Implementing Encapsulated Search for a Lazy Functional Logic Language

Wolfgang Lux

Universität Münster
wlux@uni-muenster.de

Abstract. A distinguishing feature of logic and functional logic languages is their ability to perform computations with partial data and to search for solutions of a goal. Having a built-in search strategy is convenient but not always sufficient. For many practical applications the built-in search strategy (usually depth-first search via global backtracking) is not well suited. Also the non-deterministic instantiation of unbound logic variables conflicts with the monadic I/O concept, which requires a single-threaded use of the world.
A solution to these problems is to encapsulate search via a primitive operator `try`, which returns all possible solutions to a search goal in a list. In the present paper we develop an abstract machine that aims at an efficient implementation of encapsulated search in a lazy functional logic language.

1 Introduction

A distinguishing feature of logic and functional logic languages is their ability to perform computations with partial data and to search for solutions of a goal. Having a built-in search strategy to explore all possible alternatives of a non-deterministic computation is convenient but not always sufficient. In many cases the default strategy, which is usually a depth-first traversal using global backtracking, is not well suited to the problem domain. In addition, global non-determinism is incompatible with the monadic I/O concept [PW93]. In this concept the outside world is encapsulated in an abstract data type and actions are provided to change the state of the world. These actions ensure that the world is used in a single-threaded way. Global non-determinism would defeat this single-threaded interaction with the world. Encapsulated search [SSW94,HS98] provides a remedy to both of these problems.

In this paper we develop an abstract machine for the functional logic language Curry [Han99]. Curry is a multi-paradigm language that integrates features from functional languages, logic languages and concurrent programming. Curry uses lazy evaluation of expressions and supports the two most important operational principles developed in the area of functional logic programming, narrowing and residuation. Narrowing [Red85] combines unification and reduction. With narrowing unbound logic variables in expressions may be instantiated

A. Middeldorp, T. Sato (Eds.): FLOPS'99, LNCS 1722, pp. 100–113, 1999.

non-deterministically. With the residuation strategy [ALN87], on the other hand, the evaluation of expressions containing unbound logic variables may be delayed until these variables are sufficiently instantiated by other parts of the program. Unfortunately this strategy is known to be incomplete [Han92].

Our abstract machine is a stack-based graph reduction machine similar to the G-machine [Joh84] and the Babel abstract machine [KLMR92]. Its novel feature is the implementation of encapsulated search in an efficient manner that is compatible with the overall lazy evaluation strategy of Curry. Due to the lack of space we restrict the presentation of the abstract machine to the implementation of the encapsulated search. The full semantics of the machine can be found in [LK99]. In the rest of the paper we will assume some familiarity with graph reduction machines for functional and functional logic languages [Joh84,KLMR92].

The rest of this paper is organized as follows. In the next section the computation model of Curry is briefly reviewed and the search operator `try` is introduced. The third section introduces the abstract machine. In section 4 an example is presented, which demonstrates the operation of the abstract machine. The sixth section presents some runtime results for our prototypical implementation. The last two sections present related work and conclude.

2 The Computation Model of Curry

Curry uses a syntax that is similar to Haskell [HPW92], but with a few additions.

The basic computational domain of Curry is a set of data terms. A data term t is either a variable x or constructed from an n-ary data constructor c, which is applied to n argument terms:

$$t ::= x \mid c\, t_1 \ \ldots\ t_n$$

New data constructors can be introduced through data type declarations, e.g. `data Nat = Zero | Succ Nat`. This declaration defines the nullary data constructor `Zero` and the unary data constructor `Succ`.

An expression e is either a variable x, a data constructor c, a defined function f, or the application of an expression e_1 to an argument expression e_2:

$$e ::= x \mid c \mid f \mid e_1\, e_2$$

Curry provides a predefined type `Constraint`. Expressions of this type are checked for satisfiability. The predefined nullary function `success` reduces to a constraint that is always satisfied. An equational constraint e_1`=:=`e_2 is satisfied, if e_1 and e_2 can be reduced to the same (finite) data term. If e_1 or e_2 contain unbound logic variables, an attempt will be made to unify both terms by instantiating variables to terms. If the unification succeeds, the constraint is satisfied. E.g the constraint `Succ m=:=Succ Zero` can be solved by binding `m` to `Zero`, if `m` is an unbound variable.

Functions are defined by conditional equations of the form

$$f\ t_1\ \ldots\ t_n \mid g = e$$