Call Pattern Analysis for Functional Logic Programs *

Michael Hanus
Institut für Informatik, CAU Kiel, Germany
mh@informatik.uni-kiel.de

Abstract

This paper presents a new program analysis framework to approximate call patterns and their results in functional logic computations. We consider programs containing non-strict, nondeterministic operations in order to make the analysis applicable to modern functional logic languages like Curry or TOY. For this purpose, we present a new fixpoint characterization of functional logic computations w.r.t. a set of initial calls. We show how programs can be analyzed by approximating this fixpoint. The results of such an approximation have various applications, e.g., program optimization as well as verifying safety properties of programs.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.4 [*Programming Techniques*]: Processors—Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Languages

Keywords Functional Logic Programming, Program Analysis

1. Introduction

Functional logic languages integrate the most important features of functional and logic languages to provide a variety of programming concepts. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming are combined with logic programming features like computing with partial information (logic variables), constraint solving, and nondeterministic search for solutions. This combination supports optimal evaluation strategies (Antoy 1997; Antoy et al. 2000) and leads to better abstractions in application programs (see (Hanus 2007) for a recent survey).

In this paper we propose a new method to analyze call patterns of functional logic programs, i.e., we want to approximate the arguments of all function calls occurring in a computation w.r.t. a program and a set of initial calls. Such approximations are useful in various ways. For instance, they can be used to optimize programs (e.g., partial evaluation, eliminating unnecessary code (Peyton Jones 2007)), to catch pattern-match errors due to partial function definitions at compile time (Mitchell and Runciman 2007), or to verify safety conditions of programs (e.g., which files or sockets are accessed during a computation (Albert et al. 2005)). The contributions of this work are:

1. We present a new fixpoint characterization of functional logic computations w.r.t. a set of initial calls (Section 3). We consider programs containing non-strict, nondeterministic operations with call-time choice semantics as in modern functional logic languages like Curry (Hanus 1997, 2006) or TOY (López-Fraguas and Sánchez-Hernández 1999).
2. We show the soundness of this fixpoint characterization w.r.t. the rewriting logic CRWL (González-Moreno et al. 1999), a standard semantics for such kind of languages (Hanus 2007).
3. We introduce a general framework to analyze call patterns based on this fixpoint characterization (Section 4). An example analysis with depth-bounded terms is presented in Section 5.
4. We discuss how this framework can be extended to features occurring in application programs, like higher-order functions and primitive operations to perform I/O, and present a practical evaluation of a prototypical implementation (Section 6).

Due to lack of space, the proofs of the results presented in this paper are omitted. They can be found in the full version of the paper (Hanus 2008).

2. Basic Concepts

In this section we review some concepts and notations from term rewriting (Baader and Nipkow 1998; Dershowitz and Jouannaud 1990) and functional logic programming (Hanus 1994, 2007) that are used in this paper.

* This work was partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

PPDP'08, July 15–17, 2008, Valencia, Spain.

Although modern functional logic languages are strongly typed, we ignore this aspect for the sake of simplicity. However, the distinction between defined functions and data constructors is important for the definition of the operational semantics of such languages (Hanus 2007). Therefore, we consider a *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and operation symbols, respectively. Given a set of variables $\mathcal{X}$, the set of *terms* and *constructor terms* are denoted by $\mathcal{T}(\Sigma, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. We write $\mathcal{V}ar(t)$ for the set of all the variables occurring in a term t. A term is *linear* if it does not contain multiple occurrences of a variable. A term is *operation-rooted* (*constructor-rooted*) if its root symbol is an operation (constructor).

A *pattern* is a linear term of the form $f(t_1, \ldots, t_n)$ where $f/n \in \mathcal{F}$ is an operation symbol and $t_1, \ldots, t_n$ are constructor terms. A *functional logic program* is a constructor-based rewrite system, i.e., a set of pairs of terms or *rewrite rules* of the form $l \to r$ where l is a pattern. Traditionally, term rewriting systems have the additional requirement $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. However, in functional logic programming variables occurring in $\mathcal{V}ar(r)$ but not in $\mathcal{V}ar(l)$, called *extra variables*, are often useful. Therefore, we allow rewrite rules with extra variables in functional logic programs.

Example 2.1. *The following functional logic program is based on the set of constructors $\mathcal{C} = \{0/0, S/1\}$ to represent natural numbers. It defines operations to add two natural numbers and to double the value of a number, and an operation coin that returns one of the two values* 0 *or* $S(0)$.

$$\begin{aligned} add(0, x) &\to x \\ add(S(x), y) &\to S(add(x, y)) \\ double(x) &\to add(x, x) \\ coin &\to 0 \\ coin &\to S(0) \end{aligned}$$

Note that an operation like coin is not admissible in purely functional programs since it has more than one normal form. However, in the context of functional logic programming such nondeterministic operations *are permitted and useful for programming (González-Moreno et al. 1999; Hanus 2007). Actually, it has been shown that logic variables and nondeterministic functions have the same expressive power (Antoy and Hanus 2006) although we consider both concepts for the sake of simplicity.*

To formally define computations w.r.t. a given program, additional notions are necessary. A *position* p in a term t is represented by a sequence of natural numbers. Positions are used to identify specific subterms. Thus, $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ with the term s (see (Dershowitz and Jouannaud 1990) for details). The set of all positions of a term t is denoted by $\mathcal{P}os(t)$, and the set of all positions of operation-rooted subterms of a term t is denoted by $\mathcal{FP}os(t)$. A *substitution* is an idempotent mapping $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$ such that its *domain* $\mathcal{D}om(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. We denote a substitution σ by the finite set $\{x \mapsto \sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. In particular, $\varnothing$ denotes the identity substitution. Substitutions are extended to morphisms on terms in the obvious way.

In classical term rewriting, a *rewrite step* $t \to_{p,l\to r,\sigma} t'$ w.r.t. a given rewrite system $\mathcal{R}$ is defined if there are a position p in t, a rule $l \to r \in \mathcal{R}$, and a substitution σ with $t|_p = \sigma(l)$ such that $t' = t[\sigma(r)]_p$ (where we omit the subscripts if they are not relevant). However, this classical notion of term rewriting is not suitable for functional logic programs that contain nondeterministic, non-strict functions defined by non-confluent programs. For instance, classical rewriting allows the following sequence of rewrite steps w.r.t. Example 2.1:

$$\begin{aligned} double(coin) &\to add(coin, coin) \\ &\to add(0, coin) \\ &\to add(0, S(0)) \\ &\to S(0) \end{aligned}$$

This result is not intended since the operation *double* should return only even numbers. In order to cover this intended behavior, González-Moreno et al. (1999) have proposed the rewriting logic *CRWL* (Constructor-based conditional ReWriting Logic) as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and nondeterministic operations. This rewriting logic specifies the *call-time choice* semantics where the values of the arguments of an operation are determined before the operation is evaluated. To deal with non-strict operations, CRWL considers signatures $\Sigma_\bot$ that are extended by a special symbol $\bot$ to represent *undefined values*. We define $\mathcal{C}_\bot = \mathcal{C} \cup \{\bot\}$ so that $\mathcal{T}(\mathcal{C}_\bot, \mathcal{X})$ denotes the set of partial constructor terms, e.g., $S(S(\bot))$ denotes a number greater than 1 where the exact value is undefined. Such *partial terms* are considered as finite approximations of possibly infinite values. CRWL defines the deduction of *approximation statements*[1] $e \twoheadrightarrow t$ with the intended meaning "the partial constructor term t approximates the value of e". To model call-time choice semantics, rewrite rules are only applied to partial *values*. Hence, the following notation for *partial constructor instances* of a set of rules $\mathcal{R}$ is useful:

$$[\mathcal{R}]_\bot = \{\sigma(l) \to \sigma(r) \mid l \to r \in \mathcal{R}, \sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C}_\bot, \mathcal{X})\}$$

Then CRWL is defined by the following set of inference rules (where the program is represented by a TRS $\mathcal{R}$):

$$\frac{}{e \twoheadrightarrow \bot} \quad \text{for any } e \in \mathcal{T}(\Sigma_\bot, \mathcal{X})$$

$$\frac{}{x \twoheadrightarrow x} \quad \text{for any variable } x \in \mathcal{X}$$

$$\frac{e_1 \twoheadrightarrow t_1 \;\cdots\; e_n \twoheadrightarrow t_n}{c(e_1, \ldots, e_n) \twoheadrightarrow c(t_1, \ldots, t_n)} \quad \text{for any } c/n \in \mathcal{C},\ t_i \in \mathcal{T}(\mathcal{C}_\bot, \mathcal{X})$$

$$\frac{e_1 \twoheadrightarrow t_1 \;\cdots\; e_n \twoheadrightarrow t_n \quad r \twoheadrightarrow t}{f(e_1, \ldots, e_n) \twoheadrightarrow t} \quad \text{if } f(t_1, \ldots, t_n) \to r \in [\mathcal{R}]_\bot \text{ and } t \neq \bot$$

The first rule specifies that $\bot$ approximates any expression. The condition $t \neq \bot$ in the last rule avoids unnecessary applications of this rule since this case is already covered by the first rule. The restriction to partial constructor instances in this rule formalizes non-strict functions with a call-time choice semantics. Functions might have non-strict arguments that are not evaluated since the corresponding actual arguments can be derived to $\bot$ by the first rule. If the

[1] For the sake of simplicity, we consider only unconditional rules in contrast to the original presentation of CRWL.

value of an argument is required to evaluate the right-hand side of a function's rule, it must be evaluated to a partial constructor term before it is passed to the right-hand side (since $[\mathcal{R}]_\bot$ contains only partial constructor instances), which corresponds to a call-time choice semantics. Note that this does not prohibit the use of lazy implementations since this semantical behavior can be enforced by sharing unevaluated expressions. Actually, González-Moreno et al. (1999) define a lazy narrowing calculus that reflects this behavior.[2]

In order to apply our program analysis to functional logic programs as discussed above, we intend to use CRWL as its foundation. However, it has been noted (López-Fraguas et al. 2007) that this calculus has some drawbacks compared to classical definitions of rewriting where computation steps can be directly applied to any subterm rather than decomposing terms until the function call appears at the top level. Therefore, we use in the following an alternative definition of rewrite steps conform with CRWL computations. This rewrite relation, $s \rightarrowtail t$, is defined by the following rules:

$$\begin{array}{rcll} e & \rightarrowtail & e[\bot]_p & \text{if } p \in \mathcal{FP}os(e) \\ e[f(t_1,\ldots,t_n)]_p & \rightarrowtail & e[r]_p & \text{if } f(t_1,\ldots,t_n) \to r \in [\mathcal{R}]_\bot \end{array}$$

The first rule allows the approximation of any (in particular, non-demanded) function call by $\bot$. The second rule corresponds to classical rewrite steps except that arguments must be already evaluated to partial values, which corresponds to a call-time choice semantics. As usual, we denote by $\rightarrowtail^*$ the reflexive-transitive closure of the relation $\rightarrowtail$.

Note that a similar rewrite relation has been proposed by López-Fraguas et al. (2007) but with the difference that also constructor terms can be approximated by $\bot$. Hence, the rewrite relation of (López-Fraguas et al. 2007), which we denote by $\rightarrowtail_c$, is defined as follows:

$$\begin{array}{rcll} e & \rightarrowtail_c & e[\bot]_p & \text{if } p \in \mathcal{P}os(e) \\ e[f(t_1,\ldots,t_n)]_p & \rightarrowtail_c & e[r]_p & \text{if } f(t_1,\ldots,t_n) \to r \in [\mathcal{R}]_\bot \end{array}$$

Note that only the first rule differs from our relation $\rightarrowtail$ by allowing also the replacement of constructor-rooted terms and variables by $\bot$. The equivalence of the rewrite relation $\rightarrowtail_c$ and CRWL is shown in (López-Fraguas et al. 2007). However, our rewrite relation $\rightarrowtail$ restricts the nondeterministic choices due to the approximation of values since it does not allow the approximation of constructor terms.[3] In order to justify the use of our rewrite relation, we establish a precise connection between the values computable by both rewrite relations. Basically, we show that the restriction of our relation $\rightarrowtail$ does does not change the applicability of rewrite rules but computes better approximations of constructor terms. For this purpose, we define the usual approximation ordering on partial expressions.

DEFINITION 2.2. The set of partial expressions $\mathcal{T}(\Sigma_\bot, \mathcal{X})$ is ordered by the *approximation ordering* $\sqsubseteq$ which is defined as the least partial ordering satisfying $\bot \sqsubseteq t$ and $f(s_1,\ldots,s_n) \sqsubseteq f(t_1,\ldots,t_n)$ if $s_1 \sqsubseteq t_1, \ldots, s_n \sqsubseteq t_n$, for all $s_i, t_i, t \in \mathcal{T}(\Sigma_\bot, \mathcal{X})$ and $f/n \in \Sigma$. □

Now we can establish the relation between CRWL and $\rightarrowtail$ by stating that our relation $\rightarrowtail$ computes the same or better approximations of constructor terms than $\rightarrowtail_c$ (which is equivalent to $\twoheadrightarrow$, see (López-Fraguas et al. 2007, Theorem 1)).

THEOREM 2.3. *Let $e \in \mathcal{T}(\Sigma_\bot, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{C}_\bot, \mathcal{X})$.*

1. *If $e \rightarrowtail^* t$, then $e \rightarrowtail_c^* t$.*
2. *If $e \rightarrowtail_c^* t$, then $e \rightarrowtail^* t'$ for some $t' \in \mathcal{T}(\mathcal{C}_\bot, \mathcal{X})$ with $t \sqsubseteq t'$.*

Although our rewrite relation $\rightarrowtail$ has less nondeterministic choices than $\rightarrowtail_c$ or CRWL, there is still an apparent nondeterminism due to the choice between function evaluation or approximation of functions by $\bot$ in the rules defining $\rightarrowtail$. This might cause an approximation of many partial values in a program analysis framework based on this semantics. However, this potential disadvantage can be avoided in the analysis by computing only maximal elements w.r.t. an information ordering on abstract values, as we will see below.

For the purpose of our analysis of functional logic programs, it is sufficient to use the rewrite relation $\rightarrowtail$ as a basis. Although concrete functional logic languages use sophisticated narrowing strategies to evaluate programs (Hanus 2007), narrowing derivations have a strong correspondence to rewriting derivations, i.e., each narrowing derivation usually corresponds to a rewriting derivation after applying the substitutions computed by narrowing. Thus, if we approximate the call patterns occurring in all rewriting derivations (as done in the following), we obtain also approximations of all possible call patterns occurring in narrowing derivations.

3. Fixpoint Semantics for Functional Logic Computations

In this section we develop a fixpoint characterization of functional logic computations that will be later used to approximate the call patterns occurring in concrete computations. From now on, we assume a fixed signature $\Sigma = \mathcal{F} \cup \mathcal{C}$ and a set of variables $\mathcal{X}$.

In contrast to related abstractions of term rewriting systems (e.g., (Alpuente et al. 2002; Bert and Echahed 1995; Bert et al. 1993)), we intend to approximate only those call patterns that might occur in concrete computations for specific applications, i.e., starting from some set of initial function calls. Therefore, we assume a given set

$$\mathcal{M} \subseteq \{f(t_1,\ldots,t_n) \mid f/n \in \mathcal{F}, t_1,\ldots,t_n \in \mathcal{T}(\mathcal{C},\mathcal{X})\}$$

of *initial* or *main calls*, i.e., functions applied to constructor terms from which any concrete computation starts. Having only constructor terms as arguments is not a restriction, since nested function calls can be easily removed by introducing new auxiliary functions.

Our fixpoint semantics is based on interpretations that consist of pairs or equations of expressions describing the computed input/output relation of all functions. Thus, the *base domain* is the set of equations

$$\mathcal{E} = \{f(t_1,\ldots,t_n) \doteq t \mid f/n \in \mathcal{F}, t_1,\ldots,t_n, t \in \mathcal{T}(\mathcal{C}_\bot,\mathcal{X})\}$$

[2] There are also lazy narrowing calculi that model sharing by graph structures, e.g., (Echahed and Janodet 1998). Since this requires the handling of complex graph structures and their approximations, we base our development on the conceptually simpler rewriting logic CRWL.

[3] The advantage of such a restriction has been also recognized in (Cleva et al. 2004) in the context of CRWL.

Note that we allow partial constructor terms in order to deal with the partial construction of the input/output relation during the fixpoint computation. A particular *interpretation* is a subset of $\mathcal{E}$.

An important aspect of our semantics is the stepwise extension of the set of initial calls with more function calls occurring during the computation. For this purpose, we have to evaluate concrete terms occurring in a program w.r.t. a current interpretation as defined next.

DEFINITION 3.1 (Evaluation of terms). Let $I \subseteq \mathcal{E}$ be an interpretation. The *evaluation of a term t w.r.t. I* is a mapping $eval_I : \mathcal{T}(\Sigma, \mathcal{X}) \to 2^{\mathcal{T}(\mathcal{C}_\bot, \mathcal{X})}$ defined by

$$\begin{aligned} eval_I(x) &= \{x\} \\ eval_I(c(e_1, \ldots, e_n)) &= \{c(t_1, \ldots, t_n) \mid t_i \in eval_I(e_i), \\ &\qquad\qquad i = 1, \ldots, n\} \\ eval_I(f(e_1, \ldots, e_n)) &= \{\bot\} \cup \{t \mid t_i \in eval_I(e_i), i = 1, \ldots, n, \\ &\qquad\qquad f(t_1, \ldots, t_n) \doteq t \in I\} \end{aligned}$$

□

For instance, consider the interpretation

$$\begin{aligned} I = \{&coin \doteq 0,\ coin \doteq S(0), \\ &double(0) \doteq 0,\ double(S(0)) \doteq S(S(0))\} \end{aligned}$$

Then

$$eval_I(coin) = \{\bot, 0, S(0)\}$$

and

$$eval_I(double(coin)) = \{\bot, 0, S(S(0))\}\ .$$

Note that $eval_I(double(coin))$ does not contain $S(0)$ since this value does not occur in any right-hand side of a *double* equation in interpretation I.

Obviously, partial constructor terms are always evaluated to their own values, as formally stated in the following proposition.

PROPOSITION 3.2. *$eval_I(t) = \{t\}$ for all $t \in \mathcal{T}(\mathcal{C}_\bot, \mathcal{X})$ and interpretations $I \subseteq \mathcal{E}$.*

A specific term evaluation that is sometimes used is the interpretation where all function calls are replaced by $\bot$.

DEFINITION 3.3 (Bottom evaluation).
Let $e \in \mathcal{T}(\Sigma, \mathcal{X})$. Then the *bottom evaluation* $e^\bot$ of e is defined by $e^\bot = t$ for $\{t\} = eval_\varnothing(e)$. □

PROPOSITION 3.4. *$e^\bot \in eval_I(e)$ for all $e \in \mathcal{T}(\Sigma, \mathcal{X})$ and interpretations $I \subseteq \mathcal{E}$.*

Now we are able to define a transformation on interpretations (similarly to the immediate consequence operator in logic programming) that covers information about potential function calls and their computed results.

DEFINITION 3.5 (Transformation $T_{\mathcal{R},\mathcal{M}}$). Let $\mathcal{R}$ be a functional logic program and $\mathcal{M}$ a set of main calls. The set of *initial equations* $\mathcal{M}_\bot$ is defined by

$$\mathcal{M}_\bot = \{s \doteq \bot \mid s \in \mathcal{M}\}$$

The transformation $T_{\mathcal{R},\mathcal{M}}$ on interpretations $I \subseteq \mathcal{E}$ is defined as follows:

$$\begin{aligned} &T_{\mathcal{R},\mathcal{M}}(I) = \\ &\quad \mathcal{M}_\bot \cup \{s \doteq r' \mid s \doteq t \in I,\ s \to r \in [\mathcal{R}]_\bot,\ r' \in eval_I(r)\} \\ &\qquad \cup \{f(t_1, \ldots, t_n) \doteq \bot \mid s \doteq t \in I,\ s \to r \in [\mathcal{R}]_\bot, \\ &\qquad\qquad p \in \mathcal{FP}os(r) \text{ with } r|_p = f(e_1, \ldots, e_n), \\ &\qquad\qquad t_i \in eval_I(e_i),\ i = 1, \ldots, n\} \end{aligned}$$

As usual, we define

$$\begin{aligned} T_{\mathcal{R},\mathcal{M}} \uparrow 0 &= \varnothing \\ T_{\mathcal{R},\mathcal{M}} \uparrow k &= T_{\mathcal{R},\mathcal{M}}(T_{\mathcal{R},\mathcal{M}} \uparrow (k-1)) \quad (\text{for } k > 0) \end{aligned}$$

□

Informally speaking, the transformation $T_{\mathcal{R},\mathcal{M}}$ adds to the set of initial equations in each iteration

1. better approximations of the rules' right-hand sides ($s \doteq r'$) and
2. new function calls occurring in right-hand sides ($f(t_1, \ldots, t_n) \doteq \bot$).

As we will see later, the least fixpoint of the transformation $T_{\mathcal{R},\mathcal{M}}$ contains the information about all function calls and all results computed during derivations starting from the main calls. Thus, the construction described by $T_{\mathcal{R},\mathcal{M}}$ is similar to the notion of *minimal function graphs* introduced in (Jones and Mycroft 1986) to analyze (strict) functional programs and relating program analysis frameworks based on operational (Cousot and Cousot 1977) and denotational semantics. Minimal function graphs have been also used to optimize logic programs (e.g., (Gallagher and Bruynooghe 1991; Winsborough 1992)). Although all these frameworks are based on a common principle (incremental computation of all reachable calls and their results), the concrete computational methods are different. In our case, we have to model CRWL computations that might be non-strict, i.e., functions might reduce on arguments that have no value. For this purpose, our transformation $T_{\mathcal{R},\mathcal{M}}$ evaluates (by $eval_I$) arbitrary argument expressions to partial constructor terms before adding the new function calls so that it will be checked (in the next iteration step) whether these calls can be evaluated by applying a rewrite rule or stay unevaluated in the interpretation. The modelling of interpretations as sets of equations provides an intuitive understanding of the elements of an interpretation: If an interpretation computed from a set of initial calls contains an equation $f(t_1, \ldots, t_n) \doteq t$, then the call $f(t_1, \ldots, t_n)$ might occur in a computation of some initial call and t is an approximated result value of this call. Before stating this intuition more formally, we provide an example computation w.r.t. $T_{\mathcal{R},\mathcal{M}}$.

EXAMPLE 3.6. *Consider the program of Example 2.1 extended by the rule*

$$main \to double(coin)$$

If the set of main calls is $\mathcal{M} = \{main\}$, the transformation $T_{\mathcal{R},\mathcal{M}}$ computes the following sequence of interpretations,

where we write T_i for $T_{\mathcal{R},\mathcal{M}} \uparrow i$.

$$
\begin{array}{rcl}
T_0 & = & \varnothing \\
T_1 & = & \{main \doteq \bot\} \\
T_2 & = & T_1 \cup \{coin \doteq \bot, double(\bot) \doteq \bot\} \\
T_3 & = & T_2 \cup \{add(\bot,\bot) \doteq \bot, coin \doteq 0, coin \doteq S(0)\} \\
T_4 & = & T_3 \cup \{double(0) \doteq \bot, double(S(0)) \doteq \bot\} \\
T_5 & = & T_4 \cup \{add(0,0) \doteq \bot, add(S(0),S(0)) \doteq \bot\} \\
T_6 & = & T_5 \cup \{add(0,0) \doteq 0, add(0,S(0)) \doteq \bot, \\
 & & \quad add(S(0),S(0)) \doteq S(\bot)\} \\
T_7 & = & T_6 \cup \{add(0,S(0)) \doteq S(0), \\
 & & \quad double(0) \doteq 0, double(S(0)) \doteq S(\bot)\} \\
T_8 & = & T_7 \cup \{add(S(0),S(0)) \doteq S(S(0)), \\
 & & \quad main \doteq 0, main \doteq S(\bot)\} \\
T_9 & = & T_8 \cup \{double(S(0)) \doteq S(S(0))\} \\
T_{10} & = & T_9 \cup \{main \doteq S(S(0))\} \\
T_{11} & = & T_{10}
\end{array}
$$

Thus, a fixpoint is reached after 11 iterations. Note that the fixpoint contains 0 *and* $S(S(0))$ *as values of main but not* $S(0)$*, as expected for functional logic programs with call-time choice semantics.*

Next we state the formal properties of the transformation $T_{\mathcal{R},\mathcal{M}}$. In the following we assume a fixed functional logic program $\mathcal{R}$ and a set of main calls $\mathcal{M}$. The first important property of $T_{\mathcal{R},\mathcal{M}}$ ensures the existence of a fixpoint.

PROPOSITION 3.7. *The mapping $T_{\mathcal{R},\mathcal{M}}$ is continuous on $2^{\mathcal{E}}$.*

Thus, the mapping $T_{\mathcal{R},\mathcal{M}}$ has a least fixpoint $T_{\mathcal{R},\mathcal{M}} \uparrow \omega$ which is the least upper bound of $\{T_{\mathcal{R},\mathcal{M}} \uparrow k \mid k \geq 0\}$.

DEFINITION 3.8. The *least fixpoint semantics* of a program $\mathcal{R}$ w.r.t. a set of main calls $\mathcal{M}$ is defined as $C_{\mathcal{R},\mathcal{M}} = T_{\mathcal{R},\mathcal{M}} \uparrow \omega$. □

The following theorems justify the use of this fixpoint semantics to analyze programs, i.e., they show that interesting properties of concrete computations are correctly represented by $C_{\mathcal{R},\mathcal{M}}$. The first theorem shows that the least fixpoint contains all function calls (where unevaluated arguments are approximated by $\bot$) occurring in computations starting from main calls. For this purpose, we denote by

$$calls(s) = \{t|_p \mid s \rightarrowtail^* t,\ p \in \mathcal{FP}os(t)\}$$

the set of all function calls occurring in derivations starting with the term s.

THEOREM 3.9 (Call covering). *Let $s \in \mathcal{M}$ be an initial call and $f(e_1,\ldots,e_n) \in calls(s)$. Then $f(e_1^\bot,\ldots,e_n^\bot) \doteq \bot \in C_{\mathcal{R},\mathcal{M}}$.*

The next theorem states the soundness of the least fixpoint w.r.t. the rewrite relation $\rightarrowtail$, i.e., each equation contained in the least fixpoint corresponds to a rewriting derivation.

THEOREM 3.10 (Soundness). *If $s \doteq t \in C_{\mathcal{R},\mathcal{M}}$, then $s \rightarrowtail^* t$.*

Since we have constructed the least fixpoint w.r.t. a set of main calls, it represents all those rewriting derivations that start from these main calls. This is formally stated in the next theorem. Since operation-rooted subterms occurring during derivations might contain unevaluated arguments, we interpret these arguments w.r.t. $eval_{C_{\mathcal{R},\mathcal{M}}}$.

THEOREM 3.11 (Completeness). *Let $s \in \mathcal{M}$ be an initial call and $f(e_1,\ldots,e_n) \in calls(s)$. If $f(e_1,\ldots,e_n) \rightarrowtail^* u$, then there exist $t_i \in eval_{C_{\mathcal{R},\mathcal{M}}}(e_i)$, $i = 1,\ldots,n$, such that $f(t_1,\ldots,t_n) \doteq u^\bot \in C_{\mathcal{R},\mathcal{M}}$.*

This theorem implies that the least fixpoint contains all values of the main calls.

COROLLARY 3.12. *If $s \in \mathcal{M}$, $t \in \mathcal{T}(\mathcal{C}_\bot,\mathcal{X})$ with $s \rightarrowtail^* t$, then $s \doteq t \in C_{\mathcal{R},\mathcal{M}}$.*

The bottom elements of the least fixpoint could indicate unsuccessful computations, i.e., computations that do not deliver a result since no rule is applicable at some point or since they loop. Since the least fixpoint contains *all* approximations of result values, one can infer unsuccessful computations only if the bottom element is maximal. Therefore, we define the notion of maximal elements of an interpretation.

DEFINITION 3.13. $s \doteq t \in I$ is called *maximal* in an interpretation $I \subseteq \mathcal{E}$ if there is no $s \doteq t' \in I$ with $t' \neq t$ and $t \sqsubseteq t'$. The set of all maximal elements of an interpretation $I \subseteq \mathcal{E}$ is denoted by $max(I)$. □

COROLLARY 3.14. *If $s \in \mathcal{M}$ and $s \doteq \bot \in max(C_{\mathcal{R},\mathcal{M}})$, then s is not evaluable to a value, i.e., there is no $t \in \mathcal{T}(\mathcal{C},\mathcal{X})$ with $s \rightarrowtail^* t$.*

The latter corollary is useful to detect unsuccessful computations at compilation time if we can provide an appropriate and computable approximation of $C_{\mathcal{R},\mathcal{M}}$. This is the purpose of the next section.

4. Abstraction of Functional Logic Computations

Abstract interpretation (Cousot and Cousot 1977; Nielson et al. 1999) is a framework to construct program analyses by approximating the concrete transformation function of the operational semantics in order to obtain an approximation of the program's behavior. Since we already developed a fixpoint characterization of functional logic computations, we can easily apply the abstract interpretation framework in order to analyze functional logic programs.

We are mainly interested in the call patterns and input/output relation of functions occurring in the program. We already described this information by interpretations where the basic elements are equations of the form $f(t_1,\ldots,t_n) \doteq t$ with $f \in \mathcal{F}$ and $t_1,\ldots,t_n,t \in \mathcal{T}(\mathcal{C}_\bot,\mathcal{X})$, i.e., the concrete semantics is based on partial constructor terms. Therefore, an approximation of the concrete semantics can be based on *abstract partial constructor terms*. In order to provide a general framework that can be used with different abstract domains, we assume an abstract domain AC representing abstract partial constructor terms and a concretization function $\tau : AC \rightarrow 2^{\mathcal{T}(\mathcal{C}_\bot,\mathcal{X})}$ that maps abstract partial constructor terms into sets of concrete terms. To ensure finite computations on the abstract domain, one could require that AC is finite. However, this is not strictly necessary since there exist other methods to ensure termi-

nating abstract computations even in the presence of infinite abstract domains (Cousot and Cousot 1977).

The concrete domain $\mathbb{E}$ is the powerset $2^{\mathcal{E}}$ ordered by set inclusion as already introduced in Section 3. Similarly, the *abstract domain* $\mathbb{A}$ is the powerset of the base set

$$\{f(a_1,\ldots,a_n) \doteq a \mid f \in \mathcal{F}, a_1,\ldots,a_n,a \in AC\}$$

ordered by some set ordering. Based on the concretization function τ, we define a Galois insertion of $\mathbb{A}$ into $\mathbb{E}$ by

$$\begin{aligned}\alpha(I) &= \{f(a_1,\ldots,a_n) \doteq a \mid f(t_1,\ldots,t_n) \doteq t \in I \\ &\qquad \text{for all } t_1 \in \tau(a_1),\ldots,t_n \in \tau(a_n), t \in \tau(a)\} \\ \gamma(A) &= \{f(t_1,\ldots,t_n) \doteq t \mid f(a_1,\ldots,a_n) \doteq a \in A, \\ &\qquad t_1 \in \tau(a_1),\ldots,t_n \in \tau(a_n), t \in \tau(a)\}\end{aligned}$$

The theory of abstract interpretation shows that an optimal abstract version $T^{\alpha}_{\mathcal{R},\mathcal{M}} : \mathbb{A} \to \mathbb{A}$ of $T_{\mathcal{R},\mathcal{M}}$ can be defined by $T^{\alpha}_{\mathcal{R},\mathcal{M}} = \alpha \circ T_{\mathcal{R},\mathcal{M}} \circ \gamma$. In general, weaker abstract transformations are sufficient to ensure the correctness of abstract interpretation, i.e., if there is a continuous mapping $T^{\alpha} : \mathbb{A} \to \mathbb{A}$ with $\alpha \circ T_{\mathcal{R},\mathcal{M}} \circ \gamma \sqsubseteq T^{\alpha}$, then the least fixpoint (*lfp*) of $T_{\mathcal{R},\mathcal{M}}$ is correctly approximated by the least fixpoint of T^{α}, i.e., $C_{\mathcal{R},\mathcal{M}} \sqsubseteq \gamma(lfp(T^{\alpha}))$.

In order to define a mapping T^{α} for a specific abstract domain AC, one could try to follow the definition of $T_{\mathcal{R},\mathcal{M}}$ and replace the operations on concrete terms, like pattern matching for rule application or *eval* by corresponding abstract versions. In particular, the following abstract values and operations are sufficient to define a mapping T^{α}:

1. Abstract bottom element $\bot^{\alpha}$ with $\bot \in \tau(\bot^{\alpha})$
2. Abstract variable $\top^{\alpha}$ with $\tau(\top^{\alpha}) = \mathcal{T}(\mathcal{C}_{\bot},\mathcal{X})$
3. Abstract constructor application $c^{\alpha} : AC^n \to AC$ for each n-ary constructor c such that $\tau(c^{\alpha}(a_1,\ldots,a_n)) \supseteq \{c(t_1,\ldots,t_n) \mid t_i \in \tau(a_i), i = 1\ldots,n\}$
4. Abstract matching $match^{\alpha} : \mathcal{T}(\mathcal{C},\mathcal{X}), AC \to Sub(AC) \cup \{fail\}$ that approximates the concrete matching of linear constructor terms, where $Sub(AC)$ denotes the set of all *abstract substitutions* that are mappings from $\mathcal{X}$ into abstract values from AC with a finite domain.

The abstract bottom element is necessary when introducing unknown values during a fixpoint computation. The abstract variable is necessary for free variables occurring in a rewrite rule. In a concrete computation, such free variables are instantiated to all possible partial constructor terms (compare the definition of $[\mathcal{R}]_{\bot}$) so that we need an element in the abstract domain to capture the set of these values.

Based on these abstract entities, the definition of an abstract version $T^{\alpha}_{\mathcal{R},\mathcal{M}} : \mathbb{A} \to \mathbb{A}$ of $T_{\mathcal{R},\mathcal{M}}$ is straightforward by abstracting each base operation. First, we define the abstract initial equations $\mathcal{M}^{\alpha}_{\bot}$ for the set $\mathcal{M}$ of main calls by (the abstract evaluation of terms $eval^{\alpha}_{\varnothing}$ will be defined below)

$$\mathcal{M}^{\alpha}_{\bot} = \{f(eval^{\alpha}_{\varnothing}(\varnothing,t_1),\ldots,eval^{\alpha}_{\varnothing}(\varnothing,t_n)) \doteq \bot^{\alpha} \mid f(t_1,\ldots,t_n) \in \mathcal{M}\}$$

We define $T^{\alpha}_{\mathcal{R},\mathcal{M}}$ for all $I \in \mathbb{A}$ as follows (here we use a straightforward extension of $match^{\alpha}$ where we apply it to (abstract) constructor terms wrapped with a top-level operation symbol):

$$\begin{aligned}&T^{\alpha}_{\mathcal{R},\mathcal{M}}(I) = \\ &\quad \mathcal{M}^{\alpha}_{\bot} \cup \{s \doteq a \mid s \doteq t \in I,\ l \to r \in \mathcal{R}, \\ &\qquad\qquad match^{\alpha}(l,s) = \sigma \neq fail,\ a \in eval^{\alpha}_I(\sigma,r)\} \\ &\qquad \cup \{f(a_1,\ldots,a_n) \doteq \bot^{\alpha} \mid s \doteq t \in I,\ l \to r \in \mathcal{R}, \\ &\qquad\qquad match^{\alpha}(l,s) = \sigma \neq fail, \\ &\qquad\qquad p \in \mathcal{FP}os(r) \text{ with } r|_p = f(t_1,\ldots,t_n), \\ &\qquad\qquad a_i \in eval^{\alpha}_I(\sigma,t_i),\ i = 1,\ldots,n\}\end{aligned}$$

To distinguish between variables occurring in a rule's left-hand side and extra variables, we pass the abstract substitution of the rule matching to the *abstract evaluation of a (concrete) term* which is defined by

$$\begin{aligned}eval^{\alpha}_I(\sigma,x) &= \{\sigma(x)\} && \text{if } x \in \mathcal{D}om(\sigma) \\ eval^{\alpha}_I(\sigma,x) &= \{\top^{\alpha}\} && \text{if } x \notin \mathcal{D}om(\sigma) \\ eval^{\alpha}_I(\sigma,c(t_1,\ldots,t_n)) &= \{c^{\alpha}(a_1,\ldots,a_n) \mid a_i \in eval^{\alpha}_I(\sigma,t_i), \\ &\qquad\qquad i = 1,\ldots,n\} \\ eval^{\alpha}_I(\sigma,f(t_1,\ldots,t_n)) &= \{\bot^{\alpha}\} \cup \{a \mid a_i \in eval^{\alpha}_I(\sigma,t_i), \\ &\qquad\qquad i = 1,\ldots,n, \\ &\qquad\qquad f(a_1,\ldots,a_n) \doteq a \in I\}\end{aligned}$$

Note that we abstract extra variables by $\top^{\alpha}$, i.e., the set of all constructor terms. One can have the impression that this is a weak approximation, since many implementations of functional logic languages (e.g., based on needed narrowing (Antoy et al. 2000)) instantiate such variables only to those constructor terms that can be unified with the left-hand side of some rule. However, it has been shown (Antoy and Hanus 2006) that this is equivalent to a demand-driven instantiation to all constructor terms and, actually, there are implementations where extra variables are replaced by operations that nondeterministically evaluate to all constructor terms (Braßel and Huch 2007a). Therefore, our abstraction is reasonable and makes the analysis less dependent on particular implementation strategies.

In the next section, we show the application of this abstract interpretation framework with a specific abstract domain.

5. Abstract Interpretation with Depth-bounded Terms

An interesting finite abstraction of an infinite set of constructor terms are sets of terms up to a particular depth k, e.g., as already used in the abstract diagnosis of functional programs (Alpuente et al. 2002) or in the abstraction of term rewriting systems (Bert and Echahed 1995; Bert et al. 1993). Although this domain is useful in practice only for depth $k = 1$ (due to its quickly growing size for $k > 1$), we present the general case since we use it also with $k > 1$ in our initial experiments (see below). This domain is discussed here to provide a concrete application of our analysis framework. The application of our framework with other more sophisticated domains is left for future work.

In the domain of depth-bounded terms, subterms that exceed the given depth k are replaced by the specific constant $\top$ that represents any term, i.e., the abstract domain of *depth-k terms* is the set $AC = \mathcal{T}(\mathcal{C}_{\bot} \cup \{\top\},\varnothing)$ together

with the concretization function

$$\begin{array}{rcl} \tau(\bot) & = & \{\bot\} \\ \tau(\top) & = & \mathcal{T}(\mathcal{C}_\bot, \mathcal{X}) \\ \tau(c(t_1,\ldots,t_n)) & = & \{c(t'_1,\ldots,t'_n) \mid t'_i \in \tau(t_i), i = 1,\ldots,n\} \end{array}$$

Furthermore, the abstract entities according to the previous section are defined as follows:

1. Abstract bottom element: $\bot^\alpha = \bot$
2. Abstract variable: $\top^\alpha = \top$
3. Abstract constructor application:

   $$c^\alpha(t_1,\ldots,t_n) = cut_k(c(t_1,\ldots,t_n))$$

   where the cut operation cut_k is defined by

   $$\begin{array}{rcll} cut_k(\bot) & = & \bot & \\ cut_0(t) & = & \top & \text{if } t \neq \bot \\ cut_k(c(t_1,\ldots,t_n)) & = & c(cut_{k-1}(t_1),\ldots,cut_{k-1}(t_n)) & \\ & & & \text{if } k > 0 \end{array}$$

4. Abstract matching of linear constructor terms against depth-k terms:

   $$\begin{array}{rcl} match^\alpha(x,t) & = & \{x \mapsto t\} \\ match^\alpha(c(t_1,\ldots,t_n),\bot) & = & fail \\ match^\alpha(c(t_1,\ldots,t_n),\top) & = & \\ & & \{x \mapsto \top \mid x \in \mathcal{V}ar(c(t_1,\ldots,t_n)\} \\ match^\alpha(c(\cdots),d(\cdots)) & = & fail \quad \text{if } c \neq d \\ match^\alpha(c(t_1,\ldots,t_n),c(s_1,\ldots,s_n)) & = & \\ & & \begin{cases} \sigma_1 \circ \cdots \circ \sigma_n & \text{if } match^\alpha(t_i,s_i) = \sigma_i \\ & \sigma_i \neq fail, i = 1,\ldots,n \\ fail & \text{otherwise} \end{cases} \end{array}$$

As shown in Section 4, these definitions are sufficient to define a transformation on $\mathbb{A}$ whose least fixpoint approximates all concrete computations. Since the abstract domain is finite, the abstract least fixpoint can be computed in a finite number of steps.

Example 5.1. *Consider the program and main call of Example 3.6. With a depth bound of 3, our abstract semantics computes exactly the same fixpoint as shown in Example 3.6. With a depth bound of 1 (i.e., each term is abstracted to its top-level constructor), the following fixpoint is computed:*

$\{add(\bot,\bot) \doteq \bot,$
$add(0,0) \doteq \bot,$
$add(0,0) \doteq 0,$
$add(S(\top),S(\top)) \doteq \bot,$
$add(S(\top),S(\top)) \doteq S(\bot),$
$add(S(\top),S(\top)) \doteq S(\top),$
$add(\top,S(\top)) \doteq \bot,$
$add(\top,S(\top)) \doteq S(\bot),$
$add(\top,S(\top)) \doteq S(\top),$
$coin \doteq \bot,$
$coin \doteq 0,$
$coin \doteq S(\top),$
$double(\bot) \doteq \bot,$
$double(0) \doteq \bot,$
$double(0) \doteq 0,$
$double(S(\top)) \doteq \bot,$
$double(S(\top)) \doteq S(\bot),$
$double(S(\top)) \doteq S(\top),$
$main \doteq \bot,$
$main \doteq 0,$
$main \doteq S(\bot),$
$main \doteq S(\top)\}$

In this example, the computed abstract information is not very useful. More interesting examples are situations where one has not the complete concrete information available at analysis time. This is the case when unknown values, e.g., logic variables are present. For instance, consider the rule

$$main \to add(x, add(y, S(z)))$$

where x, y, z are extra variables. With a depth bound of 1, our abstract semantics computes the fixpoint as follows for the set of main calls $\{main\}$:

$$\begin{array}{rcl} T_0 & = & \varnothing \\ T_1 & = & \{main \doteq \bot\} \\ T_2 & = & T_1 \cup \{add(\top,\bot) \doteq \bot, add(\top,S(\top)) \doteq \bot\} \\ T_3 & = & T_2 \cup \{add(\top,\bot) \doteq S(\bot), add(\top,S(\top)) \doteq S(\bot), \\ & & \quad add(\top,S(\top)) \doteq S(\top)\} \\ T_4 & = & T_3 \cup \{add(\top,\bot) \doteq S(\top), add(\top,S(\bot)) \doteq \bot, \\ & & \quad main \doteq S(\bot), main \doteq S(\top)\} \\ T_5 & = & T_4 \cup \{add(\top,S(\bot)) \doteq S(\bot)\} \\ T_6 & = & T_5 \cup \{add(\top,S(\bot)) \doteq S(\top)\} \\ T_7 & = & T_6 \end{array}$$

Thus, the fixpoint contains the equation $main \doteq S(\top)$ which shows that the result of evaluating the main call is always headed by the constructor S, i.e., it is a positive number.

As one can see in these examples, the abstract semantics contains many elements where one is less evaluated than the other. For instance, the previous example contains the elements

$add(\top,\bot) \doteq \bot$
$add(\top,S(\top)) \doteq \bot$
$add(\top,S(\top)) \doteq S(\top)$

where the first two contain less information than the last element. This is due to the fact that we do not know at analysis time how far a function call will be evaluated during run time. Since more elements in the abstract semantics require more computation and one is usually interested in more precise values for call patterns, it is reasonable to transfer the approximation ordering of Definition 2.2 also to abstract values, i.e., compute only the maximal elements of the abstract semantics and remove smaller elements (e.g., the first two elements above) during the fixpoint computation. The positive effect of this improvement will be shown in the practical evaluation below.

As a further example, consider the function

$$\begin{array}{rcl} f(0) & \to & 0 \\ f(S(x)) & \to & f(f(x)) \end{array}$$

from (Bert et al. 1993). With a depth bound of 1 and the set of main calls $\{f(x)\}$, the least fixpoint of our abstract

semantics contains the equation $f(\top) \doteq 0$ indicating that f is a constant function.

The final example in this section includes nondeterministic operations as well as extra variables. The operation *half* (taken from (Antoy 1997)) is nondeterministic due to two possible roundings of odd numbers, and the *main* operation checks whether doubling a half of some natural number is one (where *double* is defined as above):

$$\begin{array}{rcl} half(0) & \to & 0 \\ half(S(0)) & \to & 0 \\ half(S(0)) & \to & S(0) \\ half(S(S(x))) & \to & S(half(x)) \\ \\ isOne(0) & \to & False \\ isOne(S(0)) & \to & True \\ isOne(S(S(x))) & \to & False \\ \\ main & \to & isOne(double(half(x))) \end{array}$$

With a depth bound of 2, our analysis computes a fixpoint with $main \doteq False$ as the only maximal equation for *main*. Thus, the soundness of our analysis implies that doubling a half of a natural is always different from one. Note that the consideration of a call-time choice semantics is relevant here since *main* might reduce to $True$ w.r.t. a traditional term rewriting semantics.

Although our semantics is designed to compute call patterns w.r.t. some main calls, the examples show that we can also use our semantics to approximate the complete input/output relation for all functions in the sense of (Alpuente et al. 2002): we just have to put an initial call of the form $f(x_1, \ldots, x_n)$ (where the arguments x_i are pairwise distinct variables) for each n-ary function f into the set of main calls.

6. Extensions for Application Programs

Our analysis presented so far is based on programs described by first-order constructor-based term rewriting systems. In order to apply our analysis to realistic functional logic programs, we have to support two additional concepts apparent in modern functional logic languages like Curry (Hanus 1997, 2006): higher-order functions and primitive operations. This will be discussed in this section.

6.1 Higher-order Features

Higher-order features of functional (logic) languages can be supported through a transformation into first-order programs by defining a predicate *apply* that implements the application of an arbitrary function occurring in the program to an expression. This technique is also known as "defunctionalization" (Reynolds 1972) (and conceptually also used in logic languages (Warren 1982)) and enough to support the higher-order features of current functional (logic) languages (e.g., lambda abstractions can be replaced by new function definitions). For instance, consider a program with the operations *add* and *double* of Example 2.1 and the following rules (here we use the Curry/Haskell notation for lists):

$$\begin{array}{rcl} map(f, []) & \to & [] \\ map(f, x : xs) & \to & apply(f, x) : map(f, xs) \\ main(xs) & \to & map(apply(add, S(0)), map(double, xs)) \end{array}$$

map is the standard higher-order function that applies a function to each element of a list. The application operation is defined by the following rules that can be automatically derived from the program:

$$\begin{array}{rcl} apply(add, x1) & \to & add(x1) \\ apply(add(x1), x2) & \to & add(x1, x2) \\ apply(double, x1) & \to & double(x1) \\ apply(map, x1) & \to & map(x1) \\ apply(map(x1), x2) & \to & map(x1, x2) \\ apply(main, x1) & \to & main(x1) \end{array}$$

Thus, for each n-ary function, n corresponding *apply* rules are defined. After adding the *apply* rules to the program, we obtain a standard program, i.e., a first-order term rewriting system[4] so that we can apply our analysis to it. For instance, a depth-k analysis with a depth bound of 2 computes the following least fixpoint:

$$A = \{main \doteq S(\top) : (\top : \top),\ main \doteq S(\top) : [], \ldots\}$$

Thus, one can infer that the first element of the computed list is always a positive integer. Furthermore, the equations for *apply* contained in A reflect the functions that are used in a higher-order manner, e.g., all *apply* equations contained in A have left-hand sides of the form $apply(add(S(\top)), \ldots)$ or $apply(double, \ldots)$ so that we know that only the second and third rule defining *apply* are used during run time. This information could be used to optimize the code of programs based on the defunctionalization technique, e.g., as in Prolog-based implementations (Antoy and Hanus 2000).

6.2 Primitive Operations

Real world programs contain various calls to primitive operations that are not explicitly defined by rewrite rules, e.g., arithmetic operations on integers or floats, I/O operations etc. Although some of these operations can be conceptually explained with infinite sets of rewrite rules (Bonnier and Maluszynski 1988), we need a constructive method to deal with such operations at analysis time. Since we only need to approximate the meaning of such operations, we can approximate an n-ary primitive operation f by the following rewrite rule:

$$f(x_1, \ldots, x_n) \to x$$

(where $x_1, \ldots, x_n, x$ are pairwise different variables). This rule specifies that the result of a call to the primitive function is arbitrary. Of course, one could add more precise descriptions for specific functions and abstract domains, but this "least specific" description is always sufficient, in particular, for input operations where one does not know the user input at analysis time.

I/O operations can be treated in declarative languages by the well-known monadic approach where I/O actions are considered as transformations on the outside world (Wadler 1997). Thus, each I/O action takes a state of the world and returns a pair consisting of the desired result value and a new state of the world. For instance, the operation *getChar* that reads and returns the next character from the keyboard takes a state of the world and returns a character from the

[4] Note that n-ary functions applied to less than n arguments are considered as constructors so that the resulting program is constructor-based.

Program	Rules	Depth k	FP Size (# equations)	MFP Size (# equations)	Speedup (FP time / MFP time)	MFP Time (ms)
`addadd`	3	1	12	2	5.0	4
`addlast`	7	2	20	7	4.5	4
`bertconc`	3	1	12	2	15.0	1
`bertf0`	3	1	7	3	8.0	1
`doublecoin`	6	1	22	9	2.7	4
`family`	29	1	43	29	2.2	40
`halfdouble`	11	2	60	15	6.9	12
`head`	12	1	90	19	14.3	28
`mapadddouble`	12	2	424	35	102.4	72
`readfile`	34	8	160	17	28.5	16
`risers`	9	1	13	9	2.3	4
`tails`	15	1	31	7	13.3	4

Table 1. Analysis of example programs

input and a state of the world. Thus, we can approximate this operation by the following rule (where we ignore the fact that the state of the world is usually changed):

$$getChar(w) \to (c, w)$$

The basic sequence combinator on I/O actions is usually called *bind*: $bind(a, f, w)$ executes action a on the given world w and applies the action function f to the result of the first action. Since each action returns a value together with a new state of the world, we use an auxiliary operation $bind'$ to decompose these items and apply the action function f, i.e., we define these operations by the following rules:

$$\begin{aligned} bind(a, f, w) &\to bind'(apply(a, w), f) \\ bind'((r, w), f) &\to apply(apply(f, r), w) \end{aligned}$$

With this representation of primitive operations, we can apply our analysis also to programs with I/O operations. For instance, consider the following program where the *main* call reads a character from standard input and returns the contents of a file with this name stored in the directory "`/tmp`" (note that strings are represented as lists of characters):

$$\begin{aligned} conc([], ys) &\to ys \\ conc(x : xs, ys) &\to x : conc(xs, ys) \\ tmpDir &\to \texttt{'/'} : \texttt{'t'} : \texttt{'m'} : \texttt{'p'} : \texttt{'/'} : [] \\ readTmpFile(c) &\to readFile(conc(tmpDir, c : [])) \\ main(w) &\to bind(getChar, readTmpFile, w) \end{aligned}$$

where *readFile* is a primitive to read the contents of a file. Applying a depth-k analysis with bound 8 to this program returns the following abstract equation for the primitive *readFile* (all other equations for *readFile* contain less information):

$$readFile(\texttt{'/'} : \texttt{'t'} : \texttt{'m'} : \texttt{'p'} : \texttt{'/'} : \top : [], \top) \doteq \cdots$$

Thus, we can infer that this program only accesses files in the directory `/tmp`, as expected. Of course, verifying more interesting safety properties requires other sophisticated domains, e.g., regular types (Dart and Zobel 1992).

6.3 Practical Evaluation

In order to provide some data about the practical application of our framework, we have implemented the proposed fixpoint analysis as a prototype system. The analyzer is generic w.r.t. the abstract domain, i.e., the operations implementing the abstract domain as described in Section 4 are passed to the generic fixpoint computation. The analyzer is implemented in Curry in a straightforward way where the depth-bounded term analysis, as described in Section 5, is used as an abstract domain. It is not very efficient but a high-level implementation to compare the analysis of simple programs. The analyzer is executed with the Curry implementation KiCS (Braßel and Huch 2007b) that compiles Curry programs into Haskell programs executed by the Glasgow Haskell Compiler.

Table 1 contains, for various example programs, the number of rewrite rules (including rules for primitive operations), the depth bound used for the analysis, the number of equations of the least fixpoint (FP Size) and the least fixpoint containing only maximal elements (MFP Size) as well as the speedup obtained by computing only maximal elements instead of all elements of the abstract semantics (see discussion in Section 5 above), and the time in milliseconds to compute the fixpoint of maximal elements (note that values lower than 10 ms are not quite accurate). Timings were done on a 3.0 Ghz Linux PC (AMD Athlon). `addadd` is the double call to *add* of Example 5.1, `addlast` concatenates a two-element list at the end of an arbitrary list and checks whether the resulting list is a one-element list, `bertf0` and `bertconc` are the two examples of (Bert et al. 1993), `doublecoin` was presented in Example 3.6, `family` is a family database represented by nondeterministic operations where a recursive ancestor function is evaluated, `halfdouble` is the final example of Section 5, `head` is an example to verify the correct use of the partially defined function `head` in various situations, `mapadddouble` and `readfile` are the examples of Sections 6.1 and 6.2, respectively, and `risers` and `tails` are the examples of (Mitchell and Runciman 2007) to verify safe pattern matching in Haskell by static analysis. All programs are available with the implementation of the prototype.

7. Conclusions and Related Work

We have presented an approach to analyze call patterns and their computed results occurring in functional logic computations. For this purpose, we have introduced a new fixpoint

characterization of functional logic computations w.r.t. a set of main calls. An approximation of the concrete behavior can be obtained by approximating the operations used to compute this fixpoint. If the abstract domain of this approximation is finite, the complete approximation can be computed in a finite amount of time. We have demonstrated the application of this idea by a depth-bounded term analysis. Furthermore, it has been shown how to cover higher-order features and primitive operations in order to approximate realistic programs. The analysis results can be used to optimize programs (e.g., for partial evaluation), to catch pattern-match errors at compile time, or to verify safety conditions of programs. We have implemented this fixpoint analysis as a prototype system which is able to compute all examples in this paper.

Although our approach is the first one to approximate call patterns in functional logic computations, there are various related works. We have already mentioned the works on minimal function graphs (e.g., (Gallagher and Bruynooghe 1991; Jones and Mycroft 1986; Winsborough 1992)) that have similar aims as this paper, i.e., the computation of function or predicate calls and their corresponding results. Jones and Mycroft (1986) introduced this notion for strict functional languages which has been also applied to logic programs (e.g., (Gallagher and Bruynooghe 1991; Winsborough 1992)) or extended to lazy functional programs (Jones and Andersen 2007). Some works proposed rather general frameworks to compute minimal function graphs (Gallagher and Bruynooghe 1991), but the application of this idea to a concrete programming language must take into account the details of its operational semantics. In the case of modern functional logic languages, the combination of nondeterministic and non-strict computations is essential. Although nondeterministic computations are handled by logic programs and non-strict computations by functional programs, the combination of both requires a carefully designed calculus (CRWL (González-Moreno et al. 1999)). Therefore, one can not simply combine the works in both areas (Gallagher and Bruynooghe 1991; Jones and Andersen 2007; Winsborough 1992) so that we defined an appropriate fixpoint characterization of the intermediate states of CRWL computations starting from a set of initial calls.

Bert et al. (1993) proposed abstract rewriting (which was extended in (Bert and Echahed 1995) to conditional term rewriting systems). The objective of abstract rewriting is the approximation of the top-level constructors of term evaluations in order to improve E-unification. For this purpose, they associate to a set of rewrite rules an abstract rewrite system that is able to compute finite approximations of top-level constructors. However, their framework is restricted to constructor-based, confluent and terminating rewrite systems without partial functions and, therefore, too limited for functional logic programming.

Alpuente et al. (2002) presented a fixpoint characterization of the input/output relation of functions defined by term rewriting systems in order to detect program errors. Their approach is not goal-oriented, i.e., does not approximate call patterns, and uses the classical notion of rewriting instead of a rewrite relation suitable for modern functional logic languages with non-strict, nondeterministic operations. A fixpoint characterization of functional logic programs based on CRWL was presented in (Molina-Bravo and Pimentel 1997). Although it covers the input/output relation of functions similarly to our semantics, it is not goal-oriented and, thus, not suitable to approximate call patterns.

Albert et al. (2005) proposed a method to verify safety properties of logic programs in the spirit of proof-carrying code. Their method derives call and success patterns w.r.t. sophisticated abstract domains. Since they analyze logic programs, their analysis is based on and/or graphs (as well as similar methods for logic programs, e.g., (Bruynooghe 1991)). Such a method is not applicable to functional logic programs due their demand-driven evaluation strategy.

A different method with similar goals has been presented by Mitchell and Runciman (2007). In order to check a Haskell program for the absence of pattern-match errors due to functions with incomplete patterns in their definitions, they propose a static checker that extracts constraints from pattern-based definitions and tries to solve them by simplification and fixpoint iteration. Since they do not use the framework of abstract interpretation, the correctness of their approach is not proved. Furthermore, they consider only the restricted class of purely functional programs rather than general functional logic programs as in this paper. However, it is interesting to note that such kinds of pattern-match errors can be also easily detected by our framework. For instance, one can complete all partial function definitions by rules for the missing patterns that rewrite to some error function. For instance, if the function *head* that extracts the first element of a list is defined by

$$head(x : xs) \;\to\; x$$

one can complete its definition by adding the rule

$$head([]) \;\to\; matchError$$

If our analysis shows that the function *matchError* will not be called, i.e., there is no equation of the form $matchError \doteq \cdots$ in the least fixpoint, the correctness of our framework ensures the absence pattern-match errors. Actually, we have successfully applied our analysis with depth-bounded terms to the examples given in (Mitchell and Runciman 2007), where it was sufficient to use a depth of $k = 1$ (see Table 1).

For future work, we intend to implement the fixpoint analysis more efficiently in order to apply it to larger programs. Furthermore, we want to apply this analysis with other domains that could be more suitable to verify safety properties of programs.

References

E. Albert, G. Puebla, and M. Hermenegildo. An abstract interpretation-based approach to mobile code safety. *Electronic Notes in Theoretical Computer Science*, 132:113–129, 2005.

M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2002)*, pages 1–16. Springer LNCS 2664, 2002.

S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic*

and Logic Programming (ALP'97), pages 16–30. Springer LNCS 1298, 1997.

S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.

S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

D. Bert and R. Echahed. Abstraction of conditional term rewriting systems. In *Proc. of the 1995 International Logic Programming Symposium*, pages 147–161. MIT Press, 1995.

D. Bert, R. Echahed, and M. Østvold. Abstract rewriting. In *Proc. Third International Workshop on Static Analysis*, pages 178–192. Springer LNCS 724, 1993.

S. Bonnier and J. Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pages 311–326. MIT Press, 1988.

B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007a.

B. Braßel and F. Huch. The Kiel Curry system KiCS. In *Proc. 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007)*, pages 215–223. Technical Report 434, University of Würzburg, 2007b.

M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming (10)*, pages 91–124, 1991.

J.M. Cleva, J. Leach, and F.J. López-Fraguas. A logic programming approach to the verification of functional-logic programs. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 9–19. ACM Press, 2004.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.

R. Echahed and J.-C. Janodet. Admissible graph rewriting and narrowing. In *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 325–340, 1998.

J.P. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3/4):305–334, 1991.

J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

M. Hanus. Call pattern analysis for functional logic programs. Technical report 0803, Christian-Albrechts-Universität Kiel, 2008.

M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.

N. Jones and N. Andersen. Flow analysis of lazy higher order functional programs. *Theoretical Computer Science*, 375 (1-3):120–136, 2007.

N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 296–306, 1986.

F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 197–208. ACM Press, 2007.

N. Mitchell and C. Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming*, volume 6, pages 15–30. Intellect, 2007.

J.M. Molina-Bravo and E. Pimentel. Modularity in functional-logic programming. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 183–197. MIT Press, 1997.

F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

S. Peyton Jones. Call-pattern specialization for Haskell programs. In *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pages 327–337, 2007.

J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.

P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.

W. Winsborough. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming*, 13(2&3):259–290, 1992.