> (app x1 ys z1) } }
```

3.2 Program Analysis

As shown in Section 1, our programming environment is intended to integrate various analysis tools for declarative programs. This demands for a unique interface for the implementation of each program analysis to be integrated into CIDER. Since the intermediate language FlatCurry is a reasonable basis for writing program analyzers, we require that each program analyzer must implement a function of type

```
type ProgAnalysis = Prog -> [(String,AnaRes)]
```

where

```
data AnaRes = Message String | Graph DvGraph
```

is the result type of analyzing an individual function in a program. Thus, a program analysis 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 of analyzing this function. In our current implementation, it is sufficient that this result is either a string s ("`Message` s", e.g., the type of a function, "overlapping"/"not overlapping", the list of called functions) to be shown in the bottom window of the main interface, or a graph g ("`Graph` g", e.g., the function's dependency graph) to be visualized in separate window with the graph visualization tool daVinci (see Fig. 2).

As a simple example, the analysis of overlapping rules can be implemented as follows:

```
analyseOverlappings :: Prog -> [(String,AnaRes)]
analyseOverlappings (Prog _ _ _ funs _ _) = map overlapFun funs
  where overlapFun (Func name _ _ (Rule _ e))
              | orInExpr e = (name, Message "overlapping")
              | otherwise  = (name, Message "not overlapping")
```

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

It is interesting to note that the lazy evaluation strategy of our implementation language Curry becomes quite handy here. Although a program analysis is defined as a function operating on the entire program, which may cause a complex computation, the lazy evaluation strategy performs the program analysis in a demand-driven manner. If a user selects an analysis (in the right window of the main interface, see Fig. 1), the corresponding analysis function is applied to the currently loaded program module. Due to lazy evaluation, this application is not evaluated at this analysis selection time but only if the user selects an individual function in the left window of the main interface. If such a function is selected, the analysis is performed to show its result. However, note that only the selected function is analyzed which can be done locally (e.g., "overlapping" analysis) or might require the consideration of other parts of the program (e.g., computing the dependency graph). Furthermore, if the same function is selected again, the result is directly

available due the fact that lazy evaluation evaluates each expression at most once. This behavior is very desirable for our environment. In a strict or imperative language, one needs some additional effort to implement such a behavior.

3.3 Main Interface

The main graphical user interface (see Fig. 1) 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 [12]. With this library, the structure of the interface (i.e., the different widgets) is described as a data term containing call-back functions for implementing the functionality of the individual widgets.

One difficulty in the implementation of user interfaces in a declarative language is the handling of internal states. This is necessary for keeping the name of the currently loaded program module, the selected program analysis, the selected function etc. GUI libraries for purely functional languages (e.g., [8]) advocates the use of monads for this purpose. Since our GUI library is not based on monads but exploits the functional and logic features of Curry, we handle the internal state of the GUI in an object-oriented style by exploiting Curry's port concept sketched in Section 2. Thus, the state of the main interface is kept as an argument of a recursive function implementing a "`GUIServer`". Basically, this function has type

```
serveGUI :: GUIServerState -> [GUIServerMsg] -> Success
```

where `GUIServerState` contains all data in the state of the main interface and `GUIServerMsg` are the messages to be processed by the server. This function is implemented by a case distinction on the different first messages where the function is (tail recursively) called with a modified state (depending on the message) and the list of remaining messages. For instance, the message `Terminate` terminates the server and the message "`SetAna` a" sets the current program analysis. Thus, we have the following defining rules, among others (the function `changeAna` computes a new state with a changed current analysis component):

```
serveGUI _     (Terminate:_)     = success --the end, no recursive call
serveGUI state (SetAna a : msgs) = serveGUI (changeAna a state) msgs
```

Now, the main interface is implemented as a GUI (based on the `Tk` library as discussed above) where the call-back functions only send messages to the GUI server. For instance, if the user selects an analysis a, the message "`SetAna` a" is sent. The GUI server is also responsible to compute the analysis results when they are requested by the GUI. More details about this technique to keep states in GUIs can be found in [12]. Moreover, [14] contains a general description of the object-oriented functional logic programming style applied here.

It is interesting to note that the separation of the implementation into the GUI itself

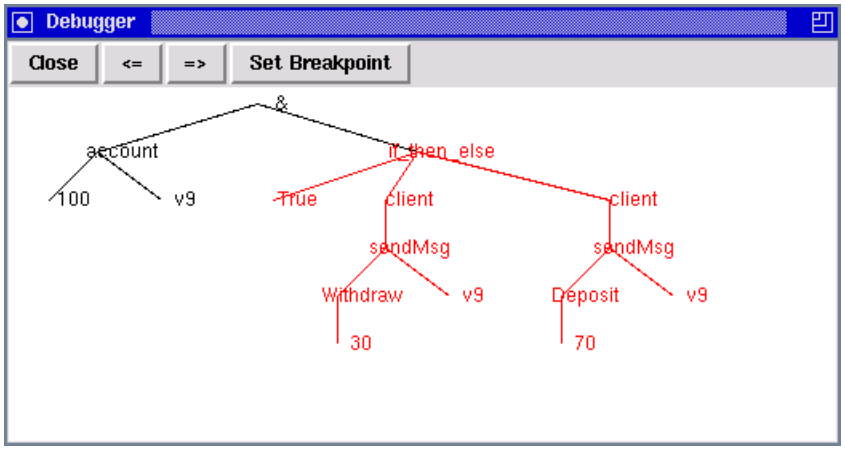

Figure 3: Visualization of a (concurrent) computation

and the GUI server provides also the possibility to move the computation-intensive tasks to a powerful "compute server" with only minor changes in our implementation. Since a port can be also made externally accessible (by changing the method to create a port, i.e., replacing `openPort` by `openNamedPort`, see Section 2), we can start the GUI server on another machine than the GUI itself. This requires only a change of two program lines (replacing internal by external ports) in our implementation and might be reasonable for more complex program analyses.

3.4 Graphical Tracer

As mentioned at the beginning, CIDER also contains 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. [4]). The subexpression reduced in the next step (the next `redex`) is always colored in red. Similarly, a variable is colored 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 (compare Section 2). One can trace forward and backward through all evaluation steps. Furthermore, one can also set a breakpoint to skip uninteresting parts of a computation. A snapshot of the debugger is shown in the Fig. 3.

The debugger is implemented as a meta-interpreter for (Flat)Curry in Curry, i.e., it is based on a "step function" that maps a single (FlatCurry) expression into a list of

expressions according to the operational semantics of Curry as defined in [10, 16]. The resulting list of expressions collects all "don't know" alternatives of a single evaluation step, e.g., for purely functional programs this list contains at most one element. Since the debugger works on the level of FlatCurry expressions, it can visualize all kinds of computations including concurrency, higher-order functions, etc. As a drawback, it has only a limited performance but, nevertheless, it is useful to explain and understand the computation model of Curry. Thus, the debugger is intended as a teaching tool to visualize the operational semantics of Curry rather than a tool to locate bugs in larger programs. For the latter purpose, it might be interesting to develop more efficient debuggers as done for purely lazy functional languages (e.g., [7]). The visualization part of our debugger is also implemented in Curry by the use of the `Tk` library.

4 Extending the Development Environment

This section discusses how one can extend the current development environment in various directions. We already mentioned that CIDER should integrate various analysis tools for declarative programs. In order to add a new analysis, one has to implement it as a function of type `ProgAnalysis` (see Section 3.2). For the simple addition of new analysis tools, our implementation has a configuration module which contains the definition of a constant

```
anaList :: [(String,ProgAnalysis)]
```

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 analysis as a function as described above. For instance, the current definition of `anaList` contains an element

```
("Overlapping Rules",analyseOverlappings)
```

(compare Section 3.2). Therefore, in order to integrate a new analysis into CIDER, one only needs to add a new element to this list, recompile the CIDER system, and the new analysis is easily accessible through the graphical interface.

Another possible extension of CIDER is the language of programs to be loaded and analyzed. As explained above, CIDER is based on the FlatCurry representation of programs which can be also used for other source languages (e.g., Prolog, Haskell, or Toy [20]). Since the program editor in the main window is just a standard text editor, CIDER is largely independent of the concrete syntax of the source language. Thus, in order to adapt CIDER to another source language $\mathcal{X}$, one only needs to replace the Curry front end (which is used in the GUI server to load source programs) by another front end that translates $\mathcal{X}$ programs into the corresponding FlatCurry representation.[6] Thus, the configuration

[6] If the source language $\mathcal{X}$ also requires a different operational semantics of FlatCurry, then the implementation of the tracer (see Section 3.4) must be changed, too.

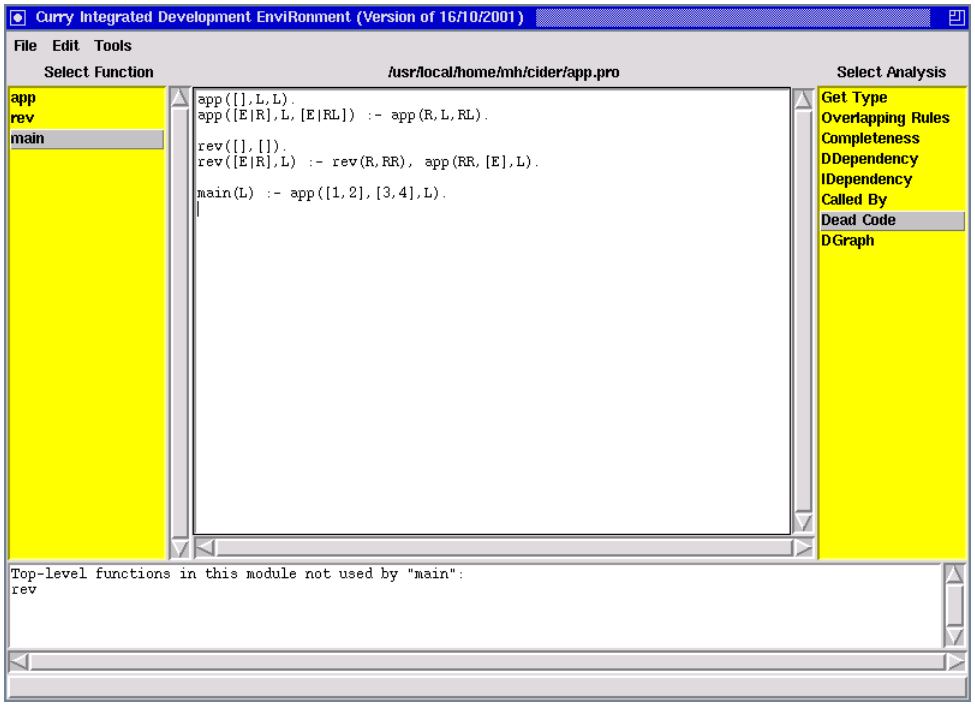

Figure 4: CIDER adapted to the source language Prolog

module of CIDER contains also the definition of the actual preprocessor that translates source programs into the corresponding FlatCurry representation. For instance, we have implemented such a preprocessor for pure Prolog programs and Fig. 4 shows the main window of CIDER after loading and analyzing a Prolog program with this preprocessor.

5 Conclusions

We have presented CIDER, a graphical programming and development environment for declarative multi-paradigm programs. The main motivation of CIDER is to support the construction and debugging of programs by offering a wide range of analysis tools in an integrated manner. Although the current implementation offers only a few basic analysis tools and is targeted at the language Curry, we have also shown that it is fairly easy to extend CIDER with new analysis tools or adapt it to a new declarative source language. As far as we know, CIDER is the first development environment for declarative multi-paradigm programs designed to integrate various analysis tools. The mostly related system is $\mathcal{IDE}$ [9], 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 tools for analyzing and debugging programs.

CIDER is completely implemented in Curry and we have discussed how advanced programming techniques are exploited in this implementation. For instance, functions as first-class citizens are useful to apply higher-order programming techniques in the implementation of analysis tools or to integrate such tools as functions in data structures. Laziness is practical to define a program analysis in a conceptual clean manner but compute only those parts that are actually required by the user. Furthermore, concurrent programming is useful to structure and distribute the various tasks possibly on different machines.

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 tracer. 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 of CIDER is freely available from the web page `http://www.informatik.uni-kiel.de/~pakcs/cider/` and requires only an installed Curry system with the libraries for application programming distributed with the PAKCS implementation [13].

For future work we intend to add more analysis tools, in particular, advanced type-based analysis tools which can be used to analyze particular properties of multi-paradigm programs [15]. Furthermore, better (declarative) debugging tools should be integrated into CIDER as well as the direct efficient execution of programs by connecting compilers based on FlatCurry, like [6].

References

[1] 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)*, pp. 381–398. Springer LNCS 1955, 2000.

[2] 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)*, pp. 326–342. Springer LNCS 2024, 2001.

[3] S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.

[4] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.

[5] 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.

[6] 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.

[7] O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood – A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 176–193. Springer LNCS 2011, 2001.

[8] K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *Proc. of the International Conference on Functional Programming (ICFP'97)*, pp. 251–262. ACM SIGPLAN Notices Vol. 32, No. 8, 1997.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] M. Hanus, F. Huch, and P. Niederau. An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 89–106. Springer LNCS 2011, 2001.

[15] 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.

[16] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.

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

[18] S. Janson, J. Montelius, and S. Haridi. Ports for Objects in Concurrent Logic Programs. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[19] J. Koj. A graphical programming environment for declarative programming languages (in german). Master's thesis, RWTH Aachen, 2000.

[20] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.

[21] J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.

[22] V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[23] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.