In *Proc. of the International Conference on Logic Programming, ICLP 2005.*
In Springer LNCS 3668, pp. 265-279, 2005

Nondeterminism Analysis of Functional Logic Programs*

Bernd Braßel and Michael Hanus

Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany.
{bbr,mh}@informatik.uni-kiel.de

Abstract. Information about the nondeterminism behavior of a functional logic program is important for various reasons. For instance, a nondeterministic choice in I/O operations results in a run-time error. Thus, it is desirable to ensure at compile time that a given program is not going to crash in this way. Furthermore, knowledge about nondeterminism can be exploited to optimize programs. In particular, if functional logic programs are compiled to target languages without builtin support for nondeterministic computations, the transformation can be much simpler if it is known that the source program is deterministic.
In this paper we present a nondeterminism analysis of functional logic programs in form of a type/effect system. We present a type inferencer to approximate the nondeterminism behavior via nonstandard types and show its correctness w.r.t. the operational semantics of functional logic programs. The type inference is based on a new compact representation of sets of types and effects.

1 Introduction

Functional logic languages [8] aim to integrate the best features of functional and logic languages in order to provide a variety of programming concepts. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming can be combined with logic programming features like computing with partial information (logic variables), constraint solving, and nondeterministic search for solutions. This combination leads to optimal evaluation strategies [2] and new design patterns [3] that can provide better programming abstractions, e.g., for implementing graphical user interfaces [10] or dynamic web pages [11]. One of the key points in this integration is the treatment of nondeterministic computations. Usually, the top-level of an application written in a functional logic language is a sequence of I/O operations applied to the outside world (e.g., see [25]). Since the outside world (e.g., file system, Internet) cannot be copied in nondeterministic branches, all nondeterminism in logic computations must be encapsulated, as proposed in [4, 13] for the declarative multi-paradigm language Curry [15], otherwise a run-time

* The research described in this paper has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-1.

error occurs. Therefore, it is desirable to ensure at compile time that this cannot happen for a given program. Since this is undecidable in general, one can try to approximate the nondeterminism behavior by some program analysis. As a further motivation, the results of such an analysis can also be used for program optimization. For instance, if functional logic programs are compiled to target languages without builtin support for nondeterministic computations (e.g., imperative or functional languages), the compilation process can be considerably simplified for deterministic source programs.

Existing determinism analyses for (functional) logic languages cannot be directly adapted to Curry due to its advanced lazy operational semantics ensuring optimal evaluation for large classes of programs [2]. This demand-driven semantics has the effect that the occurrence of nondeterministic choices depends on the demandedness of argument evaluation (see also [14]). Therefore, analyses for languages like Prolog [24, 6], Mercury [16], or HAL [7] do not apply because they do not deal with lazy evaluation. On the other hand, analyses proposed for narrowing-based functional logic languages dealing with lazy evaluation cannot handle residuation, which additionally exists in Curry and is important to connect external operations, and rely on the non-ambiguity condition [20] which is too restrictive in practice. Furthermore, these analyses are either applied during run time (like in Babel [20] and partially in K-Leaf [19]), or are unable to derive groundness information for function calls in arguments (like in K-Leaf).

We present a static analysis of functional logic programs with a demand-driven evaluation strategy. The analysis has the form of a type/effect system [22]. Such systems can be seen as extensions of classical type systems known from functional languages. In our analysis the types represent information about the groundness of the considered expressions, and the effects provide information about the possible source of nondeterministic branches. The inclusion of groundness information is necessary since the same function might evaluate deterministically or not, depending on the instantiation of its arguments. The idea of this type/effect system has been proposed in [14]. In the current paper we propose a slightly modified system and show its correctness w.r.t. a recently developed high-level operational semantics of functional logic programs [1] that covers all operational aspects, in particular, the sharing of subterms which is important in practice but has not been addressed in [14]. Furthermore, we present a new method to *infer* types and effects (which was not covered in [14]) and show the correctness of this inference. In order to make the type/effect inference feasible, we introduce a new compact representation of sets of types and effects.

Due to lack of space, all proofs and details about the implementation are omitted. They can be found in the full version of the paper that is available from `http://www.informatik.uni-kiel.de/~mh`.

2 The Type/Effect Analysis

In this section we define a type/effect system based on the ideas in [14] and show its correctness w.r.t. the operational semantics of functional logic programs

$P ::= D_1 \ldots D_m$	(program)	*Domains*	
$D ::= f(x_1, \ldots, x_n) = e$	(function definition)		
$e ::= x$	(variable)	$P_1, P_2, \ldots \in \mathit{Prog}$	(Programs)
$\mid s(e_1, \ldots, e_n)$	(constructor or function call)	$x, y, z, \ldots \in \mathit{Var}$	(Variables)
		$a, b, c, \ldots \in \mathcal{C}$	(Constructors)
$\mid$ *let* $x = e_1$ *in* e_2	(let binding)	$f, g, h, \ldots \in \mathcal{F}$	(Functions)
$\mid e_1$ *or* e_2	(disjunction)	$s, t, u, \ldots \in \mathcal{C} \cup \mathcal{F}$	
$\mid$ *case* e *of* $\{\overline{p_k \to e_k}\}$	(rigid case)	$p_1, p_2, \ldots \in \mathit{Pat}$	(Patterns)
$\mid$ *fcase* e *of* $\{\overline{p_k \to e_k}\}$	(flexible case)	$e, e_1, e_2, \ldots \in \mathit{Exp}$	(Expressions)
$p ::= c(x_1, \ldots, x_n)$	(pattern)		

Fig. 1. Syntax of flat programs

developed in [1]. We assume familiarity with the basic ideas of functional logic programming (see [8] for a survey).

2.1 Flat Functional Logic Programs

Since a determinism analysis of functional logic programs should provide information about nondeterministic branches that might occur during run time, it requires detailed information about the operational behavior of programs. Recently, it has been shown that an intermediate *flat* representation of programs [12] is a good basis to provide this information. In flat programs, the pattern matching strategy (which determines the demand-driven evaluation of goals) is explicitly given by case expressions. This flat representation constitutes the kernel of modern functional logic languages like Curry [9, 15] or Toy [21]. Thus, our approach is applicable for general lazy functional logic languages although the examples and implementation are for Curry.

The syntax of flat programs is shown in Figure 1. There and in the following we write $\overline{o_k}$ to denote a sequence $o_1, \ldots, o_k$ ($\overline{o_0}$ is empty). A flat program is a set of function definitions, i.e., the arguments are pairwise different variables and the right-hand side consists of variables, constructor/function applications, let bindings, disjunctions to represent nondeterministic choices, and case expressions to represent pattern matching. The difference between *case* and *fcase* corresponds to principles of residuation and narrowing: if the argument is a logic variable, *case* suspends whereas *fcase* proceeds with a nondeterministic binding of the variable in one branch of the case expression (cf. Section 2.1). A flat program is called *normalized* if all arguments of constructor and function calls are variables. Any flat program can be normalized by introducing *let* expressions [1, 18]. The operational semantics is defined *only on normalized* programs in order to model sharing, whereas our type-based analysis is defined for flat programs.

Any Curry program can be translated into this flat representation.

Example 1 (Flat Curry representation). The concatenation function on lists

```
app []     ys = ys
app (x:xs) ys = x : app xs ys
```

is represented by the (normalized) flat program

```
app(xs,ys) = fcase xs of {[] -> ys, z:zs -> let a = app(zs,ys) in z:a}
```

Note that all variables occurring in the right-hand side of a function definition must occur in the left-hand side or be introduced by an enclosing let binding. In order to avoid a special declaration for logic variables, they are represented as self-circular let bindings. E.g., the expression "`let xs=xs in app(xs,[])`" introduces the logic variable `xs` in the expression "`app(xs,[])`".

Based on the principles developed in [18], [1] introduces a natural semantics of normalized flat programs. As this semantics adequately resembles the behavior of modern multi-paradigm languages like Curry [9, 15] or Toy [21], it is a good reference to show the correctness of program analyses for functional logic languages. There are some special properties of this semantics we have to consider in order to examine our type/effect analysis.

The only difference we have to consider is the treatment of circular data structures which are allowed in [1]. Since the nondeterminism analysis of [14] as well as ours do not consider such structures, we restrict the set of permissible programs to those without circular data structures. This is not a restriction in practice since the current definitions of Curry [9, 15] or Toy [21] do not support such structures. Note that the definition of infinite data structures is still possible since they can be defined by functions, e.g., "`repeat x = x : repeat x`".

Definition 1 (Cycle restriction). *The set of programs $P^{\otimes}$ is defined exactly like P except for the definition of let-clauses: For any expression let $x = e$, if x occurs in e then $e = x$.*

This definition allows only non-circular let-expressions with the single exception being logic variables defined by "`let x=x`".

Having defined the set of programs we want to examine, we now turn to the semantics of these programs. In contrast to an operational semantics based on term rewriting (e.g., [2, 15]), the semantics considered here correctly models sharing of common subterms as necessary for optimal evaluation and done in implementations. Sharing is modeled by introducing *heaps*. A heap, here denoted by Γ, Δ, or Θ, is a finite partial mapping from variables to expressions (the *empty heap* is denoted by `[]`). The value associated to variable x in heap Γ is denoted by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'[x] = e$ (in the rules, this notation is used as a condition as well as an update of a heap). A logic variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *value* v is a constructor-rooted term $c(\overline{x_n})$ (i.e., a term whose outermost function symbol is a constructor symbol) or a logic variable (w.r.t. the associated heap). ρ represents a substitution of variables in expressions by other variables, i.e., ρ is a renaming.

The natural semantics uses judgements of the form "$\Gamma : e \Downarrow \Delta : v$" which are interpreted as: "In the context of heap Γ, the expression e evaluates to value v and produces a new heap Δ." Figure 2 shows the rules defining this semantics (also called big-step semantics) of normalized flat programs, where the current program P is considered as a global constant. The rules VarCons and VarExp

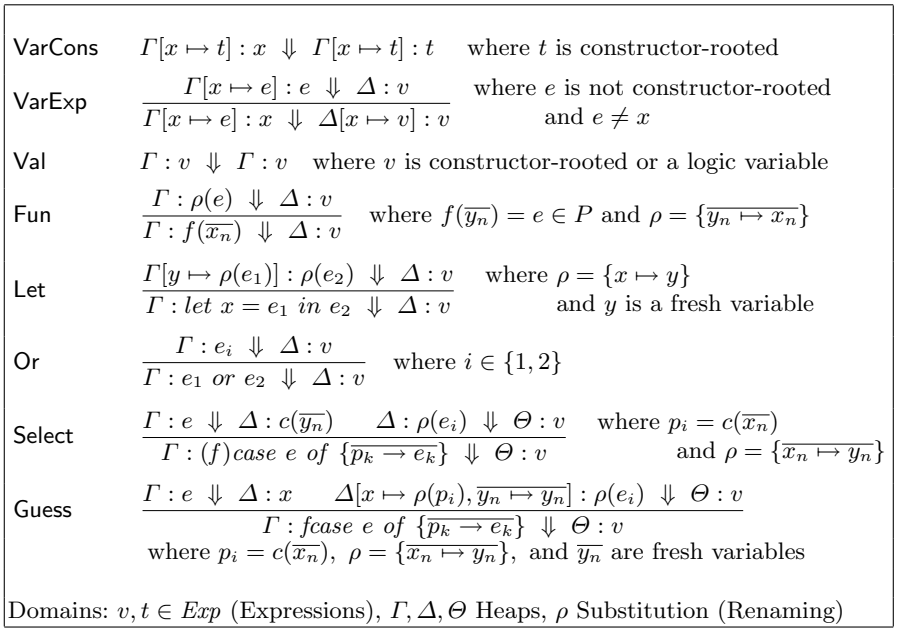

Fig. 2. Natural semantics of *normalized* flat programs [1]

are responsible to retrieve expressions from the heap, the difference being that VarCons retrieves values, whereas the expressions retrieved by VarExp have to be further evaluated. VarCons and Val form the base of proof trees generated by the big-step semantics. They treat values, i.e., expressions which are either logic variables or evaluated to head normal form. VarCons is merely a shortcut for applying VarExp and Val once each. The rule Let introduces a new binding for the heap, Fun is used to unfold function applications, and Or introduces a nondeterministic branching. Select and Guess deal with `case` expressions. Select determines the corresponding branch to continue with, if the first argument of `case` was reduced to a constructor rooted term. Guess treats the case that the first argument evaluates to a logic variable. If so, Guess introduces a nondeterministic branching where the logic variable is bound nondeterministically to one of the patterns of the `case`-expression. Remember that there are two kinds of `case`-expressions in flat programs. Only `fcase` (with `f` for "flexible") can introduce nondeterminism if the number of branches is greater than one. In short, `fcase` models narrowing whereas `case` is used to model the operational behavior of residuation. We often write `(f)case` to denote both kinds of cases.

The restriction to non-circular data structures introduced in Definition 1 implicates that no circular structures are produced during program execution, which is the content of Lemma 1.

Lemma 1 (Well-founded heaps). *Let Γ be a heap occurring in a derivation $[] : e \Downarrow \Delta : v$ w.r.t. a program $P \in P^{\otimes}$, and $\Gamma_0 := \Gamma$, $\Gamma_{n+1} := \hat{\Gamma} \circ \Gamma_n$ for $n \geq 0$ where $\hat{\Gamma}$ is the homomorphic extension of Γ. Then there is no non-trivial circular structure in Γ, i.e., there is no natural number n for which a variable x exists with $\Gamma_n(x) = t$ such that x occurs in t and $t \neq x$.*

As programs in $P^{\otimes}$ produce only well founded heaps, we can extract a complete substitution from the heap as follows:

Definition 2 (σ_Γ). *For a well-founded heap Γ, Γ^* is defined as the least fixpoint of $\{\Gamma_0 := \Gamma, \Gamma_{n+1} := \hat{\Gamma} \circ \Gamma_n\}$. Then σ_Γ is the substitution with $dom(\sigma_\Gamma) = dom(\Gamma)$ and $\{x \mapsto \Gamma^*(x)\}$.*

Example 2 (Substitution σ_Γ). Consider the following definition:

```
main  = let z = 3 in let y = c(z,z) in let x = f(y) in f(x)
```

Evaluating `main` yields heap $\Gamma := [][\texttt{z'} \mapsto \texttt{3}][\texttt{y'} \mapsto \texttt{c(z',z')}][\texttt{x'} \mapsto \texttt{f(y')}]$. For this heap, $\sigma_\Gamma(\texttt{z'}) = \texttt{3}$, $\sigma_\Gamma(\texttt{y'}) = \texttt{c(3,3)}$ and $\sigma_\Gamma(\texttt{x'}) = \texttt{f(c(3,3))}$.

The main purpose of Definition 1 is to ensure that the substitution σ_Γ is well defined.

2.2 Type/Effect Analysis Revisited

The basic ideas of the type/effect analysis used in this paper were first proposed in [14]. Here we use a slightly different definition (e.g., without a rule for subtyping but `let` clauses to describe sharing that is not covered in [14]) and base it on the natural semantics introduced in the previous section. The analysis uses the idea to attach to expressions and functions two kinds of information: a type to describe the ground status and an effect to describe the nondeterminism behavior. Similarly to standard types in typed functional languages, there are also typing rules that define well-typed expressions w.r.t. this type/effect system. Before defining these rules, two preliminary Definitions are needed. The analysis of a given program is always performed w.r.t. a *type environment* E which associates types/effects to functions, constructors and variables in the given program. Such an association is called *type annotation* and denoted by $s :: \overline{\tau}_n \xrightarrow{\varphi} \tau$ (resp. $s :: \tau/\varphi$ for constants or variables). Note that there may be more than one type annotation for a function or constructor. The purpose of the type inference described in Section 3 is to provide a method to derive appropriate type environments. In this section, we assume that a correct type environment (defined below) is given. In a type annotation $s :: \overline{\tau_n} \xrightarrow{\varphi} \tau$ for a function or constructor s each $\tau_{(i)}$ describes whether the corresponding argument or result of the function is a *ground* value, denoted by G, or if it might contain logic variables, and, hence, is of *any* value, denoted by A. The set of *effects* φ describes the possible causes for nondeterminism which might occur while evaluating s, if s is a function. Effects are either *or* or *guess*. The meaning of these effects is that one of the nondeterministic rules Or or Guess could be applied while evaluating an expression or function.

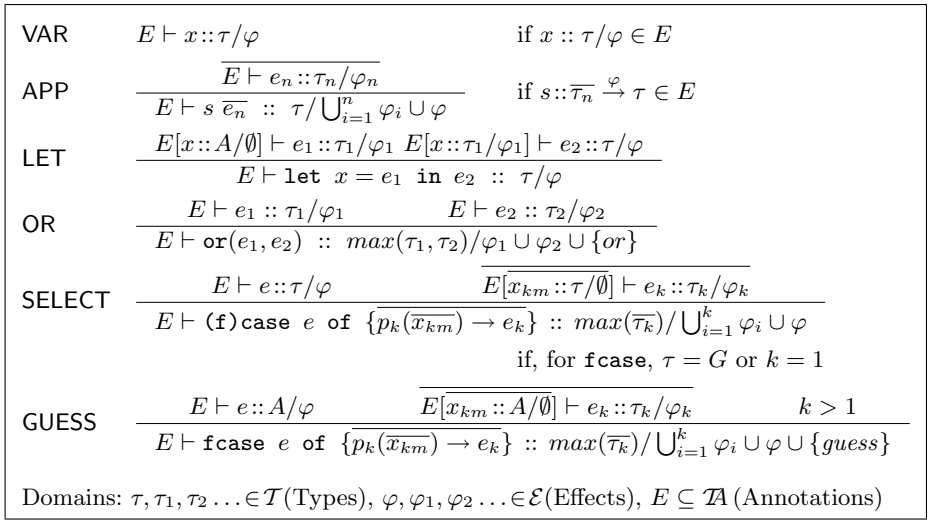

Fig. 3. Typing rules for flat expressions

Definition 3 (Type/Effects, Type Annotation, Type Environment). *The set of types $\mathcal{T}$ is defined as $\mathcal{T} = \{A, G\}$, the set of Effects $\mathcal{E}$ is defined as $\mathcal{E}$=$\{or, guess\}$, the set of type/effects for arity n is defined as $\mathcal{TE}_n$=$\{\overline{\tau_n} \xrightarrow{\varphi} \tau \mid \tau, \tau_i \in \mathcal{T}, \varphi \subseteq \mathcal{E}\}$. For $n = 0$ instead of $\xrightarrow{\varphi} \tau$ we write τ/φ. And, finally, a type environment E is a subset of the set of all type annotations $\mathcal{TA} = \{x :: \chi \mid x$ is a variable, $\chi \in \mathcal{TE}_0\} \cup \{s :: \xi \mid s \in \mathcal{F} \cup \mathcal{C}, s$ is of arity $n, \xi \in \mathcal{TE}_n\}$.*

Before defining the typing rules and giving an example, we have to introduce an ordering on the types to compare different abstract results. In general, an *ordering* is a reflexive, transitive and anti-symmetric relation.

Definition 4 (Type/effect ordering $\leq$, max, min). *$\leq$ denotes an ordering on types and effects that is the least order relation satisfying $G \leq A$ and, for effects $\varphi \leq \varphi'$ iff $\varphi \subseteq \varphi'$. Type/effects are ordered by $\tau_1 \xrightarrow{\varphi_1} \tau_2 \leq \tau_1' \xrightarrow{\varphi_2} \tau_2'$ iff $\tau_1' \leq \tau_1$, $\tau_2 \leq \tau_2'$ and $\varphi_1 \leq \varphi_2$. Furthermore, $max(\overline{\tau_k})$ (resp. $min(\overline{\tau_k})$) denotes the maximum (minimum) of the $\overline{\tau_k}$ with respect to $\leq$.*

Note the difference between argument and result in the definition of $\leq$ for functional types. Informally speaking, for functions with the same result type, it holds: the bigger the argument type, the smaller is the type of the whole function. This makes perfect sense if we think of the type as a grade of nondeterminism. A function of type $A \xrightarrow{\emptyset} G$ is more deterministic than one of type $A \xrightarrow{\emptyset} A$. However, $A \xrightarrow{\emptyset} A$ is still more deterministic than $G \xrightarrow{\emptyset} A$ because a function of the latter type might not merely map logic variables to logic variables but could introduce new ones. We are now ready to define the typing rules as given in Figure 3.

Example 3 ((In)correct type annotation). Consider the (flat) function

```
and(x,y) = fcase x of {False -> False; True -> y}
```

Correct types for `and` would be $GA \xrightarrow{\emptyset} A$ and $GG \xrightarrow{\emptyset} G$. The first type can be intuitively read as: "If the first argument is ground and the second possibly contains a logic variable, then the result may also contain a logic variable." However, $AG \xrightarrow{\emptyset} A$ is not a valid type. If the first argument is a logic variable, `fcase` will instantiate this variable nondeterministically (cf. Figure 2). Thus, the correct type for these input arguments is $AG \xrightarrow{\{guess\}} G$. The difference in the actual type check by the rules of Figure 3 is that rule SELECT is applicable for input vector GA, whereas the case AG is covered by rule GUESS.

The correctness of type annotations is now defined in two steps.

Definition 5 (Constructor-correct). *A type environment E is called correct with respect to constructor symbols, or* constructor-correct *for short, iff E contains only the types $c :: \overline{\tau_n} \xrightarrow{\emptyset} max(\overline{\tau_n})$ for any constructor symbol c.*

This definition implies that constructors do not influence the deterministic type of their arguments at all. If any argument is of type A, then the whole term is as well. Furthermore, constructors do never yield any nondeterministic effect. Constructor-correctness is a requirement for our definition of general correctness.

Definition 6 (Correctness). *A type annotation $f :: \overline{\tau_n} \xrightarrow{\varphi} \tau$ contained in a type environment E is* correct *for a definition $f(\overline{x_n}) = e$ if $E[\overline{x_n :: \tau_n/\emptyset}] \vdash e :: \tau/\varphi$. E is* correct *if it is constructor-correct and contains only correct type annotations.*

The aim of this section is to show that correct type environments correctly indicate the nondeterminism caused by the evaluation of a given function. Whenever the evaluation of a function call $f\ \overline{e_n}$ involves a nondeterministic branching by an *or* or a flexible case expression, a correct type environment must contain the corresponding type indicating the effect *or* or *guess*. And whenever the correct type environment indicates that a function f with arguments of a certain type evaluates to a ground term, then no evaluation of f with corresponding arguments yields a result containing a logic variable. The first step towards proving this correctness is the observation that expressions of the same type are indistinguishable by the type/effect system.

Lemma 2 (Substitution Lemma). *Let E be a correct type environment for a flat program. Then for each expression e holds: $E[\overline{x_n :: \tau_n/\emptyset}] \vdash e :: \tau/\varphi$ if and only if replacing each x_i (by a substitution σ) with a term e_i of the same type also yields the same type for e, i.e., $\overline{E \vdash e_n :: \tau_n/\emptyset}$ also implies $E \vdash \sigma(e) :: \tau/\varphi$. Furthermore, if some of the $\overline{e_n}$ have a non-empty effect, i.e., $E \vdash e_i :: \tau_i/\varphi_i$, then $E \vdash \sigma(e) :: \tau/\bigcup_{i=1}^{n} \varphi_i \cup \varphi$, i.e., the type τ of e remains the same but the effect inferred for e is larger.*

Lemma 2 is a typical requirement in type systems. The correctness of the type analysis is mainly based on the following theorem. We use the notation E^{free} for a type environment that extends a type environment E by annotations for free variables, i.e., if $x :: \tau/\varphi \in E$, then $x :: \tau/\varphi \in E^{free}$, otherwise $x :: A/\emptyset \in E^{free}$.

Theorem 1 (Type-descending). *Let E be a correct type environment for a non-circular program P in $P^{\otimes}$, e an expression with $\Gamma : e \Downarrow \Delta : v$ built in a proof tree for an expression $[] : e' \Downarrow \Delta' : v'$, and $E^{free} \vdash \sigma_\Gamma(e) :: \tau/\varphi$ and $E^{free} \vdash \sigma_\Delta(v) :: \tau'/\varphi'$. Then $\tau \geq \tau'$ and $\varphi \supseteq \varphi'$.*

Theorem 1 implies that the type analysis correctly indicates the evaluation of expressions to ground terms:

Corollary 1 (Correctness for ground terms). *Let E be a correct type environment for a non-circular program P in $P^{\otimes}$. If, for some expression e, $E^{free} \vdash e :: G/\varphi$ and e reduces in finitely many steps to a* value *v (i.e., a term without defined function symbols), then v is a ground term.*

The last property to prove is that the analysis is not only decreasing for types but also gathers all effects. This finally leads to the proposition that all potential effects in the evaluation of a given expression are correctly predicted.

Lemma 3 (Gathering of effects). *Let E be a correct type environment for a non-circular program P in $P^{\otimes}$. Let Γ be a well-founded heap, T be a proof tree for $\Gamma : e \Downarrow \Delta : v$ and $E^{free} \vdash \sigma_\Gamma(e) :: \tau/\varphi$. Then, for any $\Gamma' : e' \Downarrow \Delta' : v'$ in T with $E^{free} \vdash \sigma_\Gamma(e') :: \tau'/\varphi'$, $\varphi' \subseteq \varphi$ holds.*

Lemma 3 implies the final important property of the type/effect system:

Corollary 2 (Identification of nondeterminism). *If, for a non-circular program $P \in P^{\otimes}$ and expression e, there are two proof trees T and T' for $[] : e \Downarrow \Delta : v$ and $[] : e \Downarrow \Delta' : v'$ differing in more than variable names, then any type of e w.r.t. a correct type environment for P contains an effect* `or` *or* `guess`*.*

3 Type/Effect Inference

In this section we introduce a method to infer the types and effects introduced in the previous section. In order to obtain a feasible inference method, we introduce *base annotations*, a compact representation of sets of types and effects.

3.1 Base Annotations

The definition of well-typed programs is usually not sufficient. Instead one wants to compute all of the correct type environments for a given program. On a first glance, this problem seems quite hard, as for each n-ary function there are 2^{n+1} possible types even with an empty effect. However, a closer observation shows that one need only to consider $n+1$ types, namely the type where all arguments are ground (G) and the n types where a single argument is any (A) and all others are ground. The remaining types can be deduced by combining these $n+1$ base types, which we also call a *type base*. For instance, the type for $GGAGAG \xrightarrow{\varphi} \tau$ is the result of combining the type for $GGGGAG \xrightarrow{\varphi_1} \tau_1$ and $GGAGGG \xrightarrow{\varphi_2} \tau_2$. Before introducing the compact representation of type/effects, we first show the soundness of this combination of two types.

Definition 7 (Supremum $\sqcup$, $\tau/\varphi_1/\varphi_2$). *For types $\tau_1, \tau_2 \in \mathcal{T}$ the type $\tau_1 \sqcup \tau_2$ is their supremum, i.e. $max(\tau_1, \tau_2)$. For type/effects $\overline{\tau_n} \xrightarrow{\varphi_1} \tau, \overline{u_n} \xrightarrow{\varphi_2} u \in \mathcal{TE}_n$, the type/effect $(\overline{\tau_n} \xrightarrow{\varphi_1} \tau) \sqcup (\overline{u_n} \xrightarrow{\varphi_2} u)$ denotes $\overline{max(\tau_n, u_n)} \xrightarrow{\varphi_1 \sqcup \varphi_2} max(\tau, u)$. For type environments $E_1, E_2 \subseteq \mathcal{TA}$, $E_1 \sqcup E_2 = \{s :: \xi \sqcup \xi' \mid s \in Var \cup \mathcal{C} \cup F, s :: \xi, s :: \xi' \in E_1 \cup E_2\}$. Finally, $\bigsqcup x$ denotes the supremum of a set x and the notation $\tau/\varphi/\varphi'$ is used to denote $\tau/\varphi \cup \varphi'$.*

Lemma 4 (Compositionality). *If, for any function declaration $f\ \overline{x_n} = e$, there are correct type annotations $\mathcal{A}_1 = \overline{\tau_n} \xrightarrow{\varphi_1} \tau$ and $\mathcal{A}_2 = \overline{u_n} \xrightarrow{\varphi_2} u$ for environments E_1 and E_2, respectively, then $\mathcal{A}_1 \sqcup \mathcal{A}_2$ is also a correct type for f for the environment $E_1 \sqcup E_2$.*

Lemma 4 ensures that every correct type can be easily derived from a correct type base, i.e., a set containing the $n + 1$ basic types as mentioned above. This fact is the basis for the compact representation of correct type environments. Instead of an exponential number of types, it is sufficient to consider only the $n + 1$ elements of a type base. Furthermore, we can pack the information of the type base into a single structure with at most n elements, which we call a *base annotation* for a function. A base annotation for a function f is either $\underline{A}$ (or $\underline{G}$) if the result of f is of type A (or G) regardless of the types of its arguments, or it is a term indicating which arguments influence the type of f. For instance, if f has type A whenever its first argument is of type A, then the base annotation for f is Π^1 (Π denotes a kind of projection). If f has type A whenever either its second or its fourth argument is A, the annotation for f is $\Pi^2 \sqcup \Pi^4$. To determine the type for a given application of f, the Πs are replaced by the actual types of the corresponding arguments. For complex annotations, like $\Pi^2 \sqcup \Pi^4$, the result type is the supremum of the replacements. Therefore, we reuse the symbol $\sqcup$ although it is used here as a term constructor for base annotations. Furthermore, the effect *guess* is extended by a base annotation, e.g., $guess(\Pi^1)$. The reason for this will be explained soon.

Definition 8 (Syntax of base annotations). *Let $s \in \mathcal{C} \cup F$ be of arity n. Then the set of well formed base types for s, $\mathcal{BT}_s$, is the smallest set satisfying: $(\{\underline{G}, \underline{A}, \Pi^1, \ldots, \Pi^n\} \subseteq \mathcal{BT}_s) \wedge (\nu, \mu \in \mathcal{BT}_s \Rightarrow \nu \sqcup \mu \in \mathcal{BT}_s)$. The set of well formed base effects $\mathcal{BE}_s$ for s is the smallest set satisfying: $(\{\underline{or}, \underline{guess}\} \subseteq \mathcal{BE}_s) \wedge (\nu \in \mathcal{BT}_s \Rightarrow \underline{guess}(\nu) \in \mathcal{BE}_s)$. The set of well formed base annotations $\mathcal{BA}_s$ is defined as $\mathcal{BA}_s = \{s :: \nu/\varepsilon \mid \nu \in \mathcal{BT}_s, \varepsilon \in \mathcal{BE}_s\}$. We also use $\mathcal{BT}$, $\mathcal{BE}$, $\mathcal{BA}$ (without index) to denote the set of all base types, effects, annotations.*

Example 4 (Some correct base annotations).
-For each n-ary constructor c: $\underline{c} = \Pi^1 \sqcup \ldots \sqcup \Pi^n/\emptyset$ if $n > 0$, otherwise $\underline{c} = \underline{G}/\emptyset$

`-f1 x = 1`	`f1` :: $\underline{G}/\emptyset$
`-f2 = let x=x in x`	`f2` :: $\underline{A}/\emptyset$
`-f3 x y = y`	`f3` :: $\Pi^2/\emptyset$
`-f4 x y = fcase x of {1->1; 2->y}`	`f4` :: $\Pi^2/\{\underline{guess}(\Pi^1)\}$

Function `f4` also illustrates the meaning of a guess effect depending on a type. The rule Guess of the natural semantics (Figure 2) will only be applied if the

first argument of `f4` is a logic variable. Therefore, $\underline{guess}(\Pi^1)$ will yield the effect $\underline{guess}$ only if Π^1 is replaced by type $\underline{\mathrm{A}}$ and no effect if it is replaced by type $\underline{\mathrm{G}}$.

The general meaning of base annotations is best conveyed by defining the set of type/effects each of them represents. In the next section we will show how to compute base annotations for a given program. For both purposes, we need the notion of a normal form for base annotations as a means to effectively decide the equivalence on base annotations. The normal forms are obtained by rewriting with the following set of confluent and terminating rewrite rules.

Definition 9 (Normal form $\lfloor\nu/\varepsilon\rfloor$). *We denote by $\lfloor\nu\rfloor$ and $\lfloor\varepsilon\rfloor$ the simplification of base type ν and base effect ε, respectively, with the rules*

$$\begin{array}{rr} \underline{G}\sqcup\nu\to\nu & \nu\sqcup\underline{G}\to\nu \\ \underline{A}\sqcup\nu\to\underline{A} & \nu\sqcup\underline{A}\to\underline{A} \\ \Pi^i\sqcup\Pi^j\to\Pi^j\sqcup\Pi^i, i>j & \\ \nu\sqcup\nu\to\nu & \end{array} \qquad \begin{array}{r} \{\underline{guess}(\underline{G})\}\to\{\} \\ \underline{guess}(\underline{A})\to\underline{guess} \\ \underline{guess}(\nu)\to\underline{guess}(\lfloor\nu\rfloor) \end{array}$$

(the simplification rules for guess become applicable after the transformation shown in the subsequent definition, where the last rule only maintains the sorting of the Π by index). Similarly, $\lfloor\nu/\varepsilon\rfloor$ denotes component-wise simplification.

As motivated above, the base annotations of a given function represents all of its (minimal) types. The following definition describes this representation in detail.

Definition 10 (Base annotations and types). *Let f be an n-ary function. To each base annotation b for f we associate a set of type annotations $types(n,\lfloor b\rfloor)$:*

$$types(n,\underline{G}/\varepsilon) = \{\overline{\tau_n} \overset{eff(\overline{\tau_n},\varepsilon)}{\to} G \mid \overline{\tau_n}\in\mathcal{T}\}$$
$$types(n,\underline{A}/\varepsilon) = \{\overline{\tau_n} \overset{eff(\overline{\tau_n},\varepsilon)}{\to} A \mid \overline{\tau_n}\in\mathcal{T}\}$$
$$types(n,\Pi^{i_1}\sqcup\ldots\sqcup\Pi^{i_j}/\varepsilon) =$$
$$\bigsqcup(\{G^n \overset{eff(G^n,\varepsilon)}{\to} G\}\cup\{\underbrace{G^{k-1}AG^{n-k}}_{\tau_k} \overset{eff(\tau_k,\varepsilon)}{\to} A \mid k\in\{i_1\ldots i_j\}\})$$

where G^j is the usual notation for a sequence of Gs with length j and $eff(\overline{\tau_n},\varepsilon) = \lfloor\{\overline{\Pi^n\mapsto\tau_n}\}\varepsilon\rfloor$.[1]

Example 5 (Continuing Example 4). The types associated with the base annotations from Example 4 are:
-For a unary constructor $c: types(1,\Pi^1/\emptyset) = \{G\overset{\emptyset}{\to}G, A\overset{\emptyset}{\to}A\}$
`-f1:` $types(1,\underline{G}/\emptyset) = \{G\overset{\emptyset}{\to}G, A\overset{\emptyset}{\to}G\}$
`-f2:` $types(0,\underline{A}/\emptyset) = \{A/\emptyset\}$
`-f3:` $types(2,\Pi^2/\emptyset) = \{GG\overset{\emptyset}{\to}G, GA\overset{\emptyset}{\to}A, AG\overset{\emptyset}{\to}G, AA\overset{\emptyset}{\to}A\}$
`-f4:` $types(2,\Pi^2/\{\underline{guess}(\Pi^1)\}) = \{GG\overset{\emptyset}{\to}G, GA\overset{\emptyset}{\to}A, AG\overset{\{guess\}}{\to}G, AA\overset{\{guess\}}{\to}A\}$

[1] $\{\overline{\Pi^n\mapsto\tau_n}\}\varepsilon$ denotes the replacement of all occurrences of Π^i by τ_i in ε for $i\in\{1,\ldots,n\}$.

This representation of groundness information has some similarities to the domain *Prop* of propositional formulas used in groundness analysis of logic programs [5]. However, we are interested in covering all sources of nondeterminism which is usually the effect non-ground function arguments (apart from function definitions with overlapping right-hand sides, represented by *or*). Therefore, we use *projections* Π^i in the base annotations to associate potential nondeterministic behavior to the instantiation of particular arguments.

Finally, we define an ordering on base annotations. This ordering is used to define the type inference in the next section and show its correctness. For the latter purpose, it is important to note that the order is finite.

Definition 11 (Ordering on base annotations $\sqsubseteq$). *The ordering $\sqsubseteq$ is used on base types, base effects, base annotations and sets of base annotations (base environments). It is defined as the least ordering satisfying*

- $\underline{G} \sqsubseteq \nu$ *and* $\nu \sqsubseteq \underline{A}$ *for all* $\nu \in \mathcal{BT}$
- $\Pi^{i_1} \sqcup \ldots \sqcup \Pi^{i_m} \sqsubseteq \Pi^{j_1} \sqcup \ldots \sqcup \Pi^{j_n}$ *if* $\{\Pi^{i_1}, \ldots, \Pi^{i_m}\} \subseteq \{\Pi^{j_1}, \ldots, \Pi^{j_n}\}$
- $\underline{guess}(\nu) \sqsubseteq \underline{guess}$ *and* $\underline{guess}(\nu) \sqsubseteq \underline{guess}(\nu')$ *if* $\nu \sqsubseteq \nu'$ *for all* $\nu, \nu' \in \mathcal{BT}$
- $\overline{For}\ \varepsilon, \varepsilon' \in \mathcal{BE}$*:* $\varepsilon \sqsubseteq \varepsilon'$ *if* $\overline{\forall x \in \varepsilon\ \exists x' \in \varepsilon' : x \sqsubseteq x'}$
- *Ordering on base type/effects:* $\nu/\varepsilon \sqsubseteq \nu'/\varepsilon'$ *if* $\lfloor \nu \rfloor \sqsubseteq \lfloor \nu' \rfloor$ *and* $\lfloor \varepsilon \rfloor \sqsubseteq \lfloor \varepsilon' \rfloor$
- *Ordering on base environments:* $B \sqsubseteq B'$ *if* $\forall x \in B\ \exists x' \in B' : x \sqsubseteq x'$

$\sqcup$ *(resp.* $\bigsqcup$*) denotes the* $\sqsubseteq$*-supremum of two (resp. a set of) base annotations.*

3.2 Inferring Base Annotations

After having defined the structure of base annotations, we are ready to define the inference of them. Figure 4 shows the rules to infer base annotations for a given expression. The complete inference is defined as a fix-point iteration on a given flat program. Before we can define the iteration, we need to observe that the inference is monotone, i.e., the inference always computes greater types for greater environments (with respect to $\sqsubseteq$).

Lemma 5 ($\rhd$ respects $\sqsubseteq$). *Let B and B' be two base environments with $B \sqsubseteq B'$. Then, for each e with $B \rhd e :: \nu/\varepsilon$, there is a derivation $B' \rhd e :: \nu'/\varepsilon'$ with $\nu/\varepsilon \sqsubseteq \nu'/\varepsilon'$.*

Because of the monotonicity of $\rhd$, we can define the inference of a base environment as follows:

Definition 12 (Type inference). *The mapping Inf associates to a flat program P a type environment. It is defined by the following fix-point iteration based on the inference system in Figure 4:*

$$\begin{aligned}
Inf_0(P) &= \{c :: \underline{G}/\emptyset \mid c \text{ is a 0-ary constructor}\} \cup \\
&\quad \{c :: \Pi^1 \sqcup \ldots \sqcup \Pi^n/\emptyset \mid c \text{ is an } n\text{-ary constructor}, n > 0\} \cup \\
&\quad \{f :: \underline{G}/\emptyset \mid f \text{ is a defined function}\} \\
Inf_{i+1}(P) &= \{f :: \lfloor \nu/\varepsilon \rfloor \mid f\ \overline{x_n} = e \in P, Inf_i(P)[\overline{x_n :: \Pi^n/\emptyset}] \rhd e :: \nu/\varepsilon\} \\
Inf(P) &= Inf_j(P), \text{ if } j \in \mathbb{N} \text{ is smallest with } Inf_j(P) = Inf_{j+1}(P)
\end{aligned}$$

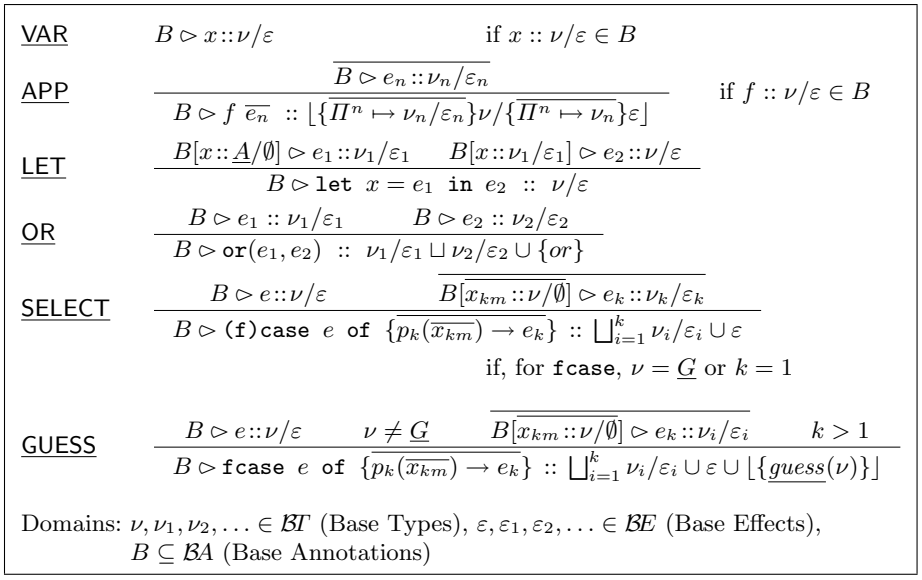

Fig. 4. Inference rules

After proving that $Inf(P)$ is indeed well defined, we will give examples for inferring types for a given program.

Lemma 6 (Type increase). *Let P be a flat program and f a function defined in P. If $f :: \nu/\varepsilon \in Inf_i(P)$ and $f :: \nu'/\varepsilon' \in Inf_{i+1}(P)$, then $\nu/\varepsilon \sqsubseteq \nu'/\varepsilon'$.*

Corollary 3 ($Inf(P)$ is well defined). *For each finite program P there is a natural number n with $Inf_n(P) = Inf_{n+1}(P)$.*

Corollary 3 states that the iteration of the inference finally terminates.

Example 6 (Type inference). As an example for the type inference, consider the flat program (c_0, c_1 are constructors of arity $0, 1$):

$$P = \begin{cases} f_1(x) & = \texttt{fcase}\ x\ \texttt{of}\ \{c_0 \to g, c_1(y) \to f_1(y)\} \\ f_2(x, y) & = f_1(y) \\ g & = \texttt{let}\ x = x\ \texttt{in}\ x \end{cases}$$

Remember that "$\texttt{let}\ x = x$" defines a logic variable x so that g evaluates to a new logic variable. The type environments are computed by the iterations:

$Inf_0(P) = \{c_0 :: \underline{G}/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: \underline{G}/\emptyset, f_2 :: \underline{G}/\emptyset, g :: \underline{G}/\emptyset\}$
$Inf_1(P) = \{c_0 :: \underline{G}/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: \underline{G}/\{\underline{guess}(\Pi^1)\}, f_2 :: \underline{G}/\emptyset, g :: \underline{A}/\emptyset\}$
$Inf_2(P) = \{c_0 :: \underline{G}/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: \underline{A}/\{\underline{guess}(\Pi^1)\}, f_2 :: \underline{G}/\{\underline{guess}(\Pi^2)\}, g :: \underline{A}/\emptyset\}$
$Inf_3(P) = \{c_0 :: \underline{G}/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: \underline{A}/\{\underline{guess}(\Pi^1)\}, f_2 :: \underline{A}/\{\underline{guess}(\Pi^2)\}, g :: \underline{A}/\emptyset\}$
$Inf(P) = Inf_3(P)$

The inference shows that a call to f_2 might produce a non-ground result but causes nondeterministic steps only if the second argument is non-ground.

To complete this section about the type inference, we show that its computed results correctly and completely correspond to the results of the type/effect analysis of Section 2.

Theorem 2 (Correctness of the inference). *Let P be a flat program, $E(P) = \{s :: \overline{\tau_n} \xrightarrow{\varphi} \tau \mid s$ is n-ary, $s :: \nu/\varepsilon \in Inf(P), \overline{\tau_n} \xrightarrow{\varphi} \tau \in types(n, \nu/\varepsilon)\}$, and E be a correct environment for P. Then:*

Soundness: *$E(P)$ is a correct environment in the sense of Definition 6.*
Completeness: *If $\mathcal{A} \in E$ is a type annotation, then $E(P)$ contains a type annotation $\mathcal{A}'$ with $\mathcal{A}' \leq \mathcal{A}$ (cf. Definition 4).*

4 Conclusions

We have presented a program analysis to approximate the nondeterminism behavior of functional logic programs. Unlike existing nondeterminism analyses for logic languages, we have considered a language with a demand-driven evaluation strategy. Such a strategy has good properties for executing (e.g., optimal evaluation [2]) and writing programs (e.g., more modularity due to the use of infinite data structures [17]), it considerably complicates the analysis of programs since, in contrast to logic languages with an eager evaluation model (e.g., Prolog, Mercury, HAL), there is no direct correspondence between the program structure and its evaluation order. Therefore, we have abstracted the information about the run-time behavior of the program in form of a non-standard type and effect system. The program analysis is then an iterative type inference process based on a compact structure to represent sets of types and effects.

For future work we plan to improve the preliminary implementation of the type inference and apply it to larger application programs. Furthermore, we are working on a compilation for the functional logic language Curry [9, 15] into the functional language Haskell [23]. This compilation should take great advantage of the presented analysis.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *JSC*, Vol. 40, No. 1, pp. 795–829, 2005.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
3. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of FLOPS 2002*, pp. 67–87. Springer LNCS 2441, 2002.
4. B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming*, No. 6, 2004.
5. A. Cortesi, G. File, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. IEEE Symposium on Logic in Computer Science*, pp. 322–327, 1991.

6. S.K. Debray and D.S. Warren. Detection and Optimization of Functional Computations in Prolog. In *Proc. Third International Conference on Logic Programming (London)*, pp. 490–504. Springer LNCS 225, 1986.
7. B. Demoen et al. Herbrand constraint solving in HAL. In *Proc. of ICLP'99*, pp. 260–274. MIT Press, 1999.
8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
9. M. Hanus. A Unified Computation Model for Functional and Logic Programming. *Proc. 24th ACM Symp. on Principles of Programming Languages*, pp. 80–93, 1997.
10. 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.
11. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
12. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, Vol. 9, No. 1, pp. 33–75, 1999.
13. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
14. M. Hanus and F. Steiner. Type-based Nondeterminism Checking in Functional Logic Programs. In *Proc. of PPDP 2000*, pp. 202–213. ACM Press, 2000.
15. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
16. F. Henderson, T. Somogyi, Z. Conway. Determinism analysis in the Mercury compiler. In *Proc. 19th Australian Computer Science Conference*, pp. 337–346, 1996.
17. J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pp. 17–42. Addison Wesley, 1990.
18. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, pp. 144–154. ACM Press, 1993.
19. F. Liu. Towards lazy evaluation, sharing and non-determinism in resolution based functional logic languages. In *Proc. of FPCA'93*, pp. 201–209. ACM Press, 1993.
20. R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Theoretical Computer Science 142*, pp. 59–87, 1995.
21. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
22. F.Nielson, H.R.Nielson, C.Hankin. *Principles of Program Analysis*. Springer, 1999.
23. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
24. P. Van Roy, B. Demoen, and Y.D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection, and determinism. In *Proc. of the TAPSOFT '87*, pp. 111–125. Springer LNCS 250, 1987.
25. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.