A Modular and Generic Analysis Server System for Functional Logic Programs

Michael Hanus and Fabian Skrlac

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh|fre}@informatik.uni-kiel.de

November 15, 2013

Abstract. We present the design, implementation, and application of a system, called CASS, for the analysis of functional logic programs. The system is generic so that various kinds of analyses (e.g., groundness, non-determinism, demanded arguments) can be easily integrated. In order to analyze larger applications consisting of dozens or hundreds of modules, CASS supports a modular and incremental analysis of programs. Moreover, it can be used by different programming tools, like documentation generators, analysis environments, program optimizers, as well as Eclipse-based development environments. For this purpose, CASS can also be invoked as a server system to get a language-independent access to its functionality. CASS is completely implemented in the functional logic language Curry as a master/worker architecture to exploit parallel or distributed execution environments.

1 Introduction

Automated program analyses are useful for various purposes. For instance, compilers can benefit from their results to improve the translation of source into target programs. Analysis information can be helpful to programmers to reason about the behavior and operational properties of their programs. Moreover, this information can also be documented by program documentation tools or interactively shown to developers in dedicated programming environments. On the one hand, declarative programming languages provide interesting opportunities to analyze programs. On the other hand, their complex or abstract execution model demands good tool support to develop reliable programs. Examples are the detection of type errors in languages with higher-order features and the detection of mode problems in the use of Prolog predicates.

This work is related to functional logic languages that combine the most important features of functional and logic programming in a single language (see [7,19] for recent surveys). In particular, these languages provide higher-order functions and demand-driven evaluation from functional programming together with logic programming features like non-deterministic search and computing with partial information (logic variables). This combination has led to new design patterns [5,8] and better abstractions for application programming. Moreover, program implementation and analysis aspects for functional as well as logic languages can be considered in a unified framework. For instance, test cases for functional programs can be generated by executing functions with logic variables as arguments [13].

Automated program analyses have already been used for functional logic programming in various situations. For instance, CurryDoc [16] is an automatic documentation tool for the functional logic language Curry that analyzes Curry programs to document various operational aspects, like the non-determinism behavior or completeness issues. CurryBrowser [17] is an interactive analysis environment that unifies various program analyses in order to reason about Curry applications. KiCS2 [10], a recent implementation of Curry that compiles into Haskell, includes an analyzer to classify higher-order and deterministic operations in order to support their efficient implementation which results in highly efficient target programs. Similar ideas are applied in the implementation of Mercury [33] which uses mode and determinism information to reorder predicate calls. Non-determinism information as well as information about definitely demanded arguments has been used to improve the efficiency of functional logic programs with flexible search strategies [18]. A recent Eclipse-based development environment for Curry [29] also supports the access to analysis information during interactive program development.

These kinds of program analyses and their different implementations demand a unifying framework. This is the motivation for the current work. We present CASS (Curry Analysis Server System) which is intended to be a central component of current and future tools for functional logic programs. CASS provides a generic interface to support the integration of various program analyses. Although the current implementation is strongly related to Curry, CASS can also be used for similar declarative programming languages, like TOY [27]. The analyses are performed on an intermediate format into which source programs can be compiled. CASS supports the analysis of larger applications by a modular and incremental analysis. The analysis results for each module are persistently stored and recomputed only if it is necessary. Since CASS is implemented in Curry, it can be used as a library in tools implemented in Curry, like the documentation generator CurryDoc, the analysis environment CurryBrowser, or the Curry compiler KiCS2. CASS can also be invoked as a server system providing a text-based communication protocol in order to interact with tools implemented in other languages, like the Eclipse plug-in for Curry. CASS is implemented as a master/worker architecture, i.e., it can distribute the analysis work to different processes in order to exploit parallel or distributed execution environments.

In the next section, we review some features of Curry. Section 3 discusses the basic ideas of our analysis framework and shows how various kinds of program analyses can be implemented and integrated into CASS. Some uses of CASS are presented in Section 4 before its implementation is sketched in Section 5 and evaluated in Section 6. Section 7 concludes with a discussion of related work.

2 Curry and FlatCurry

In this section we review some aspects of Curry that are necessary to understand the functionality and implementation of our analysis tool. More details about Curry's computation model and a complete description of all language features can be found in [15,23].

Curry is a declarative multi-paradigm language combining in a seamless way features from functional, logic, and concurrent programming. Curry has a Haskell-like syntax[1] [30] extended by the possible inclusion of free (logic) variables in conditions and right-hand sides of defining rules. Curry also offers standard features of functional languages, like polymorphic types, modules, or monadic I/O which is identical to Haskell's I/O concept [34]. Thus, "`IO` α" denotes the type of an I/O action that returns values of type α.

A *Curry program* consists of the definition of *functions* or *operations* 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. As an example, consider the following program:

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

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

last xs | _ ++ [x] =:= xs  = x  where x free
```

The data type declarations define `True` and `False` as Boolean values 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 list concatenation operation `(++)` specifies that `(++)` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[2] The definition of the operation `last` demonstrates the logic programming features of Curry: the last element `x` of a list `xs` is computed by solving the equational constraint "`_ ++ [x] =:= xs`". Note that, in contrast to Prolog, logic (free) variables must be explicitly declared by "`free`" (except for anonymous variables denoted by "`_`").

The operational semantics of Curry [1,15] is a conservative extension of lazy functional programming (if free variables do not 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

[1] 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 ("$f\ e$").

[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.

representation of Curry programs has been shown to be useful. Programs of this intermediate language, called FlatCurry, contain a single rule for each function where the pattern matching strategy is represented by case expressions. The basic structure of FlatCurry is defined as follows:[3]

$$
\begin{array}{lll}
\tau & ::= \alpha & \text{(type variable)} \\
 & \mid\ T\ \tau_1 \ldots \tau_n & \text{(type constructor application)} \\
 & \mid\ \tau_1 \to \tau_2 & \text{(functional type)}
\end{array}
$$

$$
\begin{array}{lll}
P & ::= D_1 \ldots D_m & \\
D & ::= f :: \tau & \\
 & \quad\ f\ x_1\ \ldots\ x_n = e & \\
p & ::= c\ x_1\ \ldots\ x_n & \\
e & ::= x & \text{(variable)} \\
 & \mid\ c\ e_1\ \ldots\ e_n & \text{(constructor application)} \\
 & \mid\ f\ e_1\ \ldots\ e_n & \text{(function application)} \\
 & \mid\ \mathit{case}\ e_0\ \mathit{of}\ \{\overline{p_k \to e_k}\} & \text{(rigid case distinction)} \\
 & \mid\ \mathit{fcase}\ e_0\ \mathit{of}\ \{\overline{p_k \to e_k}\} & \text{(flexible case distinction)} \\
 & \mid\ e_1\ \mathit{or}\ e_2 & \text{(non-deterministic choice)} \\
 & \mid\ \mathit{let}\ \overline{x_k}\ \mathit{free\ in}\ e & \text{(free variable introduction)}
\end{array}
$$

A type expression τ is a type variable, a constructed type (or a base type if $n = 0$), or a functional type. 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 function calls, case expressions, disjunctions, and introduction of free (unbound) variables. A case expression has the form $(f)\mathit{case}\ e\ \mathit{of}\ \{c_1\ \overline{x_{n_1}} \to e_1, \ldots, c_k\ \overline{x_{n_k}} \to e_k\}$, 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* non-deterministically binds this variable to the pattern in a branch of the case expression (which corresponds to narrowing).

The higher-order constructs of Curry are translated into FlatCurry by defunctionalization [32]. Thus, lambda abstractions are transformed into top-level functions and there is a predefined operation *apply* to apply an expression of functional type to an argument (see [19,35] for more details).

As an example, consider the concatenation operation `(++)` shown above. Its pattern matching on the first argument can be represented by the following FlatCurry definition (where we use infix notations for readability):

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

$\texttt{(++)} :: \texttt{List}\ a \rightarrow \texttt{List}\ a \rightarrow \texttt{List}\ a$
$x_1 \texttt{ ++ } x_2 = \mathit{fcase}\ x_1\ \mathit{of}$
$\qquad\qquad\qquad \texttt{[]} \qquad \rightarrow x_2$
$\qquad\qquad\qquad y_1 : y_2 \rightarrow y_1 : (y_2 \texttt{ ++ } x_2)$

Note that it is possible to translate other functional logic languages, like TOY [27], or even Haskell into this intermediate format. Since our analysis tool is solely based on FlatCurry, it can also be used for other source languages provided that there is a translator from such languages into FlatCurry.

Mature implementations of Curry, like PAKCS [20] or KiCS2 [10], provide support for meta-programming by a library containing data types for representing FlatCurry programs 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 type

```
data Prog = Prog String     -- module name
                 [String]   -- imported modules
                 [TypeDecl] -- type declarations
                 [FuncDecl] -- function declarations
                 [OpDecl]   -- operator declarations
```

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 (of type `QName`, 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. Finally, the data type for expressions just reflects its formal definition:[4]

```
data Expr = Var  Int
          | Lit  Literal
          | Comb CombType QName [Expr]
          | Case CaseType Expr [(Pattern,Expr)]
          | Or   Expr Expr
          | Free [Int] Expr

data CombType = FuncCall | ConsCall

data Pattern = Pattern QName [Int]
```

Thus, variables are numbered, literals (like numbers or characters) are distinguished from combinations (`Comb`) which have a first argument to distinguish constructor applications and applications of defined functions. The remaining

[4] We present a slightly simplified version of the actual type definitions.

data type declarations for representing FlatCurry programs are similar but we omit them for brevity.

Using these data types, the concatenation operation `(++)`, which is defined in the module `Prelude`, is represented by the following data term of type `FuncDecl`:

```
Func ("Prelude","++")
     2
     Public
     ...
     (Rule [1,2]
       (Case Flex (Var 1)
         [(Pattern ("Prelude","[]") [],
           Var 2),
          (Pattern ("Prelude",":") [3,4],
           Comb ConsCall ("Prelude",":")
                [Var 3,
                 Comb FuncCall ("Prelude","++")
                      [Var 4, Var 2]])]))
```

3 Implementing Program Analyses

In this section we show how a program analysis is represented and implemented so that it can be used in the analysis system CASS.

3.1 Modeling Program Analyses

There are various frameworks and methods to analyze programs. Imperative programs are often analyzed by the use of control-flow graphs: the program is translated into a graph structure representing the potential control flow during run time, and a program analysis associates analysis information with execution points in the graph. This information can be taken into account by a code generator to produce efficient target code. Due to the fact that the control-flow graph typically contains cycles, the program analysis is performed by a fixpoint analysis according to the graph structure.

In declarative programs, the control flow is often not directly related to the program structure, e.g., due to demand-driven evaluation strategies. Program analyses for such languages often associates analysis information with the individual operations (functions, predicates) of the program. For instance, strictness analysis [28] computes for each function the arguments that must be evaluated in order to compute a value for a function call, or minimal function graphs [26] associates input/output pairs approximating possible function calls and their results in a given program.

For our analysis framework, we follow a similar approach. If $\mathcal{F}$ is the set of functions defined in a program and $\mathcal{A}$ an abstract domain representing analysis information, then a program analysis is a mapping $\alpha : \mathcal{F} \to \mathcal{A}$ which assigns to each operation $f \in \mathcal{F}$ an abstract value $\alpha(f) \in \mathcal{A}$. Since this is the formal basis of our *generic* analysis framework, the abstract domain $\mathcal{A}$ can vary for different program analysis. For a determinism analysis, $\mathcal{A}$ could contain the values *det* and *nondet*, where $\alpha(f) = det$ indicates that the operation f is deterministic, i.e., returns at most one value for a given argument value. For a demandedness or strictness analysis, the abstract domain might be 2^{Nat}, i.e., the analysis associates with each operation the set of demanded arguments represented by their position indices. Type inference can also be considered a program analysis [11] where the abstract domain contains all type expressions defined by the type system.

This interpretation of program analysis is appropriate for various reasons:

1. It is rather general so that quite different kinds of program analyses can be covered by appropriate domains, as discussed by the examples above.
2. The association of analysis information to operations contains useful information for a variety of applications (e.g., program understanding, code generation, program optimization), as shown later.
3. It is also a key to an efficient modular analysis: In many cases, the analysis results of an operation f depend on the analysis information associated with the operations called by f. For instance, if an operation f calls an operation g, the determinism status of f depends on the determinism status of g. Thus, if we know the determinism status of all operations called by f, the determinism status of f can be easily computed. Due to recursive calls, a fixpoint computation is required in general. However, recursive calls do not occur over module boundaries (at least, if cyclic module imports are not allowed, as in Curry). As a consequence, one can directly use the analysis results of imported operations so that fixpoint computations are required only during the analysis inside a module. Hence, the overall analysis can be performed in a modular manner, as we will see later.

This view of program analysis demands a bottom-up analysis of programs. First, the properties of base operations are analyzed, then the properties of the operations using the base operations, and so on. In contrast, a top-down analysis starting with an initial ("main") expression is not supported by our framework. On the one hand, one can argue that a bottom-up analysis is sufficient for interactive systems where the initial expression is not known at analysis time. On the other hand, it is sometimes possible to express "top-down oriented" analyses, like a groundness analysis in logic programming, in a bottom-up manner by choosing appropriate abstract domains. For instance, [9] presents a type and effect system to analyze groundness and non-determinism information in functional logic programs. This can be implemented as a bottom-up analysis, as discussed later in this paper.

A further important issue of our framework is the fact that we do not fix a particular semantical model on which we base our analysis. In order to prove the correctness of a program analysis, one has to define a concrete semantics which is approximated by abstract operations that operate on abstract values [12]. In order to cover various program analyses, the concrete semantics depends on the individual program analyses.

For example, consider a determinism analysis, as introduced above, where the abstract value $\alpha(f) = \mathit{det}$ should indicate the fact that all applications of f to some values (i.e., ground constructor terms) are evaluated in a deterministic manner. Hence, an appropriate concrete semantics for this analysis is a small-step operational semantics where the notion of a non-deterministic step is explicit. For instance, needed narrowing [4] or the small-step operational semantics described in [1] model the operational behavior of contemporary functional logic languages by a (non-deterministic) "single-step" evaluation relation "$\Rightarrow$." Now, we can define $\alpha(f) = \mathit{det}$ as the property that f is a *deterministic* operation, i.e., all evaluations of $(f\ t_1 \ldots t_n)$, where $t_1, \ldots, t_n$ are ground constructor terms, are deterministic. To be more precise, if

$$(f\ t_1 \ldots t_n) \Rightarrow e_1 \Rightarrow \cdots \Rightarrow e_k \Rightarrow e$$

and

$$(f\ t_1 \ldots t_n) \Rightarrow e_1 \Rightarrow \cdots \Rightarrow e_k \Rightarrow e'$$

are two evaluations of $(f\ t_1 \ldots t_n)$, then $e = e'$ holds. Below we will show an implementation of this determinism analysis in our framework.

As a further example, consider a demandedness analysis (also called strictness analysis in functional programming). In this case, the abstract values associated with an operation could be the set of demanded arguments of this operation. Here, "demanded" means that if this argument is undefined, the result of applying the operation to this argument is also undefined. In this case, a small-step operational semantics is not useful since it does not make the notion of "undefined" explicit. More appropriate is a declarative or denotational semantics with an explicit notion of undefined values. For instance, CRWL [14] is a standard declarative (i.e., strategy-independent) semantics for functional logic programs. In this semantics, the signature of functional logic programs is extended by a special symbol $\perp$ to represent undefined values. Hence, terms with occurrences of $\perp$ are called "partial terms." CRWL defines a calculus for approximation statements of the form $e \to t$ with the intended meaning "the partial constructor term t approximates the value of the expression e." For instance, if f denotes an infinite list defined by

$$f = 1 : f$$

then

$$f \to \perp, \quad f \to 1 : \perp, \quad f \to 1 : 1 : \perp, \ldots$$

are valid statements w.r.t. CRWL. In particular, if $e \to \perp$ is the only CRWL-statement for an expression e, then e is always undefined. Thus, the correctness of a demandedness analysis can be stated as follows: i is a demanded argument of f if, for all expressions $e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n$, $(f\ e_1 \ldots e_{i-1} \perp e_{i+1} \ldots e_n) \to \perp$ is the only statement derivable by CRWL. Thus, it is safe to evaluate a demanded argument before calling the operation. Such an optimization can improve the time and space behavior of functional logic programs, in particular, for non-deterministic computations [18].

As a consequence of the potential variety of semantical models and correctness requirements, we do not consider these aspects in this paper, since our main interest is to support the implementation of such program analyses in a practical system. Hence, we sketch in the following the steps required to implement a specific program analysis with our system.

3.2 Determinism Analysis

As discussed above, a program analysis associates with each operation analysis information describing some aspect of its semantics. Since most interesting semantic aspects are not computable, they are approximated by some abstract domain where each abstract value describes some set of concrete values [12]. For instance, the abstract value *det* of a determinism analysis denotes the set of all deterministic operations.

In order to approximate deterministic operations, a useful information is the property whether an operation is defined by "overlapping rules", i.e., whether more than one rule defining this operation is applicable for some ground call to this operation. An example for an operation that is defined by overlapping rules is the "choice" operation

```
x ? y  =  x
x ? y  =  y
```

To implement an "overlapping rules" analysis, one can use `Bool` as the abstract domain so that the abstract value `False` is interpreted as "defined by non-overlapping rules" and `True` is interpreted as "defined by overlapping rules". The "overlapping rules" analysis has the type

```
FuncDecl  →  Bool
```

which means that we associate a `Bool` value with each function definition. In contrast to the formal framework described above, we associate abstract values with function definitions rather than function names. This simplifies the implementation since we do not have to look up information in the program in order to compute analysis results, since the function definitions contain all information that is necessary for this example.

Overlapping rules are represented by `Or` constructors when source programs are mapped to FlatCurry programs [2,3]. For instance, the above choice operation is mapped into the FlatCurry program

(?) :: $a \to a \to a$
  x_1 ? $x_2 = x_1$ *or* x_2

Its representation as a data term of type `FuncDecl` is as follows:

```
Func ("Prelude","?")
     2
     Public
     ...
     (Rule [1,2]  (Or (Var 1) (Var 2)))
```

Thus, we can implement the "overlapping rules" analysis by looking for occurrences of the data constructor `Or` in the definition of each function. Based on the data type definitions sketched in Section 2 and some standard functions, we can implement this analysis as follows:

```
isOverlapping :: FuncDecl  → Bool
isOverlapping (Func _ _ _ _ (Rule _ e)) = orInExpr e

orInExpr :: Expr  → Bool
orInExpr (Var _)       = False
orInExpr (Lit _)       = False
orInExpr (Comb _ _ es) = any orInExpr es
orInExpr (Case _ e bs) = orInExpr e
                         || any (orInExpr . snd) bs
orInExpr (Or _ _)      = True
orInExpr (Free _ e)    = orInExpr e
```

As discussed above, a determinism analysis could be based on the abstract domain described by the data type

```
data DetDom = Det | NonDet
```

Here, `Det` is interpreted as "the operation always evaluates in a deterministic manner on ground arguments" (see above for a precise specification). However, `NonDet` is interpreted as "the operation *might* evaluate in different ways for given ground arguments." The apparent imprecision is due to the approximation of the analysis. For instance, if the function `f` is defined by overlapping rules and the function `g` *might* call `f`, then `g` is judged as non-deterministic (since it is generally undecidable whether `f` is actually called by `g` in some run of the program). Our analysis has to take into account such dependencies. To do so, the determinism analysis requires to examine the current function as well as all directly or indirectly called functions for overlapping rules. Due to recursive function definitions, this analysis cannot be done in one shot—it requires a fixpoint computation. CASS provides such fixpoint computations and requires only the implementation of an operation of type

```
FuncDecl  →  [(QName,a)]  →  a
```

where "`a`" denotes the type of abstract values. The second argument of type `[(QName,a)]` represents the currently known analysis values for the functions *directly* used in this function declaration. One might ask why this information is not represented as a mapping from function names to analysis results, i.e., why the second argument has not the type

```
(QName  →  a)
```

The reason is that typical analyses which take dependencies into account are *all* or *any* analysis, i.e., the analysis information about an operation combine information about *all* or *any* of the operations on which they depend. If we represent this analysis information as a mapping, we might also need the information about the domain of this mapping. Therefore, it is easier to work on a list representation where all this information is directly available.

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

```
detFunc :: FuncDecl  → [(QName,DetDom)]  → DetDom
detFunc (Func f _ _ _ (Rule _ e)) calledFuncs =
  if orInExpr e || freeVarInExpr e ||
     any (==NonDet) (map snd calledFuncs)
  then NonDet
  else Det
```

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

The actual analysis is performed by defining some start value for all functions (the "bottom" value of the abstract domain, here: `Det`) and performing a fixpoint computation for the abstract values of these functions. CASS uses a working list approach as default but also supports other methods to compute fixpoints. The termination of the fixpoint computation can be ensured by standard assumptions in abstract interpretation [12], e.g., by choosing a finite abstract domain and monotonic operations, or by widening operations.

3.3 Integrating an Analysis into CASS

To support the inclusion of different analyses in CASS, there is an abstract type "`Analysis a`" denoting a program analysis with abstract domain "`a`". Furthermore, CASS offers several constructor operations for various kinds of analyses. Each analysis has a name provided as a first argument to these constructors. The name is used to store the analysis information persistently and to pass specific analysis tasks to workers (see below for more details). For instance, a simple function analysis which depends only on a given function definition can be defined by the analysis constructor

```
funcAnalysis :: String  → (FuncDecl  → a)  → Analysis a
```

The arguments are the analysis name and the actual analysis function. For instance, the "overlapping rules" analysis can be specified as

```
overlapAnalysis :: Analysis Bool
overlapAnalysis = funcAnalysis "Overlapping" isOverlapping
```

Another analysis constructor supports the definition of a function analysis with dependencies:

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

Here, the second argument specifies the start value of the fixpoint analysis, i.e., the bottom element of the abstract domain. Thus, the complete determinism analysis can be specified as

```
detAnalysis :: Analysis DetDom
detAnalysis = dependencyFuncAnalysis "Deterministic" Det detFunc
```

It should be noted that this definition is sufficient to execute the analysis with CASS since the analysis system takes care of computing fixpoints, calling the analysis functions with appropriate values, analyzing imported modules, etc. Thus, the programmer can concentrate on implementing the logic of the analysis and is freed from many tedious implementation details.

If we have defined an analysis in this way, we have to register it so that CASS knows about its existence and can call it in the right way. In principle, this registration could be done dynamically, but currently only a static registration is supported for the sake of simplicity. For this purpose, the implementation of CASS contains a constant

```
registeredAnalysis :: [RegisteredAnalysis]
```

keeping the information about all available analyses. To register a new analysis, it has to be added to this list of registered analyses (as described below) and CASS has to be recompiled.

Abstract values, like values of type `Bool` or `DetDom`, are program entities that might be difficult to interpret for the user of CASS. Therefore, each analysis must be registered in CASS together with a "show" function that maps abstract values into strings to be shown to the user.[5] An analysis can be registered with the auxiliary operation

```
cassAnalysis :: Analysis a → (a → String) → RegisteredAnalysis
```

that has the specification of the program analysis and a show function as arguments. The explicit definition of the show function for each analysis allows for some flexibility in the presentation of the analysis information. For instance, different analyses can use the same abstract domain, like `Bool`, with different intended meanings.

To proceed with our example analyses, showing the results of the overlapping and determinism analyses can be implemented as follows:

```
showOverlap :: Bool → String
showOverlap True  = "overlapping"
showOverlap False = "non-overlapping"

showDet :: DetDom → String
showDet NonDet = "non-deterministic"
showDet Det    = "deterministic"
```

With these definitions, we can register our analyses by the definition

[5] Alternative visualizations of analysis information, e.g., as graphs, are planned for the future.

```
registeredAnalysis =
  [cassAnalysis overlapAnalysis showOverlap
  ,cassAnalysis detAnalysis     showDet  ]
```

in the CASS implementation. After compiling CASS, they are immediately available.

3.4 Analysis of Types

Sometimes one is also interested in analyzing information about data types rather than functions. For instance, the Curry implementation KiCS2 [10] has an optimization for higher-order deterministic operations. This optimization requires some information about the higher-order status of data types, i.e., whether a term of some type might contain functional values.

CASS supports such analyses by appropriate analysis constructors. A simple type analysis which depends only on a given type declaration can be specified by

```
typeAnalysis :: String → (TypeDecl → a) → Analysis a
```

A more complex type analysis depending also on information about the types used in the type declaration can be specified by

```
dependencyTypeAnalysis ::
  String → a → (TypeDecl → [(QName,a)] → a) → Analysis a
```

Similarly to a function analysis, the second argument is the start value of the fixpoint analysis and the third argument computes the abstract information about the type names used in the type declaration.

The remaining entities in a Curry program that might be analyzed are data constructors. Since their definition only contains the argument types, it may seem uninteresting to provide a useful analysis for them. However, sometimes it is interesting to analyze their context so that there is an analysis constructor of type

```
constructorAnalysis ::
  String → (ConsDecl → TypeDecl → a) → Analysis a
```

Thus, the analysis operation of type

```
(ConsDecl → TypeDecl → a)
```

gets for each constructor declaration also the corresponding type declaration as an argument. This information could be used to compute the sibling constructors, e.g., the sibling for the constructor `True` is `False`. The information about sibling constructors is useful to check whether a function is completely defined, i.e., contains a case distinction for all possible patterns. For instance, the operation (in FlatCurry notation)

```
not x = case x of True  → False
                  False → True
```

is completely defined whereas

```
cond b x = case b of True  →  x
```

is incompletely defined since it fails on `False` as the first argument. To check this property, information about sibling constructors is obviously useful. But how can we provide the information about sibling constructors, which is computed by a type analysis, in a "complete pattern" analysis for functions? For this purpose, CASS supports the combination of different analyses.

3.5 Analysis Combinators

Sometimes it is useful to define an analysis based on information computed by another analysis. We already discussed the use of sibling constructors in an analysis for complete pattern matching. Another example is the use of the "overlapping rules" analysis in the determinism analysis. For such purposes, CASS supports "analysis combinators" to implement the combination of different analyses. Thus, an analysis developer can define an analysis that is based on information computed by another analysis.

To make analysis combination possible, we need to pass information computed by one analysis into another analysis. For this purpose, there is an abstract type "`ProgInfo a`" to represent the analysis information of type "`a`" for a given module and its imports. In order to look up analysis information about some entity, there is an operation

```
lookupProgInfo:: QName  →  ProgInfo a  →  Maybe a
```

Now, CASS provides the analysis constructor operation

```
combinedFuncAnalysis :: String  →  Analysis b
    →  (ProgInfo  b  →  FuncDecl  →  a)  →  Analysis a
```

to implement a function analysis depending on some other analysis. The second argument is some base analysis computing abstract values of type "`b`". The analysis function (third argument) gets, in contrast to a simple function analysis, the analysis information computed by this base analysis as its first argument.

For instance, if the sibling constructor analysis is defined as

```
siblingCons :: Analysis [QName]
siblingCons = constructorAnalysis ...
```

then the pattern completeness analysis might be defined by

```
patCompAnalysis :: Analysis Bool
patCompAnalysis =
  combinedFuncAnalysis "PatComplete" siblingCons isPatComplete

isPatComplete :: ProgInfo [QName]  →  FuncDecl  →  Bool
isPatComplete siblinginfo fundecl = ...
```

Thus, one can use the (type analysis) information about sibling constructors (`siblinginfo`) inside the definition of pattern completeness.

Similarly, other kinds of analyses can also be combined with some base analysis by using the following analysis constructors:

```
combinedTypeAnalysis :: String → Analysis b
  → (ProgInfo  b → TypeDecl → a) → Analysis a

combinedDependencyFuncAnalysis :: String → Analysis b → a
  → (ProgInfo b → FuncDecl → [(QName,a)] → a) → Analysis a

combinedDependencyTypeAnalysis :: String → Analysis b → a
  → (ProgInfo b → TypeDecl → [(QName,a)] → a) → Analysis a
```

For instance, an analysis for checking whether a function is totally defined, i.e., always reducible for all ground arguments, can be based on the pattern completeness analysis. It is a dependency analysis so that it can be defined as follows (in this case, `True` is the bottom element since the abstract value `False` denotes "might not be totally defined"):

```
totalAnalysis :: Analysis Bool
totalAnalysis =
  combinedDependencyFuncAnalysis "Total" patCompAnalysis True isTotal

isTotal :: ProgInfo Bool → FuncDecl → [(QName,Bool)] → Bool
isTotal pcinfo fdecl calledfuncs =
  (maybe False id
         (lookupProgInfo (funcName fdecl) pcinfo))
  && all snd calledfuncs
```

Hence, a function is totally defined if it is pattern complete and depends only on totally defined functions.

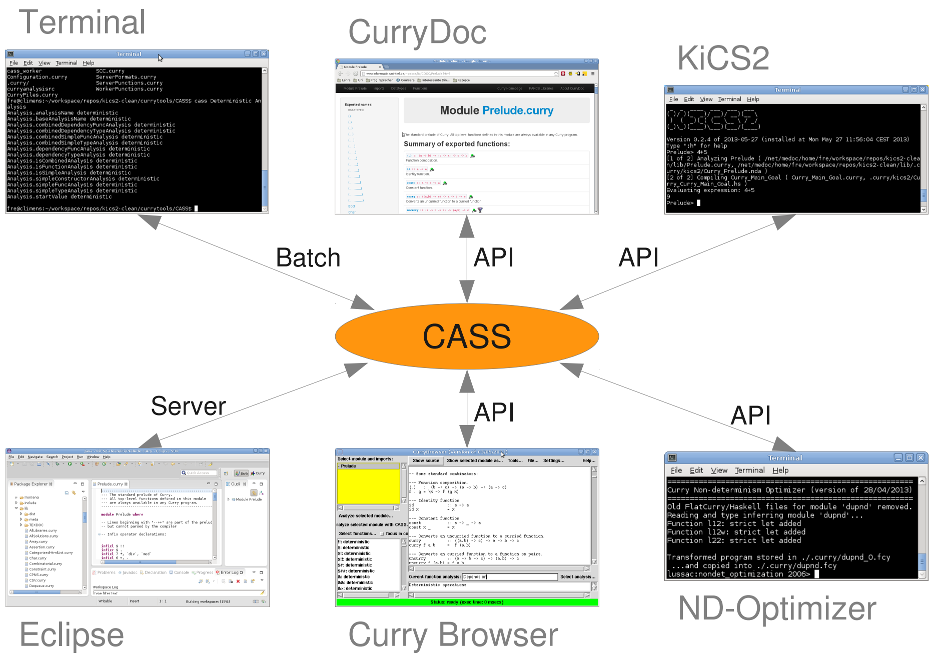
Further examples of combined analyses are the higher-order function analysis used in KiCS2 (see above) where the higher-order status of a function depends on the higher-order status of its argument types, and the non-determinism analysis of [9] where non-determinism effects are analyzed based on groundness information.

4 Using the Analysis System

As mentioned above, a program analysis is useful for various purposes, e.g., the implementation and transformation of programs, tool and documentation support for programmers, etc. Therefore, the results computed by some analysis registered in CASS can be accessed in various ways. Currently, there are three methods for this purpose:

Batch mode: CASS is started with a module and analysis name. Then this analysis is applied to the module and the results are printed (using the analysis-specific show function, see above).

API mode: If the analysis information should be used in an application implemented in Curry, the application program could use the CASS interface



```
cass_worker                    SCC.curry
Configuration.curry            ServerFormats.curry
.curry/                        ServerFunctions.curry
curryanalysisrc                WorkerFunctions.curry
CurryFiles.curry
fre@climens:~/workspace/repos/kics2-clean/currytools/CASS$ cass Deterministic An
alysis
Analysis.analysisName deterministic
Analysis.baseAnalysisName deterministic
Analysis.combinedDependencyFuncAnalysis deterministic
Analysis.combinedDependencyTypeAnalysis deterministic
Analysis.combinedSimpleFuncAnalysis deterministic
Analysis.combinedSimpleTypeAnalysis deterministic
Analysis.dependencyFuncAnalysis deterministic
Analysis.dependencyTypeAnalysis deterministic
Analysis.isCombinedAnalysis deterministic
Analysis.isFunctionAnalysis deterministic
Analysis.isSimpleAnalysis deterministic
Analysis.simpleConstructorAnalysis deterministic
Analysis.simpleFuncAnalysis deterministic
Analysis.simpleTypeAnalysis deterministic
Analysis.startValue deterministic

fre@climens:~/workspace/repos/kics2-clean/currytools/CASS$
```

```
Version 0.2.4 of 2013-05-27 (installed at Mon May 27 11:56:04 CEST 2013)
Type ":h" for help
Prelude> 4+5
[1 of 2] Analyzing Prelude ( /net/medoc/home/fre/workspace/repos/kics2-clea
n/lib/Prelude.curry, /net/medoc/home/fre/workspace/repos/kics2-clean/lib/.c
urry/kics2/Curry_Prelude.nda )
[2 of 2] Compiling Curry_Main_Goal ( Curry_Main_Goal.curry, .curry/kics2/Cu
rry_Curry_Main_Goal.hs )
Evaluating expression: 4+5
9
Prelude>
```

```
================================================================
Curry Non-determinism Optimizer (version of 28/04/2013)
================================================================
Old FlatCurry/Haskell files for module 'dupnd' removed.
Reading and type inferring module 'dupnd'...
Function l12: strict let added
Function l12w: strict let added
Function l22: strict let added

Transformed program stored in ./.curry/dupnd_O.fcy
 ...and copied into ./.curry/dupnd.fcy
lussac:nondet_optimization 2006>
```

Fig. 1. Using CASS in different contexts

operations to start an analysis and use the computed results for further processing.

Server mode: If the analysis results should be used in an application implemented in some language that does not have a direct interface to Curry, one can start CASS in a server mode. In this case, one can connect to CASS via some socket using a simple communication protocol that is specified in the documentation of CASS.

Figure 1 shows some uses of CASS which are discussed in the following. The use of CASS in batch mode is obvious. This mode is useful to get a quick access to analysis information so that one can experiment with different abstractions, fixpoint computations, etc.

If one wants to access CASS inside an application implemented in Curry, one can use some interface operation of CASS. For instance, CASS provides an operation

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

to apply an analysis (first argument) to some module (whose name is given in the second argument). The result is either the analysis information computed for this module or an error message in case of some execution error. This access to CASS is used in the documentation generator CurryDoc [16] to describe some operational aspects of functions (e.g., pattern completeness, non-determinism, solution completeness), the Curry compiler KiCS2 [10] to get information about the determinism and higher-order status of functions, and the non-determinism optimizer described in [18] to obtain information about demanded arguments and non-deterministic functions. Furthermore, there is also a similar operation

```
analyzeModule ::
  String → String → IO (Either (ProgInfo String) String)
```

which takes an analysis name and a module name as arguments and yields the textual representation of the computed analysis results. This is used in the CurryBrowser [17] which allows the user to browse through the modules of a Curry application and apply and visualize various analyses for each module or function. Beyond some specific analyses like dependency graphs, all function analyses registered in CASS are automatically available in the CurryBrowser.

The server mode of CASS is used in a recently developed Eclipse plug-in for Curry [29] which also supports the visualization of analysis results inside Eclipse. Since this plug-in is implemented in a Java-based framework, the access to CASS is implemented via a textual protocol over a socket connection. The protocol defines a couple of commands to use CASS, like

```
GetAnalysis
SetCurryPath <dir1>:<dir2>:...
AnalyzeModule <analysis name> <otype> <module name>
...
```

The command `GetAnalysis` allows to query the names and output formats (see below) of all available analyses. The Eclipse plug-in for Curry uses this command to initialize the analysis selection menus. The command `SetCurryPath` instructs CASS to use the given directories to search for modules to be analyzed. This is necessary since the CASS server might be started in a different location than its client.

To analyze a complete module, the client of CASS can use the command `AnalyzeModule` which applies an analysis to a module and returns the analysis results in the specified format (second argument). CASS currently supports plain strings, XML, or a Curry term format as output formats. However, it is up to the implementer of a program analysis to support other formats. Currently, we are working on more options to visualize analysis information in the Eclipse plug-in rather than strings, e.g., term or graph visualizations.

Beyond the analysis of a complete module, there are also commands to return analysis information of the interface of the module or individual entities like operations, types, or data constructors.

5 Implementation

As mentioned above, CASS is implemented in Curry using the features for meta-programming as sketched in Section 2. Since the analysis programmer only provides operations to analyze a function, type, or data constructor, as shown in Section 3, the main task of CASS is to supply these operations with the appropriate parameters in order to compute the analysis results.

CASS is intended to analyze larger applications consisting of many modules. Thus, a simple implementation by concatenating all modules into one large program to be analyzed would not be efficient enough. Hence, CASS performs a separate analysis of each module by the following steps:

1. The imported modules are analyzed.
2. The analysis information of the interface of the imported modules are loaded.
3. The module is analyzed. If the analysis is a dependency analysis, it is evaluated by a fixpoint computation where the specified start value is used as initial values for the locally defined (i.e., non-imported) entities.

Obviously, this scheme can be simplified in case of a simple analysis without dependencies, since such an analysis does not require the imported entities. For a combined analysis, the base analysis is performed before the main analysis is executed.

In order to speed up the complete analysis process, CASS implements a couple of improvements to this general analysis process sketched above. First, the analysis information for each module is persistently stored. Hence, before a module is analyzed, it is checked whether there already exists a storage with the analysis information of this module and whether the time stamp of this information is newer than the source program with all its direct or indirect imports. If the storage is found and is still valid, the stored information is used. Otherwise, the information is computed as described above and then persistently stored. This has the advantage that, if only the main module has changed and needs to be re-analyzed, the analysis time of a large application is still small.

To exploit multi-core or distributed execution environments, the implementation of CASS is designed as a master/worker architecture where a master process coordinates all analysis activities and each worker is responsible to analyze a single module. Thus, when CASS is requested to analyze some module, the master process computes all import dependencies together with a topological order of all dependencies. The standard prelude module (without import dependencies) is the first module to be analyzed and the main module is the last one. The master process iterates on the following steps until all modules are analyzed:

– If there is a free worker and all imports of the first module are already analyzed, pass the first module to the free worker and delete it from the list of modules.
– If the first module contains imports that are not yet analyzed, wait for the termination of an analysis task of a worker.
– If a worker has finished the analysis of a module, mark all occurrences of this module as "analyzed."

Since contemporary Curry implementations do not support thread creation, the workers are implemented as processes that are started at the beginning and terminated at the end of the entire execution. The number of workers can be defined by some system parameter.

The current distribution of CASS[6] contains fourteen program analyses, including the analyses discussed in Section 3. Further analyses include a "solution completeness" analysis (which checks whether a function might suspend due to residuation), a "right-linearity" analysis (used to improve the implementation of functional patterns [6]), an analysis of demanded arguments (used to optimize non-deterministic computations [18]), or a combined groundness/non-determinism analysis based on a type and effect system [9].

6 Practical Evaluation

We have already discussed some practical applications of CASS in Section 4. These applications demonstrate that the current implementation with a module-wise analysis, storing analysis information persistently, and incremental re-analysis is good enough to use CASS in practice. In order to get some ideas about the efficiency of the current implementation, we made some benchmarks and report their results in this section. Since all analyses contained in CASS have been developed and described elsewhere (see the references above), we do not evaluate their precision but only their execution efficiency.

CASS is intended to analyze larger systems. Thus, we omit the data for analyzing single modules but present the analysis times for four different Curry applications: the interactive environment (read/eval/print loop) of KiCS2, the analysis system presented in this paper, the interactive analysis environment CurryBrowser [17], and the module database,[7] a web application generated from an entity/relationship model with the web framework Spicey [22]. In order to get an impression of the size of each application, the number of modules (including imported system modules) is shown for each application. Typically, most modules contain 100-300 lines of code, where the largest one has more than 900 lines of code.

For the benchmarks, we applied two kinds of function analysis with dependencies, i.e., where fixpoint computations are required to compute the analysis results: an analysis of demanded arguments and a groundness analysis.

The analysis of demanded arguments has already been introduced in Section 3. The abstract domain of this analysis is

```
type DemandedArgs = [Int]
```

Thus, if the demandedness analysis computes a list `[`x_1`,...,`x_k`]` for an operation f, then the argument at position x_i is demanded ($i = 1, \ldots, k$). For instance, the list of demanded arguments of the list concatenation `(++)` is `[1]`, i.e., only the first argument is demanded. The basic structure of this analysis is described in [18] where the analysis results are used to improve non-deterministic computations by some program transformation.

[6] CASS is part of the distributions of the Curry systems KiCS2 [10] and PAKCS [20].

[7] `http://mdb.ps.informatik.uni-kiel.de/`

Application:	KiCS2 REPL		CASS		CurryBrowser		ModuleDB	
Modules:	32		46		71		85	
Analysis:	Demand	Ground	Demand	Ground	Demand	Ground	Demand	Ground
1 worker	8.32	8.50	10.19	10.27	19.22	19.36	29.61	30.79
2 workers:	5.97	5.98	6.85	6.95	12.32	12.49	20.16	20.51
4 workers:	5.58	5.57	6.14	6.24	10.21	10.66	18.30	18.19
Re-analyze:	1.41	1.39	1.24	1.26	1.99	2.00	2.44	2.43

Table 1. Using CASS in different contexts

The groundness analysis is influenced from logic programming. The groundness status of arguments are often used to improve the target code of compiled programs, as in the language Mercury with its strong mode system [33]. In functional logic programs without restrictive modes, groundness information can be used to improve the precision of a determinism analysis, as shown in [9], where a type and effect system is proposed to analyze groundness and non-determinism information in functional logic programs. For instance, the determinism analysis presented in Section 3.2 classified an operation as potentially non-deterministic if its definition contains a free variable. However, if this free variable is passed to another operation that ignores this argument or binds it in a deterministic manner, the defined operation is actually deterministic although it contains a free variable. Thus, the information about the groundness of functional calls relative to the groundness of its arguments could be quite valuable. Therefore, the abstract domain for a groundness analysis of functional logic programs can be defined as follows:

```
data Ground = G | P [Int] | A
```

The abstract value `G` denotes an operation that definitely returns a ground term, `P [`x_1`,...,`x_k`]` an operation that might return a non-ground value if the argument at position i is non-ground (for some $i \in \{x_1, \ldots, x_k\}$), and `A` an operation that might return a non-ground value. For instance, the groundness information associated with the list concatenation `(++)` is `P [1,2]`, and it is `G` for the negation operation `not` defined in Section 3.4. Since this kind of groundness information does not depend on the call structure of the program w.r.t. some main expression, it can be computed in a bottom-up manner so that it fits well to our analysis framework. Details about its computation can be found in [9].

Table 1 contains the elapsed time (in seconds) needed to analyze the applications described above for different numbers of workers. Each analysis has always been started from scratch, i.e., all persistently stored information were deleted at the beginning, except for the last row which shows the times to re-analyze the application where only the main module has been changed. In this case, the actual analysis time is quite small but most of the total time is spent to check all module dependencies for possible updates. The benchmarks were executed on a

Linux machine running Ubuntu 12.04 with an Intel Core i5 (2.53GHz) processor with four cores where CASS was compiled with KiCS2 (Version 0.3.1).

The speedup related to the number of workers is not optimal. This might be due to the fact that the dependencies between the modules are complex so that there are not many opportunities for an independent analysis of modules, i.e., workers might have to wait for the termination of the analysis of modules which are imported by many other modules. Nevertheless, the approach shows that there is a potential to exploit the computing power offered by modern computers. Furthermore, the absolute run times are acceptable. It should also be noted that, during system development, the times are lower due to the persistent storing of analysis results.

7 Conclusions and Related Work

In this paper we presented CASS, a tool to analyze functional logic programs. CASS supports various kinds of program analyses by a general notion of analysis functions that map program entities into analysis information. In order to implement an analysis that also depends on information about other entities used in a definition, CASS supports "dependency analyses" that require a fixpoint computation to yield the final analysis information. Moreover, different analyses can be combined so that one can define an analysis that is based on the results of another analysis. Using these different constructions, the analysis developer can concentrate on defining the logic of the analysis and is freed from the details to invoke the analysis on modules and complete application systems. To analyze larger applications efficiently, CASS performs a modular and incremental analysis where already computed analysis information is persistently stored. Thus, CASS does not support top-down or goal-oriented analyses but only bottom-up analyses which is acceptable for large applications or interactive systems with unknown initial goals. The implementation of CASS supports different modes of use (batch, API, server) so that the registered analyses can be accessed by various systems, like compilers, program optimizers, documentation generators, or programming environments. Currently, CASS produces output in textual form. The support for other kinds of visualizations is a topic for future work.

The analysis of programs is an important topic for all kinds of languages so that there is a vast body of literature. Most of such works is related to the development and application of various analysis methods (where some of them related to functional logic programs have already been discussed in this paper), but there are less works on the development or implementation of program analyzers. An example of such an approach, that is in some aspects similar to our work, is Hoopl [31]. Hoopl is a framework for data flow analysis and transformation. Like CASS, Hoopl eases the definition of analyses by offering high-level abstractions and releases the user from tasks like writing fixpoint computations. In contrast to our work, Hoopl works on a generic representation of data flow graphs, whereas CASS performs incremental, module-wise analyses on an already existing representation of functional logic programs.

Another related system is Ciao [24], a logic programming system with an advanced preprocessor to analyze, optimize, and verify logic programs [25]. Similarly to CASS, the Ciao preprocessor also analyzes declarative programs in a modular and incremental manner. However, the Ciao preprocessor does not offer a high-level generic interface to implement new program analyses in a type safe manner, which is the main objective of the strongly typed analysis constructors provided by CASS.

There are only a few approaches or tools directly related to the analysis of combined functional logic programs, as already discussed in this paper. The examples in this paper show that this combination is valuable since analysis aspects of pure functional and pure logic languages can be treated in this combined framework, like demand and higher-order aspects from functional programming and groundness and determinism aspects from logic programming. An early system in this direction is CIDER [21]. CIDER supports the analysis of single Curry modules together with some graphical tracing facilities. A successor of CIDER is CurryBrowser [17], already mentioned above, which supports the analysis and browsing of larger applications. CASS can be considered as a more efficient and more general implementation of the analysis component of CurryBrowser.

For future work, we will add further analyses in CASS with more advanced abstract domains. Since this might lead to analyses with substantial run times, the use of parallel architectures might be more relevant. Thus, it would also be interesting to develop advanced methods to analyze module dependencies in order to obtain a better distribution of analysis tasks between the workers.

Acknowledgements. The authors are grateful to Heiko Hoffmann for his contribution to an initial version of the analysis system and to Sandra Dylus for her suggestions to improve this paper.

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. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
3. S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 199–206. ACM Press, 2001.
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. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.
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)*, pp. 6–22. Springer LNCS 3901, 2005.

7. S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, Vol. 53, No. 4, pp. 74–85, 2010.
8. S. Antoy and M. Hanus. New Functional Logic Design Patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pp. 19–34. Springer LNCS 6816, 2011.
9. B. Braßel and M. Hanus. Nondeterminism Analysis of Functional Logic Programs. In *Proceedings of the International Conference on Logic Programming (ICLP 2005)*, pp. 265–279. Springer LNCS 3668, 2005.
10. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pp. 1–18. Springer LNCS 6816, 2011.
11. P. Cousot. Types as Abstract Interpretations. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 316–331, 1997.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.
13. S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 63–74. ACM Press, 2007.
14. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, Vol. 40, pp. 47–87, 1999.
15. 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.
16. M. Hanus. CurryDoc: A Documentation Tool for Declarative Programs. In *Proc. 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, pp. 225–228. Research Report UDMI/18/2002/RR, University of Udine, 2002.
17. 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)*, pp. 61–74, 2006.
18. M. Hanus. Improving Lazy Non-Deterministic Computations by Demand Analysis. In *Technical Communications of the 28th International Conference on Logic Programming*, volume 17, pp. 130–143. Leibniz International Proceedings in Informatics (LIPIcs), 2012.
19. M. Hanus. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pp. 123–168. Springer LNCS 7797, 2013.
20. M. Hanus, S. Antoy, B. Braßel, 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/`, 2013.
21. M. Hanus and J. Koj. CIDER: An Integrated Development Environment for Curry. In *Proc. of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, pp. 369–373. Report No. 2017, University of Kiel, 2001.
22. M. Hanus and S. Koschnicke. An ER-based Framework for Declarative Web Programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pp. 201–216. Springer LNCS 5937, 2010.

23. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
24. M. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J.F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, Vol. 12, No. 1-2, pp. 219–252, 2012.
25. M.V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, Vol. 58, No. 1-2, pp. 115–140, 2005.
26. N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 296–306, 1986.
27. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
28. A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. International Symposium on Programming*, pp. 269–281. Springer LNCS 83, 1980.
29. M. Palkus. An Eclipse-Based Integrated Development Environment for Curry. Master's thesis, Christian-Albrechts-Universität zu Kiel, 2012.
30. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.
31. N. Ramsey, J. Dias, and S. Peyton Jones. Hoopl: a modular, reusable library for dataflow analysis and transformation. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell (Haskell 2010)*, pp. 121–134. ACM Press, 2010.
32. J.C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*, pp. 717–740. ACM Press, 1972.
33. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, Vol. 29, No. 1-3, pp. 17–64, 1996.
34. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.
35. D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.