Dynamic Detection of Determinism in Functional Logic Languages

Rita Loogen and Stephan Winkler
RWTH Aachen, Lehrstuhl für Informatik II
Ahornstraße 55, W-5100 Aachen, Germany
e-mail: {rita,winkler}@zeus.informatik.rwth-aachen.de

Abstract

Programs in functional logic languages usually have to satisfy a nonambiguity condition, that semantically ensures completeness of conditional narrowing and pragmatically ensures that the defined (non-boolean) functions are deterministic and do not yield different result values for the same argument tuples. The nonambiguity condition allows the dynamic detection of determinism in implementations of functional logic languages. In this paper we show how to achieve this and what can be gained by this optimization.

1 Introduction

Functional logic languages are extensions of functional languages with principles derived from logic programming [Reddy 87]. While their syntax almost looks like the syntax of conventional functional languages, their operational semantics is based on narrowing, an evaluation mechanism that uses unification instead of pattern matching for parameter passing. Narrowing is a natural extension of reduction to incorporate unification. It means applying the minimal substitution to an expression in order to make it reducible, and then to reduce it.

In general, functional logic programs have to satisfy a nonambiguity condition, that semantically ensures completeness of narrowing [Bosco et al. 89b], [Kaplan 84] and pragmatically ensures that the defined (non-boolean valued) functions are deterministic and do not yield various different result values for the same argument tuples.

The aim of this paper is to show how to make use of the nonambiguity condition in order to perform a dynamic detection of determinism in implementations of functional logic languages. The techniques that we present are independent from the implementation method. Especially for the implementation of narrowing, a lot of machine models have been proposed in the literature. A coarse classification distinguishes machines based on the Warren Abstract Machine (WAM) [Warren 83], see e.g. [Balboni et al. 89, Bosco et al. 89a], [Hanus 90], [Mück 90], and extensions of functional reduction machines [Kuchen et al. 90, Moreno et al. 90], [Loogen 91], [Chakravarty, Lock 91]. In this paper we focus on the stack narrowing machine of [Loogen 91]. It is however no problem to incorporate a corresponding determinism check in the other machine models.

The effect of the determinism check can be compared to a dynamic and safe version of the cut operator known from Prolog. It enables a dynamic reduction of the computation tree, whenever it is safe to do so, due to the nonambiguity of programs. It is important to note that the boolean valued functions corresponding to Prolog predicates always satisfy the nonambiguity condition. Thus, the dynamic detection of determinism applies also to Prolog and logic programs. The main application is however the optimized handling of purely functional computations in functional logic languages.

The paper is organized as follows. In Section 2, we describe the syntax and operational semantics of the functional logic language BABEL [Moreno,Rodríguez 88,89]. This section also contains a discussion of the nonambiguity condition. Section 3 contains an overview of the main components of the stack narrowing machine of [Loogen 91] and the compilation of BABEL programs into code for this machine. We explain the detection of determinism in two stages. In Section 4 the simple case of function definitions without guards, which correspond e.g. to unconditional function definitions and Prolog facts, is handled, while the general case is discussed in Section 5. In both sections we first show under which conditions a rule application is deterministic, i.e. no other rule for the corresponding function symbol needs to be considered. This is
formally proved using the nonambiguity condition. Then we show how to check these conditions in the narrowing machine. Sections 6 and 7 finally contain some conclusions and a discussion of related work, respectively.

2 The Functional Logic Language BABEL

In this section, we introduce the first order weakly typed subset of the functional logic language BABEL [Moreno, Rodríguez 88, 89]. BABEL is a higher order polymorphically typed functional logic language based on a constructor discipline. Its operational semantics is based on narrowing.

2.1 Syntactic Domains

Let $DC = \bigcup_{n \in \mathbb{N}} DC^n$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ be ranked alphabets of constructors and function symbols, respectively. We assume the nullary constructors ‘true’ and ‘false’ to be predefined. Predefined function symbols are the boolean operators and the equality operator. In the following, letters $c, d, e \ldots$ are used for constructors and the letters $f, g, h \ldots$ for function symbols.

The following syntactic domains are distinguished:

- **Variables** $X, Y, Z \ldots \in \text{Var}$
- **Terms** $s, t \ldots \in \text{Term}$:
  \[
  t ::= \begin{array}{ll}
  X & \text{% Variable} \\
  c(t_1, \ldots, t_n) & \text{% } c \in DC^n, n \geq 0
  \end{array}
  \]
- **Expressions** $M, N \ldots \in \text{Exp}$:
  \[
  M ::= \begin{array}{ll}
  X & \text{% Variable} \\
  c(M_1, \ldots, M_n) & \text{% } c \in DC^n, n \geq 0 \\
  f(M_1, \ldots, M_n) & \text{% } f \in FS^n, n \geq 0 \\
  (B \rightarrow M) & \text{% guarded expression} \\
  (B \rightarrow M_1 \sqcup M_2) & \text{% conditional expression}
  \end{array}
  \]

$B \rightarrow M$ and $B \rightarrow M_1 \sqcup M_2$ are intended to mean "if $B$ then $M$ else not defined" and "if $B$ then $M_1$ else $M_2$", respectively.

2.2 Functional Logic Programs

A BABEL program consists of a finite set of defining rules for the not predefined function symbols in $FS$. Let $f \in FS^n$. Each defining rule for $f$ must have the form:

\[
\begin{array}{c}
\{f(t_1, \ldots, t_n) := \{B \rightarrow\} M\} \\
\text{lhs} \quad \text{optional guard body} \quad \text{rhs}
\end{array}
\]

and satisfy the following conditions:

1. **Term Pattern**: $t_i \in \text{Term}$.
2. **Left Linearity**: $f(t_1, \ldots, t_m)$ does not contain multiple variable occurrences.
3. **Restrictions on free variables**: Variables occurring in the right hand side (rhs), but not in the left hand side (lhs), are called free. Occurrences of free variables are allowed in the guard, but not in the body.
4. **Nonambiguity**: Given any two rules for the same function symbol $f$:

   \[
   f(t_1, \ldots, t_n) := \{B \rightarrow\} M \quad \text{and} \quad f(s_1, \ldots, s_n) := \{C \rightarrow\} N.
   \]

   one of the three following cases must hold:

   (a) **No superposition**: $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$ are not unifiable.