Nondeterminism Analysis of Functional Logic Programs*

Draft of May 13, 2005

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 to the programmer. 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 non-deterministic search for solutions. This combination leads to optimal evaluation strategies [2] and new design patterns [3] that can be applied to provide better programming abstractions, e.g., for implementing graphical user interfaces [10] or programming 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 that should be applied to the

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

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, 14] for the declarative multi-paradigm language Curry, otherwise a run-time 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. Another motivation for such an analysis is their use in optimizing programs. 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 (which is the case for many application programs, e.g., dynamic web pages [11] where logic variables are deterministically instantiated).

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 depend on the demandedness of argument evaluation (see also [15]). Therefore, analyses for languages like Prolog [24, 6], Mercury [17], 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).

In this paper we present a static analysis of functional logic programs with a demand-driven evaluation strategy. The analysis derives information about the nondeterminism behavior of defined functions and has the form of a type/effect system (see [22]). Such systems can be seen as extensions of classical type systems known from functional languages. In our analysis the types hold 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 [15]. 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 [15]. Furthermore, we present a new method to *infer* types and effects (type inference was not covered in [15]) 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.

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

Fig. 1. Syntax of flat programs

2 The Type/Effect Analysis

In this section we define a type/effect system based on the ideas in [15] and show its correctness w.r.t. the operational semantics of functional logic 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 [13] 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, 16] 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 $o_1, \ldots, o_k$. 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 nondeterministically binding of the variable in one branch of the case expression (see also Section 2.2). A flat program is called *normalized* if the arguments of constructor and function calls are always variables. Any flat program can be normalized by introducing new variables by *let* expressions [1]. 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. For instance, the concatenation function on lists

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

is represented by the (normalized) flat program

```
append xs ys = fcase xs of {[]   -> ys,
                            z:zs -> let {a = append 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 are introduced by an enclosing let binding. In order to avoid a special declaration for logic variables, they are represented as self-circular let bindings. For instance, the expression

```
let {xs=xs} in append xs [3,4]
```

introduces the logic variable `xs` in the expression "`append xs [3,4]`".

2.2 Natural Semantics of Functional Logic Programs

[1] introduces a natural semantics of flat programs. As it adequately resembles the behavior of modern multi-paradigm languages like Curry [9, 16] 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 [15] as well as ours do not consider such structures, we restrict the set of permissible programs to those without circular data structures. Note that this is not a restriction in practice since the current definitions of Curry [9, 16] or Toy [21] do not support such structures.

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 $\{\overline{x_n = e_n}\}$ there must exist an ordering ord of the equations $\overline{x_n = e_n}$, i.e., a one-on-one mapping between the equations and the numbers $1, \ldots, n$ such that: For each $x_i = e_i$ the variable x_i does not appear in any equation $x_j = e_j$ with $ord(x_j = e_j) > ord(x_i = e_i)$. Furthermore, if x_i appears in e_i then $e_i = x_i$.*

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

Example 1. Consider the following definitions:

```
main  = let {x = f y, y = c z z, z = 3} in f x
main' = let {x = f y, y = c z z, z = f x} in f x
```

The `let` expression in function `main` can be ordered as

```
let {z = 3, y = c z z, x = f y} in f x
```

whereas the one in `main'` is circular.

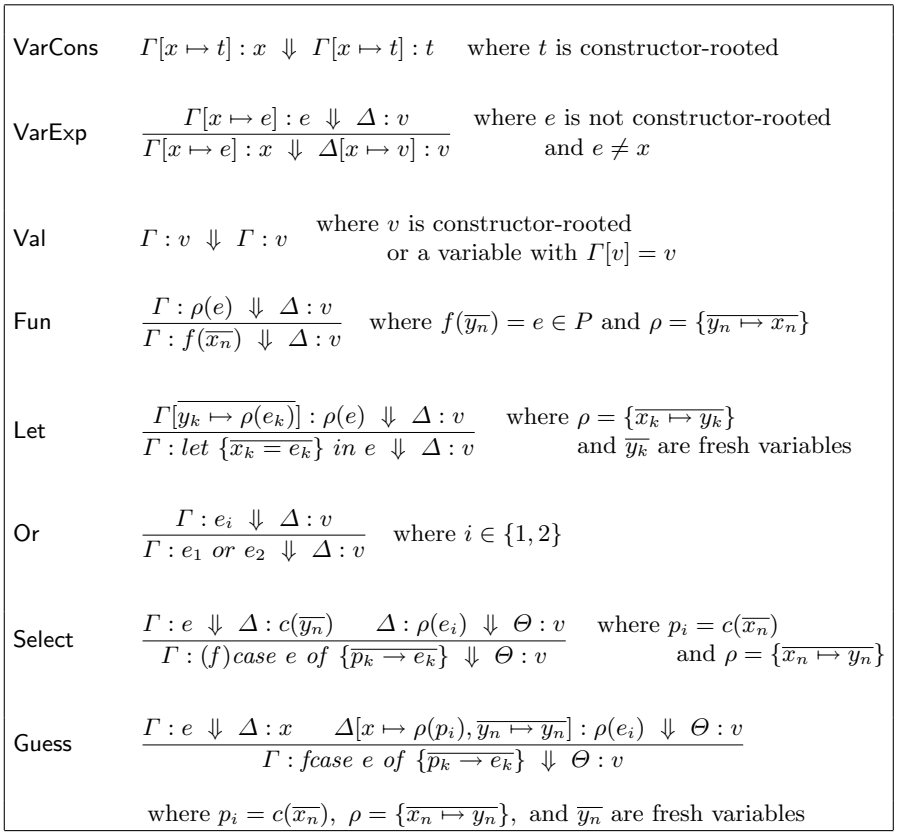

Fig. 2. Natural semantics of normalized flat programs

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, 16]), 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 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 with $\Gamma[x] = e$ and can be read as "in Γ, x points to 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 $c(\overline{e_n})$ (i.e., a term whose outermost function symbol is a constructor symbol) or a logic variable (w.r.t. the associated heap).

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 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 new bindings 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 non-deterministically 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 proves to be quite convenient. Using it, we can extract a complete substitution from the heap by simply taking its transitive closure.

Definition 2 (σ_Γ). *For a heap Γ the substitution σ_Γ is defined as the transitive closure of Γ.*

Unfolding `main` of Example 1 yields the heap $\Gamma[\mathtt{x} \mapsto \mathtt{f\ y}, \mathtt{y} \mapsto \mathtt{c\ z\ z}, \mathtt{z} \mapsto \mathtt{3}]$. For this heap, $\sigma_\Gamma(\mathtt{z}) = \mathtt{3}$, $\sigma_\Gamma(\mathtt{y}) = \mathtt{c\ 3\ 3}$ and $\sigma_\Gamma(\mathtt{x}) = \mathtt{f(c\ 3\ 3)}$.

Note that the main purpose of Definition 1 is to ensure that the substitution σ_Γ cannot substitute a variable with an infinite term if Γ belongs to an execution of a program in $P^\otimes$. This is the content of the next Lemma 1. First, we define the notion of a heap belonging to an execution of a program in $P^\otimes$.

Definition 3 (Natural heap). *A heap Γ is a* natural heap *if it was build in the derivation of the natural semantics of some expression e, i.e., Γ is part of a proof tree for $[] : e \Downarrow \Delta : v$.*

Lemma 1 (Well-founded heaps). *Let Γ be a natural heap for $[] : e \Downarrow \Delta : v$ w.r.t. program $P \in P^\otimes$, and $\Gamma^1 := \Gamma$, $\Gamma^n := \Gamma \circ \Gamma^{n-1}$ for $n > 1$. Then there is no non-trivial circular structure in Γ, i.e., there is no natural number n for which an x exists with $\Gamma^n(x) = t$ such that x occurs in t and $t \neq x$.*

Proof: To see this, one has to consider the "heap-building" rules VarExp, Let, and Guess. In case of Let, the referencing variables are fresh and, by definition of $P^\otimes$, circular structures cannot be formulated by `let`-expressions. In case of Guess, the only variable already used in the proof tree is x. x is mapped to $\rho(p_i)$,

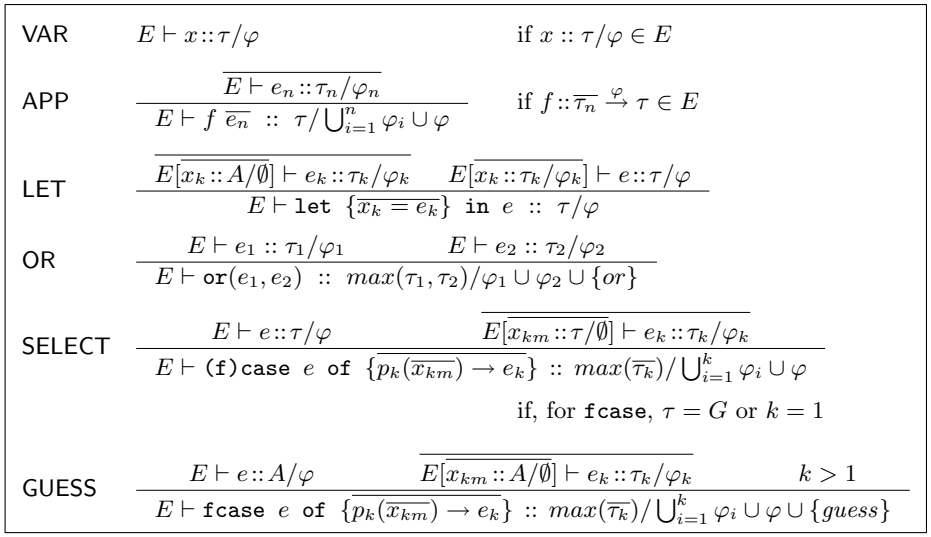

Fig. 3. Typing rules for flat expressions

which, by definition of ρ and p_i, cannot contain x. In case of VarExp, the new reference for x is on the right hand side of $\Downarrow$. As there is no rule to transfer heap references from the right to the left, the heap on the right-hand side is built from Let and Guess rules and, thus, contains no cycles. □

This lemma concludes our consideration of the natural semantics. We turn to our main object: the type/effect analysis.

2.3 Type/Effect Analysis Revisited

The basic ideas of the type/effect analysis used in this paper were first proposed in [15]. 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 [15]) 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. These rules are shown in Figure 3. The analysis of a given program is always performed w.r.t. a *type environment* E which associates types/effects to functions and variables in the given program. Such an association is also called *type annotation* and denoted by $f :: \tau/\varphi \in E$ (note that there may be more than one type annotation for a function). The purpose of the type inference described in Section 3 is the provide a method to guess appropriate type environments. In this section, we assume that a correct type environment (see below) is given.

A type annotation for a function f is of the form $f :: \overline{\tau_n} \xrightarrow{\varphi} \tau$. 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 f. As can be seen in Figure 3, each effect is 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. For instance, consider the following function:

```
and False y = False
and True  y = y
```

Its flat form is

```
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, where whereas the case AG is covered by rule GUESS.

Before defining the notion of correct type annotations, we introduce an ordering on the types in order to enable the comparison of different abstract results. In general, an *ordering* is a reflexive, transitive and anti-symmetric relation.

Definition 4 (Type/effect ordering $\leq$). *$\leq$ denotes an ordering on types and effects that is the least order relation satisfying $G \leq A$ and, for effects ϕ, ϕ', $\phi \leq \phi'$ iff $\phi \subseteq \phi'$. Types with effects are ordered by $\tau/\phi \leq \tau'/\phi'$ iff $\tau \leq \tau'$ and $\phi \leq \phi'$. For functional types $\tau_1 \xrightarrow{\phi} \tau_2 \leq \tau_1' \xrightarrow{\phi'} \tau_2'$ iff $\tau_1' \leq \tau_1$, $\tau_2 \leq \tau_2'$ and $\phi \leq \phi'$. Finally, for annotations we write $f :: \tau_1 \leq f :: \tau_2$ iff $\tau_1 \leq \tau_2$.*

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.

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 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 of the arguments of a constructor 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 $E[\overline{x_n :: \tau_n/\emptyset}] \vdash e :: \tau/\varphi$. A type environment 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 $f\ \overline{e_i}$ 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 with a term e_i of the same type (via a substitution σ) $E \vdash e_i :: \tau_i/\emptyset$ still yields $E \vdash \sigma(e) :: \tau/\varphi$. If some of the $\overline{e_n}$ have a non-empty effect, i.e., $E \vdash e_i :: \tau_i/\varphi_n$, 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.*

Proof: To verify the claim, one has to see that all inference rules are defined by induction over the term structure. In the antecedent of each rule, the type of subterm position is inferred. This finally leads to inferring the type of each x_i or e_i, respectively, treating both identically. Thereby possible effects are gathered faithfully, i.e., no effect is ever deleted and all effects of subterms are gathered. □

Lemma 2 is a typical requirement in type systems. The correctness of the type analysis is mainly based on the following theorem. There, 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 $P^{\otimes}$, 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 $\varphi \supseteq \varphi'$.*

Proof: Proof by induction over the depth of the proof tree for $\Gamma : e \Downarrow \Delta : v$.
Base case: The proof tree is of length one. Then it can only consist of an application of the rules VarCons or Val. In case of VarCons, $\sigma_{\Gamma[x\mapsto t]}(x) = \sigma_{\Gamma[x\mapsto t]}(t)$, since σ_Γ is idempotent, i.e. $= \sigma_\Gamma \circ \sigma_\Gamma$. In case of Val, the two terms related via $\Downarrow$ are identical as well.
Inductive case: Suppose that the claim holds for all proof trees of a depth $\leq d$. We have to show that it holds for all proof trees with depth $d+1$. We consider the rule of the last step of the proof construction and distinguish the rule applied there:
(VarExp): By induction hypothesis, the claim holds for $\sigma_{\Gamma[x\mapsto e]}(e)$ and $\sigma_\Delta(v)$. Analogously to (VarCons) above, $\sigma_{\Gamma[x\mapsto e]}(x)$ is equal to $\sigma_{\Gamma[x\mapsto e]}(e)$. Furthermore, proof trees for any $[] : e' \Downarrow \Delta' : v'$ cannot yield circular heap structures (Lemma 1). Hence, $\sigma_{\Delta[x\mapsto v]}(v) = \sigma_\Delta(v)$.
(Fun): By induction hypothesis, the claim holds for $\sigma_\Gamma \circ \rho(e)$ and $\sigma_\Delta(v)$. Hence we have to show that for $E^{free} \vdash \sigma_\Gamma(f(\overline{x_n})) :: \tau/\varphi$ and $E^{free} \vdash \sigma_\Gamma \circ \rho(e) :: \tau'/\varphi'$ holds $\tau \geq \tau'$ and $\varphi \supseteq \varphi'$. Let $\overline{\tau_n} \xrightarrow{\varphi} \tau$ be the type of f used in rule APP. By the type inference rule APP, the type of $\sigma_\Gamma(f(\overline{x_n}))$ is derived by the types of the $\overline{\sigma_\Gamma(x_n)}$, which must match the $\overline{\tau_n}$, and the type of f in E: $\overline{E^{free} \vdash \sigma_\Gamma(x_n) :: \tau_n/\varphi_n}$. As inferred by the definition of correct type annotations, the result type τ of f matches the type of e derived from $E[\overline{x_n :: \tau_n/\emptyset}]$. By the Substitution Lemma replacing the variables $\overline{x_n}$ by terms of the same type does not change the type derived for e and hence $\tau = \tau'$ and $\varphi \supseteq \varphi'$.
(Let): As the induction hypothesis holds for the antecedence of the rule Let, we have to show for $E^{free} \vdash \sigma_\Gamma(\text{let } \{\overline{x_k = e_k}\} \text{ in } e) :: \tau/\varphi$ and $E^{free} \vdash \sigma_{\Gamma[\overline{y_k \mapsto \rho(e_k)}]} \circ \rho(e) :: \tau'/\varphi'$ holds $\tau \geq \tau'$ and $\varphi \supseteq \varphi'$. By definition of the inference rule LET, the claim follows directly from the Substitution Lemma.
(Or): By induction hypothesis the claim holds for e_i and v. Because the inference rule OR infers the maximum type of $\{e_1, e_2\}$ and $e_i \in \{e_1, e_2\}$, the claim also holds for e_1 *or* e_2.
(Select): By induction hypothesis, the claim holds for $\sigma_\Gamma(e)$ and $\sigma_\Delta(c(\overline{y_n}))$, $\sigma_\Delta \circ \rho(e_i)$ and $\Theta(v)$ respectively. We have to establish a link between that two pairs via the type of the (f)case-expression. Let $\tau :: \varphi$ be the type derived for $\sigma_\Gamma((\text{f})\text{case } e \text{ of } \{\overline{p_k(\overline{x_{km}}) \to e_k}\})$, $\tau' :: \varphi'$ the type derived for $\sigma_\Gamma(e)$ and $\tau'' :: \varphi''$ the type derived for $\sigma_\Delta \circ \rho(e_i)$ from the according E^{free}. By induction hypothesis, the τ' is greater or equal than the type derived for $\sigma_\Delta(c(\overline{y_n}))$. Because c is a constructor symbol, the type of any of the $\overline{\sigma_\Delta(y_n)}$ is less or equal than the type of $\sigma_\Delta(c(\overline{y_n}))$ (Definition 5 and rule APP). In consequence, the type derived in $E[\overline{x_m :: \tau'}]$ for each of the e_i from case $p_i(\overline{x_m}) \mapsto e_i$ is greater or equal than the type derived for $\sigma_\Delta \circ \rho(e_i)$ (definition of ρ in Select and Substitution Lemma). Hence $\tau' \geq \tau''$. Since $max(\overline{\tau_k}) \geq \tau''$, the desired claim that τ is greater or equal than the type derived for $\sigma_\Theta(v)$ follows immediately.
(Guess): The consideration is identical to the one for Select with the simplifying difference that the type A for e is known beforehand. □

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 $P^{\otimes}$. 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.*

Proof: Assume that v is not a ground term, i.e., v contains free variables. By Definition 5, any correct type environment contains for a constructor symbol c only types of the form $c :: \overline{\tau_n} \xrightarrow{\emptyset} max(\overline{\tau_n})$. Thus, any correct analysis for v can only yield $v :: A/\emptyset$. As shown by Theorem 1 with $[] : e \Downarrow \Delta : v$, the type of e must be greater or equal than the type of v and, thus, could not have been G/φ for any φ. □

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 $P^{\otimes}$. Let T be a proof tree for $\Gamma : e \Downarrow \Delta : v$ where Γ is a natural heap 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.*

Proof: By induction over the depth d of T: The base case $d = 1$ trivially holds since $e = e'$.
Inductive case: Suppose that the claim holds for all proof trees of depth $\leq d$. We have to show that it holds for all proof trees with depth $d + 1$. To see this, consider the last applied rule:
VarExp: $\sigma_\Gamma(e) = \sigma_\Gamma(e')$
In case of Fun, Let, and Or, the definition of correct type annotations and the inference rules LET and OR, respectively, yield the claim along with the substitution lemma, because in all cases part of the inferable effect is defined by union of the effects of the arguments. The same is true for Select and Guess. The only difference is that an fcase with only one pattern will be subject to rule SELECT but to the rule Guess of the natural semantics. In both cases, however, effects are defined by union of the part effects. □

The previous lemma 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`.

Proof: For the natural semantics the only source of nontrivial nondeterminism are the rules Or and Guess (compare [1, Section 6]). If anywhere in T (or T') the rule Or is applied, then, by definition of the inference rule OR and Lemma 3, the effect will also be inferred for e. If anywhere in T (or T') the rule (Guess) is applied, the situation is of the form:

$$\frac{\Gamma : e_0 \;\Downarrow\; \Delta : x \qquad \Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : fcase\ e_0\ of\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$$

where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh variables.
Since x is a logic variable ($\sigma_\Gamma(x) = x$), any correct type environment E yields $E^{free} \vdash x :: A/\emptyset$. By Theorem 1, $E^{free} \vdash e_0 :: A/\varphi$. This implies that the inference rule GUESS can be applied to this situation if $k > 1$. If $k = 1$, the application of Guess is not a source of the difference between T and T'. The effect {guess}, once inferred, will be propagated down to e due to Lemma 3. □

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 $GGAGAG \to \tau$ is the result of combining the type for $GGGGAG \to \tau_1$ and $GGAGGG \to \tau_2$. We first show the soundness of this combination of two types.

Definition 7 (Supremum $\bigsqcup$, $\sqcup$, $\tau/\varphi/\varphi'$). *For two types τ and τ' the type $\tau \sqcup \tau'$ is the supremum of both types (w.r.t. the standard type ordering). Likewise, $\tau/\varphi \sqcup \tau'/\varphi'$ denotes the supremum of τ/φ and τ'/φ', i.e., $\tau \sqcup \tau'/\varphi \cup \varphi'$. For two environments E and E', $E \sqcup E'$ denotes $\bigsqcup(E \cup E')$ where $\bigsqcup x$ denotes the closure of x under supremum. Finally, the notation $\tau/\varphi/\varphi'$ is sometimes 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 $\mathcal{A} = \overline{\tau_n} \xrightarrow{\varphi} \tau$ and $\mathcal{A}' = \overline{\tau'_n} \xrightarrow{\varphi'} \tau'$ for environments E and E', respectively, then $\mathcal{A} \sqcup \mathcal{A}'$ is also a correct type for f for the environment $E \sqcup E'$.*

Proof: We show the claim by induction over the structure of e:
Base case: e is a variable $e = x$. Then x has to be one x_i of the variables of $\overline{x_n}$ and, thus, $\tau = \tau_i$ and $\tau' = \tau'_i$, $\varphi = \varphi' = \emptyset$. Since the type for x_i in $\mathcal{A} \sqcup \mathcal{A}'$ is $\tau_i \sqcup \tau'_i$, the typing of e with $\tau \sqcup \tau' = \tau_i \sqcup \tau'_i$ and $\varphi \cup \varphi' = \emptyset$ is also correct.
Inductive case: We assume that the claim holds for all terms smaller than e. We have to show that it also holds for e. We show this by a case distinction over the outermost structure of e:
e is an application $e = g\ \overline{y_m}$ where $y_i \in \{\overline{x_n}\}, i \in \{1 \dots m\}$: By the induction

hypothesis, $E \sqcup E' \vdash e_j :: \tau_j \sqcup \tau'_j / \varphi_j \cup \varphi'_j$ for each $j \in \{1 \dots k\}$. Since $E \sqcup E'$ is closed under supremum, for the two annotations used for g in E and E', the supremum of these annotations is in $E \sqcup E'$.
$e =$ let $\{\overline{x_k = e_k}\}$ in e_l: The claim follows directly from the induction hypothesis because the LET rule just takes the type of the derivation for l and forms a union of all the effects derived for e_l and all $\overline{e_k}$.
$e = or(e_1, e_2)$: The claim follows directly from the induction hypothesis since $max(\tau_1, \tau_2) = \tau_1 \sqcup \tau_2$, the effects are gathered and $\{or\}$ is part of all derived effects.
$e =$ (f)case e_c of $\{\overline{p_k \to e_k}\}$: Depending on the inferred type of e_c and whether $k > 1$, either rule SELECT or rule GUESS is applied. Either way the claim holds regarding the branches, since $max(max(\overline{\tau_k}), max(\overline{\tau'_k})) = max(\overline{\tau_k}) \sqcup max(\overline{\tau'_k})$ and all the effects are gathered in both rules. Furthermore, if rule GUESS was applied and, hence, the type A was inferred for e_c, the type of e_c in $E \sqcup E'$ must also be A, since $A \sqcup A = A$. □

Definition 8 (Type base). *Let $f\ \overline{x_n} = e$ be a function declaration. A set of type annotations $\{\mathcal{A}, \mathcal{A}_1, \dots, \mathcal{A}_n\}$ is called a* type base *for f if $\mathcal{A} = G \dots G \xrightarrow{\varphi} \tau$ and $\mathcal{A}_i = G \dots G A^i G \dots G \xrightarrow{\varphi_i} \tau_i$ (where A^i denotes that A is at the i-th position for $i \in \{1 \dots n\}$) for types τ, τ_i and effects φ, φ_i.*

Lemma 4 ensures that every correct type can be easily derived from a correct type base. This fact is the basis for the improvement mentioned at the beginning of this section. 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. This structure is defined as follows (note that $\sqcup$ is used in base annotations as a term constructor rather than the supremum function on types):

Definition 9 (Base annotation). *Let $f\ \overline{x_n} = e$ be a function declaration and B a type base for f. The* base annotation $\underline{f}$ *for f is defined as follows. Let*

$$\underline{\tau}^0 = G$$
$$\underline{\tau}^i = \begin{cases} \underline{\tau}^{i-1} & \text{if } G \dots G A^i G \dots G \xrightarrow{\varphi_i} G \in B \\ \Pi^i \sqcup \underline{\tau}^{i-1} & \text{otherwise} \end{cases}$$

$$\underline{\varphi} = \{guess(\Pi^i) \mid f :: G \dots G A^i G \dots G \xrightarrow{\varphi} \tau, guess \in \varphi\}$$

If $G \dots G \xrightarrow{\varphi_0} \tau_0 \in B$, then $\underline{f} = \tau_f / \varphi_f$ with

$$\tau_f = \begin{cases} A & \text{if } \tau_0 = A \\ \underline{\tau}^n & \text{otherwise} \end{cases} \quad \text{and} \quad \varphi_f = \begin{cases} \varphi_0 & \text{if } guess \in \varphi_0 \\ \varphi_0 \cup \underline{\varphi} & \text{otherwise} \end{cases}$$

Here are some examples for correct base annotations:
-For each n-ary constructor c: $\underline{c} = \Pi^1 \sqcup \dots \sqcup \Pi^n / \emptyset$ if $n > 0$, otherwise $\underline{c} = G/\emptyset$

```
-f1 x = 1                          : f1 = G/∅
-f2 = let {x=x} in x               : f2 = A/∅
-f3 x y = y                        : f3 = Π²/∅
-f4 x y = fcase x of {1->1; 2->y}: f4 = Π²/{guess(Π¹)}
```

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.

Later on, we will compute base annotations by a fix-point iteration. For this purpose we need to decide whether two given base annotations are equivalent. This is done by checking the equality of normal forms obtained by rewriting with the following set of confluent and terminating rewrite rules.

Definition 10 (Normal form $\lfloor\tau/\varphi\rfloor$). *We denote by $\lfloor\tau\rfloor$ and $\lfloor\varphi\rfloor$ the simplification of τ and φ, respectively, with the rules*

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

(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, $\lfloor\tau/\varphi\rfloor$ denotes component-wise simplification.

According to their definition, the base annotations of a given function can be used to infer all of its types as shown by the following definition.

Definition 11 (Base annotations and types). *Consider an n-ary function f. To each base annotation $\underline{f}$ we associate a set of type annotations $types(\underline{f})$ as follows:*

$$types(G/\varphi) = \{\overline{\tau_n} \overset{eff(\overline{\tau_n},\varphi)}{\to} G \mid \tau_i \in \{A, G\}, i \in \{1 \dots n\}\}$$
$$types(A/\varphi) = \{\overline{\tau_n} \overset{eff(\overline{\tau_n},\varphi)}{\to} A \mid \tau_i \in \{A, G\}, i \in \{1 \dots n\}\}$$
$$types(\Pi^{i_1} \sqcup \dots \sqcup \Pi^{i_j}/\varphi) =$$
$$\bigsqcup(\{\underbrace{G \dots G}_{\tau_G} \overset{eff(\tau_G,\varphi)}{\to} G\} \cup \{\underbrace{G \dots G A^k G \dots G}_{\tau_k} \overset{eff(\tau_k,\varphi)}{\to} A \mid k \in \{i_1 \dots i_j\}\})$$

where $eff(\overline{\tau_n}, \varphi) = \lfloor\{\overline{\Pi^n \mapsto \tau_n}\}\varphi\rfloor$.[1]
Straightforward, we extend this notation to sets containing base annotations, replacing each annotation β with the elements of $types(\beta)$.

Proposition 1 (Unique annotation, Definition $\beta(\overline{\tau_n})$). *For each base annotation β for an n-ary function and types $\overline{\tau_n}$ there is a unique annotation $\overline{\tau_n} \overset{\varphi}{\to} \tau$ in $types(\beta)$. We denote this annotation by $\beta(\overline{\tau_n})$.*

Proof: If $\beta = G/\varphi$ or $\beta = A/\varphi$, this annotation is $\overline{\tau_n} \overset{\varphi}{\to} G$ or $\overline{\tau_n} \overset{\varphi}{\to} A$, respectively.

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

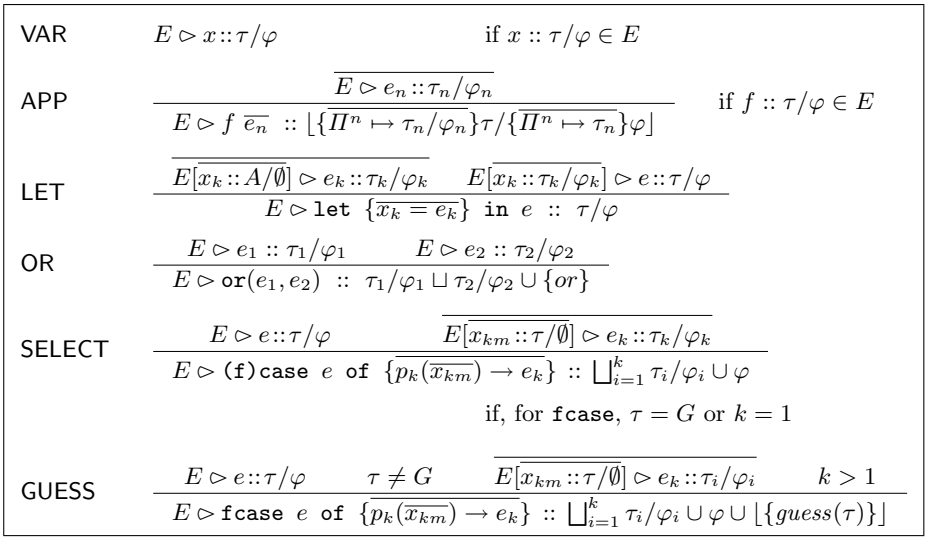

Fig. 4. Inference rules

If $\beta = \Pi^{i_1} \sqcup \ldots \sqcup \Pi^{i_j}/\varphi$, then the annotation is $\overline{\tau_n} \xrightarrow{\text{eff}(\tau_n,\varphi)} \tau$, where $\tau = A$, if $A \in \{\tau_{i_1}, \ldots, \tau_{i_j}\}$, and $\tau = G$, otherwise. □

3.2 Inferring Base Annotation

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 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:*

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

As an example for the type inference, consider the following flat program (c_0 and c_1 are data constructors of arity 0 and 1, respectively):

$$P_1 = \begin{cases} f\ x = \texttt{fcase}\ x\ \texttt{of}\ \{c_0 \to c_0, c_1\ y \to c_0\} \\ g\ \ = \texttt{let}\ \{x = x\}\ \texttt{in}\ x \\ h\ x = f\ g \end{cases}$$

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

$$\begin{aligned}
&Inf_0(P_1) = \{c_0 :: G/\emptyset, c_1 :: \Pi^1/\emptyset, f :: G/\emptyset, g :: G/\emptyset, h :: G/\emptyset\}\\
&Inf_1(P_1) = \{c_0 :: G/\emptyset, c_1 :: \Pi^1/\emptyset, f :: G/guess(\Pi^1), g :: A/\emptyset, h :: G/\emptyset\}\\
&Inf_2(P_1) = \{c_0 :: G/\emptyset, c_1 :: \Pi^1/\emptyset, f :: G/guess(\Pi^1), g :: A/\emptyset, h :: G/guess\}\\
&Inf(P_1) \;\; = Inf_2(P_1)
\end{aligned}$$

Thus, the inference correctly shows that a call to h can result in nondeterministic branches even for ground arguments. As a further example, consider the following flat program:

$$P_2 = \begin{cases} f_1\ x \;\; = \texttt{fcase}\ x\ \texttt{of}\ \{c_0 \to g, c_1\ y \to f_2\ x\} \\ f_2\ x\ y = f_1\ y \\ g \qquad = \texttt{let}\ \{x = x\}\ \texttt{in}\ x \end{cases}$$

For this program the type environments are computed as follows:

$$\begin{aligned}
&Inf_0(P_2) = \{c_0 :: G/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: G/\emptyset, f_2 :: G/\emptyset, g :: G/\emptyset\}\\
&Inf_1(P_2) = \{c_0 :: G/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: G/guess(\Pi^1), f_2 :: G/\emptyset, g :: A/\emptyset\}\\
&Inf_2(P_2) = \{c_0 :: G/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: A/guess(\Pi^1), f_2 :: G/guess(\Pi^2), g :: A/\emptyset\}\\
&Inf_3(P_2) = \{c_0 :: G/\emptyset, c_1 :: \Pi^1/\emptyset, f_1 :: A/guess(\Pi^1), f_2 :: A/guess(\Pi^2), g :: A/\emptyset\}\\
&Inf(P_2) \;\; = Inf_3(P_2)
\end{aligned}$$

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

3.3 Correctness of the Type Inference

We have defined a method to compute base annotations for all functions in a given program. Now we show that this computation is *correct* and specify what exactly "correct" means in this context. One important part of correctness is the termination of the iteration. To show this, we define an ordering on base annotations. Since this ordering is finite and the inference in each iteration always increases the set of computed base annotations w.r.t. this order, termination is ensured.

Definition 13 (Ordering on base annotations $\sqsubseteq$). *The ordering $\sqsubseteq$ is used on types, effects, base annotations and type environments. It is defined as the least ordering on base annotations satisfying the following properties:*

- $G \sqsubseteq \tau$ *and* $\tau \sqsubseteq A$ *for all types* τ
- $\Pi^{i_1} \sqcup \ldots \sqcup \Pi^{i_m} \sqsubseteq \Pi^{j_1} \sqcup \ldots \sqcup \Pi^{j_n}$ *if* $\{\Pi^{i_1}, \ldots, \Pi^{i_m}\} \subseteq \{\Pi^{j_1}, \ldots, \Pi^{j_n}\}$
- $guess(\tau) \sqsubseteq guess(\tau')$ *if* $\tau \sqsubseteq \tau'$
- *For effects* φ, φ'*:* $\varphi \sqsubseteq \varphi'$ *if* $\forall x \in \varphi\ \exists x' \in \varphi' : x \sqsubseteq x'$
- *Ordering on type/effects:* $\tau/\varphi \sqsubseteq \tau'/\varphi'$ *if* $\tau \sqsubseteq \tau'$ *and* $\varphi \sqsubseteq \varphi'$
- *Ordering on environments:* $E \sqsubseteq E'$ *if* $\forall x \in E\ \exists x' \in E' : x \sqsubseteq x'$

The next lemma shows the monotonicity of the inference, i.e., the inference always computes greater types for greater environments (with respect to $\sqsubseteq$).

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

Proof: The existence of the derivation $E' \rhd e :: \tau'/\varphi'$ is a direct consequence of E' containing a type for each expression in E. The prove of $\tau/\varphi \sqsubseteq \tau'/\varphi'$ is by induction over the length l of the inference for e:
Base case $l = 1$, $e = x$: The claim holds by definition of $E \sqsubseteq E'$.
Inductive case, the claim holds for all shorter derivations: Regard the last applied rule.
(APP) If the claim holds for τ/φ, τ'/φ', $\overline{x_n :: \tau_n/\varphi_n}$ and $\overline{x'_n :: \tau'_n/\varphi'_n}$ then it also holds for the resulting annotations

$$\begin{aligned} &\lfloor\{\overline{\Pi^n \mapsto \tau_n/\varphi_n}\}\tau/\{\overline{\Pi^n \mapsto \tau_n}\}\varphi\rfloor \\ \text{and } &\lfloor\{\overline{\Pi^n \mapsto \tau'_n/\varphi'_n}\}\tau'/\{\overline{\Pi^n \mapsto \tau_n}\}\varphi'\rfloor \end{aligned}$$

according to definition of $\sqsubseteq$: If $\tau = G$ or $\tau = A$ the claim trivially holds, and if τ is a composition of some Π^js then τ'/φ' is either A or containing the same (and maybe more) Π^js.
(LET) This rule directly derives the τ/φ type from the antecedent and thus the claim directly stems from the induction hypothesis.
(OR) The claim follows directly from the inductive assumption since the derived type is the supremum of the antecedent's types.
(Select) If the SELECT rule was applied in both derivations for E and E' then the claim holds since the resulting type is the supremum of antecedents' types unified with one of the antecedents' effect.
(GUESS) Analog to the treatment of the SELECT rule, the claim holds if both derivations apply GUESS, as the resulting type is a supremum of antecedent types.
(SELECT vs. GUESS) As $E \sqsubseteq E'$ it can happen that in the derivation for E the SELECT rule is applied, where the according derivation for E' applies GUESS. But as the resulting type for GUESS is identical to that of SELECT with the only exception that GUESS adds a $\{guess(\tau)\}$ effect, the resulting type for E' is greater than the one derived in E.
As the other rules are defined by the structure of e there are no further overlaps between them and the proof is finished accordingly. □

The first application of Lemma 5 is the observation that, in the fix-point iteration for Inf, the computed environments associate increasing types for defined functions:

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

Proof: By induction on the iterations i, where we define $Inf_i(f) = \tau/\varphi$ if $f :: \tau/\varphi \in Inf_i(P)$.

Base case $i = 1$. Since $Inf_0(f) = G/\emptyset$ the claim holds trivially.
Inductive case $Inf_j(f) \sqsubseteq Inf_i(f)$ for all $j \leq i$. Obviously the only rule that might change the type of f from Inf_i is APP, because it is the only rule accessing the function types in the environment. Thus, consider a function application $g\ \overline{x_n}$ and the types $Inf_i(g) = \tau_i/\varphi_i$ and $Inf_{i-1}(g) = \tau_{i-1}/\varphi_{i-1}$. As the mappings $\{\overline{\Pi^n \mapsto \tau_n/\varphi_n}\}$ and $\{\overline{\Pi^n \mapsto \tau_n}\}$ stay the same for each Inf_i and obviously preserve the ordering $\sqsubseteq$, the inferred type $\{\overline{\Pi^n \mapsto \tau_n/\varphi_n}\}\tau_i/\{\overline{\Pi^n \mapsto \tau_n}\}\varphi_i$ is greater than the type derived before $\{\overline{\Pi^n \mapsto \tau_n/\varphi_n}\}\tau_{i-1}/\{\overline{\Pi^n \mapsto \tau_n}\}\varphi_{i-1}$.
As $\rhd$ as a whole will compute a greater type for a greater environment (Lemma 5), this observation about APP is enough to prove the claim. □

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

Proof: Since there are only a finite number of base annotations for a given finite program P, the definition of $\sqsubseteq$ does not allow infinite chains. Thus, the claim follows directly from Lemma 6. □

Corollary 3 states that the inference finally terminates. To complete the soundness of the inference, we have to show that its computed results correctly and completely correspond to the results of the type/effect analysis of Section 2.

Lemma 7 (Correspondence between $\rhd$ and $\vdash$). *Let $f\ \overline{x_n} = e \in P$ and $\overline{\tau_n} \in \{A, G\}$ and for any i let $E_i = Inf_i(P)[\overline{x_n :: \tau_n/\emptyset}]$.*
Then, for any i, the inference $E_i \rhd e :: \beta$ directly corresponds to a derivation $types(E_i) \vdash e :: \beta(\overline{\tau_n})$ (as defined in Proposition 1).

Proof: By induction over the sum s of the lengths of the inferences for $Inf_0(P) \ldots Inf_i(P)$:
Base case, $s = 1, e = x_i$ or $e = g()$ or $e = c()$: For all three forms of e, the environments E_0 and $types(E_0)$ hold by definition the same base annotation and type which is one of $G/\emptyset$ or $A/\emptyset$. Since $G/\emptyset() = G/\emptyset$ and $A/\emptyset() = A/\emptyset$, the application of rule VAR is identical for $\rhd$ and $\vdash$.
Inductive case: The claim holds for all shorter inferences. Consider the rule last applied in $E_i \rhd e :: \beta$:
VAR: Analogously to the base case $e = x_i$ and E_i and $types(E_i)$ hold by definition the same type $G/\emptyset$ or $A/\emptyset$ and the application of rule VAR is identical for $\vdash$.
OR: By induction hypothesis, there are corresponding derivations for $types(E_i) \vdash e_{1/2} :: \tau_{1/2}/\varphi_{1/2}$. Therefore, $\tau_{1/2} \in \{A, G\}$ and $\varphi_{1/2} \subseteq \{or, guess\}$ and in consequence $max(\tau_1, \tau_2)/\varphi_1 \cup \varphi_2 \cup \{or\} = \tau_1/\varphi_1 \sqcup \tau_2/\varphi_2 \cup \{or\}$. The application of the rule OR is hence identical for $\vdash$.
SELECT: By induction hypothesis, all $\tau, \overline{\tau_k}$ are in $\{A, G\}$ and all $\varphi, \overline{\varphi_k}$ are a subset of $\{or, guess\}$. Therefore, $max(\overline{\tau_k})/\bigcup_{i=1}^{k} \varphi_i \cup \varphi = \bigsqcup_{i=1}^{k} \tau_i/\varphi_i \cup \varphi$ and the application of SELECT is identical for $\vdash$.
GUESS: By induction hypothesis, all $\tau, \overline{\tau_k}$ are in $\{A, G\}$ and all $\varphi, \overline{\varphi_k}$ are a subset of $\{or, guess\}$. Therefore, τ must be A, or rule GUESS would not have been

applicable. As $\lfloor\{guess(A)\}\rfloor = guess$, also $max(\overline{\tau_k})/\bigcup_{i=1}^{k}\varphi_i \cup \varphi \cup \{guess\} = \bigsqcup_{i=1}^{k}\tau_i/\varphi_i \cup \varphi \cup \lfloor\{guess(\tau)\}\rfloor$ and the application of GUESS is identical for $\vdash$.
LET: The claim directly stems form the induction hypothesis because of the identical structure of the rules for $\rhd$ and $\vdash$. Note that both rules only assign type $A/\emptyset$ for the new variables.
APP: By induction hypothesis all $\overline{\tau_n}$ are in $\{A, G\}$ and all $\overline{\varphi_n}$ are a subset of $\{or, guess\}$. Because of this, $\lfloor\{\overline{\Pi^n \mapsto \tau_n/\varphi_n}\}\tau/\{\overline{\Pi^n \mapsto \tau_n}\}\varphi\rfloor = \tau/\varphi(\tau_n)$, by Definition 11 and Proposition 1. As $\tau/\varphi(\tau_n)$ is by definition also in $types(\tau/\varphi)$, the application of APP in $\rhd$ directly corresponds to the one in $\vdash$. □

Theorem 2 (Correctness of the inference). *Let P be a flat program, $E(P) = types(Inf(P))$ and E be a correct environment (cf. Definition 6) 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).*

Proof:

Soundness:
A correct environment is by Definition 6 (a) constructor correct and (b) contains only correct annotations for defined functions.
(a) By Definition 12, the $Inf_i(P)$ of the iteration to compute $E(P)$ differ in the types for defined functions only. Where constructors are concerned, $Inf(P)$ equals $Inf_0(P)$ and, thus,
$(\{c :: G/\emptyset \mid c^0 \in P\} \cup \{c :: \tau \mid c^{n>0} \in P, \tau \in types(\Pi^1 \sqcup \ldots \sqcup \Pi^n/\emptyset)\}) \subset E(P)$
where $c^n \in P$ means that c is an n-ary constructor called somewhere in P. By Definition 11 of $types()$, this subset is exactly the one postulated by Definition 5 of constructor correctness. Hence, $E(P)$ is constructor correct.
(b) $E(P)$ contains only correct annotations because of the correspondence between $\rhd$ and $\vdash$ shown by Lemma 7. Let $f\ \overline{x_n} = e \in P$ and $f :: \beta \in Inf(P)$. By Lemma 7, for all $\overline{\tau_n} \in \{A, G\}$, $Inf(P)[\overline{x_n :: \tau_n/\emptyset}] \rhd e :: \beta'$ directly corresponds to a derivation $E(P)[x_n :: \tau_n/\emptyset] \vdash e :: \beta'(\tau_n)$. And as $Inf(P)$ is a fix-point regarding $\rhd$, the correspondence yields that $E(P)$ is a fix-point regarding $\vdash$. Hence, by Definition 6, $f :: \beta'(\tau_n) \in E(P)$, and in consequence $f :: \beta'(\tau_n) \in E(P)$ is correct for $f\ \overline{x_n} = e \in P$.

Completeness:
By definition of $\leq$, for any annotation $\mathcal{A} = f :: \overline{\tau_n} \xrightarrow{\varphi} \tau$ and any base annotation β for f, the annotation $\beta(\tau_n)$ is comparable to $\mathcal{A}$ and we write for short $\beta \leq \mathcal{A}$ or $\mathcal{A} \leq \beta$, respectively. To prove a contradiction, we assume $f :: \beta \in Inf(P)$ and $\mathcal{A} = f :: \overline{\tau_n} \xrightarrow{\varphi} \tau$ an annotation for $f\ \overline{x_n} = e \in P$ with $\mathcal{A} < \beta$. Furthermore, let $\overline{\beta_i}$ be the type annotations of f in all the sets $\overline{Inf_i(P)}$ computed during the fix-point iteration. As the iteration starts with $\beta_0 = G/\emptyset$, there must exist greatest j, β_j with $\beta_j \leq \mathcal{A}$. By assumption, β_j must be different from β. Also, for $Inf_j(P) \rhd e :: \beta_k$, $\beta_j < \beta_k$ and $\beta_j(\overline{\tau_n}) < \beta_k(\overline{\tau_n})$. By Lemma 7, there exists a derivation $types(Inf_j(P))[\overline{x_n :: \tau_n/\emptyset}] \vdash e :: \beta_k(\overline{\tau_n})$ and $\beta_j(\overline{\tau_n}) \leq \mathcal{A} \leq \beta_k(\overline{\tau_n})$. It is easy to verify that this means that either $\mathcal{A} = \beta_j(\overline{\tau_n})$ or $\mathcal{A} = \beta_k(\overline{\tau_n})$,

which contradicts the assumption: If $\mathcal{A} = \beta_j(\overline{\tau_n})$ then $\mathcal{A}$ is not correct and if $\mathcal{A} = \beta_k(\overline{\tau_n})$ then j was not the greatest against its definition. □

4 Implementation

We have implemented the type inference described in this paper for the functional logic language Curry [9, 16]. The implementation is done in Curry and exploits the meta-programming features and efficient data structures (e.g., tables, sets) of the PAKCS environment of Curry [12]. In the following we show some details of this implementation.

The PAKCS environment provides a front end which compiles Curry programs into a structure almost identical to the flat programs presented in this paper. There is also a library defining a Curry datatype corresponding to this structure. This library is the base for meta programming in and for Curry. A Curry module is mainly defined by a name, a list of datatype declarations and a list of function declarations. Like in flat programs (cf. Figure 1), each function is defined by a single rule, which consists of a list of variables and an expression. We give the definition of expressions in some detail:[2]

```
data Expr = Var VarIndex
          | Lit (Char|Int|Float)
          | Free [VarIndex] Expr
          | Or Expr Expr
          | Comb (FuncCall|ConsCall) QName [Expr]
          | Case (Flex|Rigid) VarIndex [Branch Pattern Expr]
          | Let (VarIndex,Expr) Expr
data Pattern = LPattern (Char|Int|Float)
             | Pattern QName [VarIndex]
```

There are a few small differences to flat programs: (a) the inclusion of literals (`Lit, LPattern`) of the base types integer (`Int`), character (`Char`) or floating-point number (`Float`), and (b) the `Free` construct to introduce logic variables instead of a directly circular let binding. In all other aspects the correspondence is direct: Constructor calls are denoted as (`Comb ConsCall`) and function calls as (`Comb FuncCall`), both followed by a qualified name (`QName`) which is the name of the function together wit that of the module it is defined in. *fcase* corresponds to (`Case Flex`), *case* to (`Case Rigid`), and variables are numbered (`VarIndex` is a synonym for `Int`).

The types and effects are defined in our implementation as:

```
data TypeEffectVar = V VarIndex
data Type = Ground | Any | VarSet (Set VarIndex)
data Effect = Effect Bool Bool (Set VarIndex)
```

[2] We take the liberty defining variants like `(Flex|Rigid)` directly in the same type declaration, although in Curry, we need a separate definition for them. We have also blended out the support of higher-order features in Curry, since the corresponding part of the nondeterminism analysis is not covered in this paper.

```
data TypeEffect = TE Type Effect | TEFunc VarIndex TypeEffect
```

The first boolean value in the definition of `Effect` correspond to the or effect, whereas the second boolean and the `Set VarIndex` model the guess effect. In order to avoid clashes among variable indices, the constructor `TEFunc` defines which variable index corresponds to which argument of a given function. For instance, a binary constructor `C` would have the type/effect

```
(TEFunc 0 (TEFunc 1 (TE VarSet {0,1} (Effect False emptySet))))
```

written as `C::a->b->a+b/{}` for ease of reading. Some convenient abbreviations are:

```
noEffect = Effect False emptySet
teGround = TE Ground noEffect
teAny = TE Any noEffect
teVar i = TE (VarSet (addToSet i emptySet)) noEffect
```

There is also a simple manipulation function

```
addEffect :: Effect -> TypeEffect -> TypeEffect
```

which is used in the abbreviations:

```
addOr         = addEffect (Effect True  False emptySet)
addGuess      = addEffect (Effect False True  emptySet)
addGuesses vs = addEffect (Effect False False vs)
```

The only thing we need to know about the implementation of `Set` is that there are functions:

```
emptySet::Set a                    : generate an empty set
elemSet :: a -> Set a -> Bool      : test if given value is element of set
union :: Set a -> Set a -> Set a   : union of two sets
unions :: [Set a] -> Set a         : union of a list of sets
delFromSet :: Set a -> a -> Set a : delete element from given set
addToSet :: a -> Set a -> Set a    : add element to given set
isEmpty :: Set a -> Bool           : test if set contains no elements
```

The supremum on type/effects can now be defined in Curry as (`||` denotes the boolean disjunction):[3]

```
supremums = foldl supremum teGround
supremum (TE s v) (TE t w) = TE (supTypes s t) (supEffects v w)
supTypes Any          _            = Any
supTypes Ground       x            = x
supTypes (VarSet _)   Any          = Any
supTypes (VarSet vs)  Ground       = VarSet vs
supTypes (VarSet vs) (VarSet vs') = VarSet (union vs vs')
supEffects (Effect o1 g1 s1) (Effect o2 g2 s2) =
            Effect (o1||o2) (g1||g2) (union s1 s2)
```

[3] Note that we define the supremum of `TE`-rooted terms only, the supremum of functional terms is neither defined nor needed.

The environment is modeled by two tables, one to look up the type/effects of already analyzed functions, and the other contains the types associated to variables. We access these tables by the functions

```
lookup         :: i -> Table i a -> a
addListToTable :: Table i a -> [(i,a)] -> Table i a
```

Now, the analysis of expressions can be defined as a case distinction on the different patterns:

```
ana funcTable varTable (Var i) = lookup varTable i
ana _ _ (Lit _) = teGround
ana fT vT (Or e e') = addOr (supremums (map (ana fT vT) [e,e']))
ana fT vT (Comb _ name args) =
   applys (lookup fT name) (map (ana fT vT) args)
ana fT vT (Free vs e) = ana fT newVT e
  where newVT = addListToTable vT (zip vs (repeat teAny))
ana fT vT (Let (v,e) e') = ana fT newVs e'
  where newVs = addToTable vT (v,ana fT vT e)
ana fT vT (Case caseType i branches) = (case lookup vT i of
    TE Any eff -> addEffect eff . if guess then addGuess else id
    TE Ground eff -> addEffect eff
    TE (VarSet vs) eff -> addEffect eff .
      if elemSet i vs && guess then addGuesses [i] else id
 ) (supremums (map (anaBranch i fT (addToTable vT (i,teGround)))
                   branches))
  where guess = caseType==Flex && length branches > 1
```

Thanks to the high programming level of Curry, the correspondence of the code to the definition of the analysis is relatively strong. To conclude the presentation of the source code, we give the definitions of two functions called from `ana`, namely `apply` and `anaBranch`:

```
applys t ts = foldl apply t ts
apply (TEFunc i t) t' = subst i t' t
subst i t (TEFunc j t') | i==j      = TEFunc j t'
                        | otherwise = TEFunc j (subst i t t')
subst i (TE t eff) (TE t' eff') = TE (subT i t t') (subE i eff eff')
subT _ _ Ground = Ground
subT _ _ Any    = Any
subT i t (VarSet vs) =  case t of
    Ground -> let newVs = delFromSet vs i
              in if isEmpty newVs then Ground else VarSet newVs
    Any -> if elemSet i vs then Any else VarSet vs
    VarSet vs' -> VarSet (union vs vs')
subE i (Effect or g vs) (Effect or' g' vs')
  | elemSet i vs' =  Effect (or||or') (g||g') (union vs vs')
  | otherwise     =  Effect or' g' vs'
anaBranch _ fT vT (Branch (LPattern _) e) = ana fT vT e
```

```
anaBranch i fT vT (Branch (Pattern _ vs) e) =
  ana fT (addListToTable vT (zip vs (repeat (teVar i)))) e
```

In order to show a few concrete results, we give a simple example program (left column) and the output generated by our implementation of the analysis (right column):

```
data N = Z | S N

four = S(S(S(S Z)))              four :: Ground/{}

x = let x free in x              x :: Any/{}

inc Z     = S Z
inc (S n) = S (inc n)            inc :: a -> Ground/NarrowIf(a)

inc' n = S n                     inc' :: a -> a/{}

plus Z     n = n
plus (S n) m = S (plus n m)      plus :: a -> b -> b/NarrowIf(a)

eight = plus four four           eight :: Ground/{}
```

5 Conclusions

We have presented a program analysis to approximate to 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 [18]), 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, 16] 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. *Journal of Symbolic Computation*, 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 the 6th International Symposium on Functional and Logic Programming (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*, Vol. 2004, 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, M.J. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. 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. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, 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, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2004.
13. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, Vol. 9, No. 1, pp. 33–75, 1999.
14. 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.
15. M. Hanus and F. Steiner. Type-based Nondeterminism Checking in Functional Logic Programs. In *Proc. of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pp. 202–213. ACM Press, 2000.
16. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
17. F. Henderson, T. Somogyi, and Z. Conway. Determinism analysis in the Mercury compiler. In *Proc. of the Nineteenth Australian Computer Science Conference*, pp. 337–346, 1996.
18. J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pp. 17–42. Addison Wesley, 1990.
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, and 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.