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

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

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

where the so-called guard g is a constraint. A conditional equation for the function f is applicable in an expression $f\ e_1\ \ldots\ e_n$, if the arguments $e_1, \ldots, e_n$ match the patterns $t_1, \ldots, t_n$ and if the guard is satisfied. The guard may be omitted, in which case the equation is always applicable if the arguments match.

A `where` clause can be added to the right hand side of an equation to provide additional local definitions, whose scope is the guard g and the expression e. Unbound logic variables can be introduced by the special syntax[1] `where` $x_1, \ldots, x_n$ `free`

A Curry program is a set of data type declarations and function definitions. The following example defines a predicate and an addition function for natural numbers.

```
nat Zero     = success      add Zero     n = n
nat (Succ n) = nat n        add (Succ m) n = add m (Succ n)
```

2.1 Reduction strategy

An *answer expression* is a pair of an expression e and a substitution σ that describes the bindings for the free variables of the expression e. An answer expression is written as $\sigma \,[]\, e$. The computational domain is a set of answer expressions. E.g. the solution of the goal `f x` where `x` is a free variable and the function `f` is defined by

```
f 0 = 1
f 1 = 2
```

is the set

$$\{\{\mathtt{x} \mapsto 0\} \,[]\, 1, \{\mathtt{x} \mapsto 1\} \,[]\, 2\}$$

A single computation step performs a reduction of exactly one unsolved expression in the set and returns a set of answer expressions for it. If this set has exactly one element, the computation step was deterministic. Otherwise the the computation either failed and the set is empty or a non-deterministic computation step was performed.

An attempt to reduce an expression $f\ e_1 \ldots e_n$ triggers the evaluation of $e_1, \ldots, e_n$ according to a left-to-right pattern-matching strategy. Thus, in order to reduce the expression `nat (add Zero Zero)`, the argument `(add Zero Zero)` is reduced to head normal form (i.e., a term without a defined function symbol at the top) in order to select an applicable equation of the function `nat`.

If an argument is an unbound variable, as e.g. in `nat n`, the further computations depend on the evaluation mechanism to be used. When narrowing is used, a non-deterministic computation step is performed that yields a set which contains an answer expression for each possible binding of the variable. In the example, the reduction would yield the set

$$\{\{\mathtt{n} \mapsto \mathtt{Zero}\} \,[]\, \mathtt{success}, \{\mathtt{n} \mapsto \mathtt{Succ\ m}\} \,[]\, \mathtt{nat\ m}\}$$

[1] The same syntax is also applicable to `let` expressions.

where `m` is a fresh, unbound variable. If residuation is used, the evaluation is delayed until the variable has been instantiated by some other concurrent computation. By default, constraint functions use narrowing as their evaluation mechanism, while all other functions use residuation. The user can override these defaults by evaluation annotations.

The concurrent evaluation of subexpressions is introduced by the concurrent conjunction c_1 `&` c_2 of two constraints, which evaluates the constraints c_1 and c_2 concurrently and is satisfied iff both are satisfiable. For example we might implement the subtraction on natural numbers with two concurrent computations. The second acts as generator of natural numbers, while the first checks if a generated number added to second argument of the subtraction yields the first argument of the subtraction:

```
sub m n | add s n=:=m & nat s = s where s free
```

2.2 Encapsulated Search

The use of monadic I/O, which assumes a single-threaded interaction with the world, conflicts with the non-determinism stemming from the instantiation of unbound variables. For that reason Curry provides the primitive search operator `try :: (a->Constraint) -> [a->Constraint]` that allows to confine the effects of non-determinism. This operator can also be used to implement find-all predicates in the style of Prolog, but with the possibility to use other strategies than the built-in depth first search [HS98].

The argument of `try` is the *search goal*. The argument of the search goal can be used to constrain a goal variable by the solutions of the goal. The result of `try` is either an empty list, denoting that the reduction of the goal has failed, or it is a singleton list containing a function `\x -> g`, where `g` is a satisfiable constraint (in solved form), or the result is a list with at least two elements if the goal can be reduced only by a non-deterministic computation step. The elements of this list are search goals that represent the different alternatives for the reduction of the goal immediately after this non-deterministic step.

For instance, the reduction of

```
try (\x -> let s free in add s x=:=Succ Zero & nat s)
```

yields the list

```
[\x -> add Zero x=:=Succ Zero & success,
 \x -> let t free in add (Succ t) x=:=Succ Zero & nat t]
```

3 The Abstract Machine

3.1 Overview

The abstract machine developed in this paper is a stack based graph reduction machine, that implements a lazy evaluation strategy. The concurrent evaluation

of expressions is implemented by assigning each concurrent expression to a new thread. The main contribution of the machine is its lazy implementation of encapsulated search, which is described in more detail below.

The state space of the abstract machine is shown in Fig. 1. The state of the abstract machine is described by an 8-tuple $\langle c, ds, es, hp, H, rq, scs, tr\rangle$, where c denotes a pointer to the instruction sequence to be executed. The data stack ds is used to supply the arguments during the construction of data terms and function applications. The environment stack es maintains the environment frames (activation records) for each function call. An environment frame comprises a size field, the return address, where execution continues after the function has been evaluated, the arguments passed to the function, and additional free space for the local variables of the function.

$$
\begin{aligned}
State \in\ & Instr^* \times Adr^* \times EnvFrame^* \times Adr * \times Heap \times ThdState^* \\
& \times SearchContext^* \times Trail \\
Instr =\ & \{\texttt{PushArg}, \texttt{PushInt}, \ldots\} \\
Adr =\ & \mathbb{N} \\
Heap =\ & Adr \rightarrow\!\!\!\!\!\!- \; Node \\
Node =\ & \{\texttt{Int}\} \times \mathbb{N} \;\cup\; \{\texttt{Data}\} \times \mathbb{N} \times Adr^* \;\cup\; \{\texttt{Clos}\} \times Instr^* \times \mathbb{N} \times \mathbb{N} \times Adr^* \\
& \cup \{\texttt{SrchCont0}\} \times ThdState \times ThdState^* \times SearchSpace \\
& \cup \{\texttt{SrchCont1}\} \times ThdState \times ThdState^* \times SearchSpace \times (Adr \cup \texttt{?}) \\
& \cup\; \{\texttt{Susp}\} \times Adr \;\cup\; \{\texttt{Lock}\} \times ThdState^* \;\cup\; \{\texttt{Var}\} \times ThdState^* \\
& \cup\; \{\texttt{Indir}\} \times Adr \\
EnvFrame =\ & \mathbb{N} \times Instr^* \times (Adr \cup \texttt{?})^* \\
ThdState =\ & Instr^* \times Adr^* \times EnvFrame^* \\
& SearchContextAdr \times Instr^* \times Adr^* \times EnvFrame^* \times ThdState^* \times Trail \\
SearchSpace =\ & Adr \times Trail \times Trail \\
Trail =\ & (Adr \times Node)^*
\end{aligned}
$$

Fig. 1. State space

The graph corresponding to the expression that is evaluated, is allocated in the heap H. The register hp serves as an allocation pointer into the heap. We use the notation $H[a/n]$ to denote a variant of the heap H which contains the node n at address a.

$$H[a/x](a') := \begin{cases} x & \text{if } a = a' \\ H(a') & \text{otherwise} \end{cases}$$

The graph is composed of tagged nodes. Integer (`Int`) nodes represent integer numbers. `Data` nodes are used for data terms and comprise a tag, which enumerates the data constructors of each algebraic data type, and a list of arguments. The arity of a data constructor is fixed and always known to the compiler, for that reason it isn't recorded in the node.

Closure (`Clos`) nodes represent functions and function applications. Besides the code pointer they contain the arity of the function, the number of additional

PushArg *n*	SwitchOnTerm *tags&labels*	TryMeElse *label*
PushInt *i*	Jmp *label*	RetryMeElse *label*
PushGlobal *n*	Jmp*Cond label*	TrustMe
PushVar	BindVar	Fail
Pop *n*	Eval	Succeed
PackData *tag*, *n*	Return	Solve
SplitData *m*, *n*	Fork *label*	
SaveLocal *n*	Delay	
Apply *n*	Yield	
Suspend *n*	Stop	

Fig. 2. Instruction set

local variables, and the arguments that have been supplied. The closures returned from the encapsulated search are represented by two kinds of search continuation nodes (`SrchCont0` and `SrchCont1`), which will be described in more detail in the next section.

Unbound logic variables are represented by variable (`Var`) nodes. The wait queue field of these nodes is used to collect those threads, that have been suspended due to an access to the unbound variable.

Suspend (`Susp`) nodes are used for the implementation of lazy evaluation. The argument of a suspend node points to the closure or search continuation, whose evaluation has been delayed. Once the evaluation of the function application begins, the node is overwritten by a `Lock` node, in order to prevent other threads from trying to evaluate the suspended application. Those threads will be collected in the wait queue of the lock. If the evaluation of the function application succeeds the node is overwritten again, this time with an indirection (`Indir`) node, that points to the result of the application. Indirection nodes are also used when a logic variable is bound. The variable node is overwritten in that case, too.

The run queue *rq* maintains the state of those threads, which are runnable, but not active. For each thread the instruction pointer and the thread's data and environment stacks are saved. The search context stack *scs* is used to save the state of the abstract machine when an encapsulated search is invoked. In each search context the pointer to the instruction, where execution continues after the encapsulated search returns, the data and environment stacks, the run queue, and the trail are saved. In addition the address of the goal variable is saved in a search context.

The final register, *tr*, holds a pointer to the trail, which is used to save the old values of nodes that have been overwritten, so that they can be restored upon backtracking or when an encapsulated search is left.

The instruction set of the abstract machine is shown in Fig. 2. Many of these instructions operate similarly to the G-machine [Joh87] and the Babel abstract machine [KLMR96]. They are not described in this paper due to lack of space.

3.2 Encapsulated Search

Representation of search goals A search goal that is reduced by the encapsulated search, is a unary function of result type `Constraint`. The encapsulated search returns if the goal either fails, succeeds or can only proceed non-deterministically. The result is a list of closures, where each closure is of the form `x->`x_1`=:=`e_1`&...&`x_n`=:=`e_n`&`c. Here $x_1, \ldots, x_n$ denote the logic variables that have already been bound to some value (the parameter `x` can be a member of this list) and c represents the yet unsolved part of the search goal. This form is not suitable to be used in the abstract machine, however, because we cannot change the code of the search goal dynamically. A more suitable representation is derived from the fact, that the closure represents the continuation of the search goal at the point, where the non-deterministic computation step takes place. Such a continuation is described by the current contents of the machine registers together with the bindings for the logic variables. A search continuation (`SrchCont1`) node is used to represent this kind of continuation (see Fig. 1).

Local search spaces The different solutions of the search goal may use different bindings for the logic variables and suspended applications contained in the search goal. For instance, in the example given earlier, the local variable `s` is bound to the constant `Zero` in the first alternative and to the data term `Succ t` in the second one.

For efficiency reasons, we do not want to copy the graph corresponding to the search goal for every solution. Instead we share the graph among all solutions and use destructive updates to change the bindings whenever a different search continuation is invoked. Therefore every search continuation is associated with a search space, that contains the list of addresses and values that must be restored, when the search continuation is invoked (the *script* of the search space), and those which must be restored, when the encapsulated search returns (the *trail* of the search space). In addition the search space also contains the address of the logic variable, that was used as an argument to the search goal in order to start its evaluation.

Invocation of search goals A new encapsulated search is started by the `Solve` instruction. This instruction will save the current machine state in a search context on the search context stack *scs*. If the argument passed to `Solve` is a closure node, i.e. the search goal is called for the first time, a fresh, unbound variable is allocated and the search goal is applied to it.

$$\begin{aligned}
&\langle \texttt{Solve} : c, ds_1 : ds, es, hp, H, rq, scs, tr\rangle \Longrightarrow \\
&\langle c', hp : \epsilon, \epsilon, hp + 1, H[hp/(\texttt{Var}, \epsilon)], \epsilon, (hp, c, ds, es, rq, tr) : scs, \epsilon\rangle \\
&\text{where } H[ds_1] = (\texttt{Clos}, c'', ar, l, a_1, \ldots, a_{ar-1}) \\
&\quad \text{and } c' = \texttt{Apply 1} : \texttt{Suspend} : \texttt{Eval} : \texttt{Succeed} : \epsilon
\end{aligned}$$

If instead the argument to the `Solve` instruction is a search continuation, the bindings from its search space have to be restored before the execution of the

goal can continue. No new goal variable needs to be allocated in this case.

$$\begin{array}{l}\langle \texttt{Solve} : c, ds_1 : ds, es, hp, H, rq, scs, tr\rangle \Longrightarrow \\ \langle c', ds', es', hp, restore(H, scr), (g, c, ds, es, rq, tr) : scs, tr'\rangle \\ \text{where } H[ds_1] = (\texttt{SrchCont1}, (c', ds', es'), rq', (g, scr, tr'))\end{array}$$

The auxiliary function *restore* is defined as follows:

$$restore(H, tr) := \begin{cases} H & \text{if } tr = \epsilon \\ restore(H[a/x], tr') & \text{if } tr = (a, x) : tr' \end{cases}$$

Returning from the encapsulated search There are three different cases to consider here. The easy case is when the search goal fails. In that case, the old heap contents and the top-most context from the search context stack are restored and an empty list is returned into that context.

$$\begin{array}{l}\langle \texttt{Fail} : c, ds, es, hp, H, rq, (g, c', ds', es', rq', tr') : scs, tr\rangle \Longrightarrow \\ \langle c', hp : ds', es', hp + 1, restore(H, tr)[hp/(\texttt{Data}, \texttt{[]})], rq', scs, tr'\rangle\end{array}$$

If the search goal succeeds, a singleton list containing the solved search goal must be returned to the context, which invoked the encapsulated search. This is handled by the `Succeed` instruction, that detects this special case from the presence of an empty return context.

$$\begin{array}{l}\langle \texttt{Succeed} : c, ds, \epsilon, hp, H, \epsilon, (g, c', ds', es', rq', tr') : scs, tr\rangle \Longrightarrow \\ \langle c', hp : ds', es', hp + 3, H'', rq', scs, tr'\rangle \\ \text{where } spc = (g, save(H, tr), tr) \\ \qquad H' = restore(H, tr) \\ \qquad H'' = H'[\, hp/(\texttt{Data}, :, hp + 1, hp + 2), \\ \qquad\qquad\quad hp + 1/(\texttt{SrchCont1}, (\texttt{Succeed} : c, \epsilon, \epsilon), \epsilon, spc, \texttt{?}), \\ \qquad\qquad\quad hp + 2/(\texttt{Data}, \texttt{[]})]\end{array}$$

The *save* function saves the bindings of all variables that have been updated destructively. These are those nodes, which have been recorded on the trail:

$$save(H, tr) := \begin{cases} \epsilon & \text{if } tr = \epsilon \\ (a, H[a]) : save(H, tr') & \text{if } tr = (a, n) : tr' \end{cases}$$

In case of a non-deterministic computation step, a list must be returned as well. However, in this case the tail of that list is not empty. Instead it will contain the search continuations for the remaining alternatives. Due to the lazy evaluation semantics of Curry, this tail has to be a suspended application. We use a second kind of search continuation node (`SrchCont0`) for that purpose. These search continuations do not accept an argument, as they just jump to the alternative continuation address encoded in the `TryMeElse` and `RetryMeElse`

instructions.

$\langle$`TryMeElse` alt : $c, ds, es, hp, H, rq, (g, c', ds', es', rq', tr') : scs, tr\rangle \Longrightarrow$
$\langle c', hp : ds', es', hp + 3, H'', rq', scs, tr'\rangle$
where $spc = (g, save(H, tr), tr)$
$H' = restore(H, tr)$
$H'' = H'[\ hp/(\texttt{Data}, (\texttt{:}, hp + 1, hp + 2)),$
$hp + 1/(\texttt{SrchCont1}, (c, ds, es), rq, spc, \texttt{?}),$
$hp + 2/(\texttt{SrchCont0}, (alt, ds, es), rq, spc)]$

The `RetryMeElse` and `TrustMe` instructions are handled similarly.

Unpacking the result In order to access the computed solution for a search goal, the (solved) search goal must be applied to an unbound logic variable. This variable will then be unified with the corresponding binding computed in the search goal (if any). The `unpack` function

```
unpack g | g x = x where x free
```

can be used for that purpose.

In the abstract machine, the `Eval` instruction therefore must also handle suspended applications of search continuations. In that case, the saved search space is merged into the current search space and then execution continues in the solved goal. This will immediately execute the `Succeed` instruction, which unifies the value, that was bound to the goal variable, with the argument applied to the search continuation and then returns into the context where `Eval` was invoked.[2]

$\langle$`Eval` : $c, ds_1 : ds, es, hp, H[ds_1/(\texttt{Susp}, a)], rq, scs, tr\rangle \Longrightarrow$
$\langle c', ds' +\!\!+ ds', (2 : c : a : g') : es', hp + 1, H', rq' +\!\!+ rq, scs, tr' +\!\!+ tr\rangle$
where $H[a] = (\texttt{SrchCont1}, (c', ds', es'), rq', (g, scr, tr'), a')$
$H' = restore(H, scr)[hp/(\texttt{Locked}, \epsilon)]$
$\langle$`Succeed` : $c, ds, es, hp, H, rq, scs, tr\rangle \Longrightarrow$
$\langle c', ds, es, hp, H, rq, scs, tr\rangle$
where $c' =$ `PushArg 1` : `PushArg 0` : `BindVar` : `Return` : ϵ
and $es \neq \epsilon$

4 An Example

In order to demonstrate the operation of the abstract machine, we will consider the function

```
sub m n | add s m=:=n & nat s = s where s free
```

again. The code for this function, together with the code for the functions `nat` and `add` introduced earlier, and the code for the primitive function `&` are shown in Fig. 3.

[2] For the purpose of the presentation, we assume that the search goal is always applied to an unbound variable, so that the `BindVar` instruction is applicable. The machine in fact allows other arguments to be applied as well.

```
     Fn "sub" sub 2 1
sub: PushVar
     SaveLocal 2
     PushArg 2
     PushGlobal "nat"
     Apply
     Suspend
     PushArg 0
     PushArg 1
     PushArg 2
     PushGlobal "add"
     Apply 2
     Suspend
     PushGlobal "=:="
     Apply 2
     Suspend
     PushGlobal "&"
     Apply
     Suspend
     Eval
     Pop 1
     PushArg 2
     Return

     Fn "&" 1 2 0
     Fork l.1
     PushArg 1
     Eval
     Pop 1
     PushArg 0
     Eval
     Return
l.1: PushArg 0
     Eval
     Stop
```

```
       Fn "add" add 2 1
add:   PushArg 0
add.1: SwitchOnTerm [<Susp>:add.2,<Var>:add.3,
                     Zero:add.4,Succ:add.5]
add.2: Eval
       Jump add.1
add.3: Delay
       Jump add.1
add.4: Pop 1
       PushArg 1
       Return
add.5: SplitData 2 1
       PushArg 1
       PackData Succ 1
       PushArg 2
       PushGlobal "add"
       Exec 2


       Fn "nat" nat 1 1
nat:   PushArg 0
nat.1: SwitchOnTerm [<Susp>:nat.2,<Var>:nat.3,
                     Zero:nat.4,Succ:nat.6]
nat.2: Eval
       Jump nat.1
nat.3: TryMeElse nat.5
       PushAtom Zero
       BindVar
nat.4: Pop 1
       PushGlobal "success"
       Exec 0
nat.5: TrustMe
       PushVariables 1
       PackData Succ 1
       BindVar
nat.6: SplitData 1 1
       PushArg 1
       PushGlobal "nat"
       Exec 2
```

Fig. 3. Sample code

The code for the function `sub` constructs a suspended application for the guard expression `add s m=:=n & nat s`. This application is then reduced to weak head normal form with the `Eval` instruction. If the evaluation of the guard succeeds, the result is discarded (it can only be the solved constraint) and the solution, which is the value bound to the logic variable `s`, is returned to the caller.

The code for the primitive function `&`, which implements the concurrent conjunction of two constraints, creates a new thread using the `Fork` instruction as its first action. The new child thread, which becomes active immediately, shares the current environment frame with its parent, in order to access the arguments passed to the function. The child thread reduces the first argument of `&` to weak head normal form and then stops. The parent thread, once it becomes active again, reduces the second argument to weak head normal form and invokes `Eval` for the first argument, too. Because the evaluation of a suspended application replaces the `Susp` node by a `Lock` node, this will block the parent thread until the child thread has completed the evaluation of the first argument.

The function `add` dispatches on the kind of its first argument with the help of the `SwitchOnTerm` instruction. If the argument is a suspend node, it will be reduced to weak head normal form with the `Eval` instruction. Otherwise, if the node is an unbound variable, the `Delay` instruction will suspend the current thread until that variable is instantiated. In both cases the code then redispatches on the result of the evaluation or the instantiated variable, resp. If the node is neither a suspend node nor an unbound variable, then it must be already in weak head normal form and the code jumps directly to the compiled code of the corresponding equation.

The function `nat` similarly dispatches on the kind of its argument. If the argument is a suspend node it will be evaluated and otherwise, if it is not an unbound variable, the abstract machine will jump directly to the code for the matching equation. If the argument is an unbound variable, the function has to instantiate that variable non-deterministically. This is implemented with the help of the `TryMeElse` and `TrustMe` instructions in this example. When the `TryMeElse` instruction is executed, the abstract machine saves the current machine state into two search continuation nodes. The first of them is a `SrchCont1` node whose instruction pointer contains the address of the instruction following the `TryMeElse` instruction, i.e. in our example the instruction `PushAtom Zero`. The second search continuation is a `SrchCont0` node, that shares the machine state with the former search continuation but has a different continuation address. Its instruction pointer contains the address of the instruction at the label of the `TryMeElse` instruction, i.e. in our example the `TrustMe` instruction. Both search continuations are packed into a list node and the abstract machine returns to the top-most context on the search context stack with the the list node on the top of the data stack. The `RetryMeElse` and `TrustMe` instructions work similarly, except that for `TrustMe` the tail of the list node is an empty list instead of a search continuation.

When a `SrchCont1` node is used as an argument to the `try` function, the saved machine state is restored and the execution continues at the addresss following the `TryMeElse` instruction. The correct local bindings for this search continuation are established with the help of the *script* of the `SrchCont1` node. Similarly the `SrchCont0` node restores the saved state and continues at the `TrustMe` instruction.

5 The Implementation

We have implemented a prototype of our abstract machine. Our compiler translates Curry source code into abstract machine code, which is then translated into native machine code using the well-known "C as portable assembler technique" [HCS95,Pey92].

In our implementation we have integrated a few optimizations. Besides using unboxed representations for integer arithmetic like in the G-machine, we were particularly interested in implementing the encapsulated search efficiently. By using destructive updates, we have already minimized the cost for accessing variables and applications. On the other hand we now have an additional overhead on entering and leaving the encapsulated search because the bindings of those nodes have to be updated. However, this update of bindings can be delayed until a search goal is called, that uses a different search space. Because (nearly) all updates affect nodes, that are local to the search space, there is no need to restore those bindings when the encapsulated space is left.[3] If the search goal, that is invoked next, uses the same bindings, as will happen quite often in the case of a depth-first search strategy, then no actions need to be taken and the only overhead, that is caused by the use of the encapsulated search, is due to the creation of the search continuations.

To test the efficiency of our abstract machine, we have run a few benchmarks on it. The results for three of these benchmarks are shown in Fig. 4. The first column lists the execution times for a functional version of the program, the second column shows the same benchmark, but using logical style. The third column shows the results for the benchmarks when translated into Prolog and compiled with Sicstus Prolog. The fourth column contains the execution times for the functional version of the benchmarks compiled with a state-of-the-art Haskell compiler (Glasgow Haskell). All times are given in seconds.

The first benchmark tries to find a solution for the 8-puzzle using a best-first approach. The second benchmark computes the number of solutions for the 8-queens problems. The third benchmark shows the result for naive reverse with a list of 250 elements. The test is repeated 100 times. All benchmarks were run on an otherwise unloaded Ultra Sparc 1 equipped with 128 MBytes of RAM.

[3] Any suspended applications that were used as additional arguments for the search goal have to be restored, however. These can easily be identified, if each suspend node is tagged with the search space, in which it was allocated. In practice such nodes seem to occur rarely.

Benchmark	Functional	Logic	Sicstus	ghc
8-puzzle	1.7	4.1	14.4	0.13
queens	2.9	6.5	3.3	0.92
nrev	13.7	33.2	22.7	1.7

Fig. 4. Runtime results

From the figures one can see that our prototype compares well with a mature Prolog implementation. The functional version of the code is always slightly faster than the corresponding Prolog code, while the logical version is a little bit slower, except for the 8-puzzle. However, our implementation is still much slower than the code generated by the Glasgow Haskell compiler.

6 Related Work

The G-machine [Joh87] implements graph reduction for a functional language. The Babel abstract machine [KLMR96] implements graph reduction for a functional language whose operational semantics is based on narrowing. It incorporates ideas from the G-machine and the WAM [War83], which is the standard abstract machine for implementing Prolog. Our abstract machine extends this further by adding concurrent evaluation and encapsulated search.

The encapsulated search in Curry is a generalization of the search operator of Oz [Smo95]. Their abstract machine [MSS95] differs substantially from ours, because of the different computation model employed by Oz. In particular Oz uses eager evaluation instead of lazy evaluation and therefore lacks the possibility to compute only parts of a search tree. E.g. they could not handle a search goal like `nats` directly.

The implementation of multiple binding environments has been studied in the context of OR-parallel implementations of logic programming languages [GJ93].

7 Conclusion

In this paper we have developed an abstract machine designed for an efficient implementation of Curry. The main contribution of the machine is the integration of encapsulated search into a functional logic language, that employs a lazy reduction strategy. One of the goals of the implementation was to minimize the overhead, that stems from the use of encapsulated search. The prototype, that we have implemented, works reasonably fast compared with the same programs compiled in Prolog. However, we are still at least a magnitude slower than a current state-of-the-art compiler for a functional language. This is due to the fact, that our present compiler does not perform any sensible analysis on the source code. We are currently developing dedicated optimizations in order to include them into the Curry compiler.

References

[ALN87] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, Equations, and Functions. *Proc. ILPS'87*, pp. 17–23, 1987.

[GJ93] G. Gupta and B. Jayaraman. Analysis of Or-Parallel Execution Models. *ACM TOPLAS*, 15(4):659–680, Sept. 1993.

[Han92] M. Hanus. On the Completeness of Residuation. *Proc. JICSLP'92*, pp. 192–206. MIT Press, 1992.

[Han99] M. Hanus. Curry: An integrated functional logic language, (version 0.5). `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.

[HCS95] F. Henderson, Th. Conway, and Z. Somogyi. Compiling Logic Programs to C Using GNU C as a Portable Assembler. *Proc. of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages*, pp. 1–15, 1995.

[HPW92] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell (version 1.2). *SIGPLAN Notices*, 27(5), 1992.

[HS98] M. Hanus and F. Steiner. Controlling Search in Declarative Programs. *Proc. PLILP'98*, pp. 374–390, 1998.

[Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. *Proc. SIGPLAN'84 Symposium on Compiler Construction*, pp. 58–69, 1984.

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers Univ. of Technology, 1987.

[KLMR92] H. Kuchen, R. Loogen, J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-Based Implementation of a Functional Logic Language. *Proc. ALP 1990*, pp. 298–317. Springer LNCS 463, 1992.

[KLMR96] H. Kuchen, R. Loogen, J. Moreno-Navarro, and M. Rodríguez-Artalejo. The Functional Logic Language Babel and its Implementation on a Graph Machine. *New Generation Computing*, 14:391–427, 1996.

[LK99] W. Lux and H. Kuchen. An Abstract Machine for Curry. Technical Report, Univerisity of Münster, 1999.

[MSS95] M. Mehl, R. Scheidhauer, and Ch. Schulte. An Abstract Machine for Oz. *Proc. PLILP'95*, pp. 151–168. Springer, LNCS 982, 1995.

[Pey92] S. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(1):73–80, Jan 1992.

[PW93] S. Peyton Jones and P. Wadler. Imperative Functional Programming. *Proc. 20th POPL'93*, pp. 123–137, 1993.

[Red85] U. Reddy. Narrowing as the Operational Semantics of Functional Languages. *Proc. ILPS'85*, pp. 138–151, 1985.

[Smo95] G. Smolka. The Oz Programming Model. In J. van Leeuwen (ed.), *Current Trends in Computer Science*. Springer LNCS 1000, 1995.

[SSW94] Ch. Schulte, G. Smolka, and J. Würtz. Encapsulated Search and Constraint Programming in Oz. *Proc. of the Second Workshop on Principles and Practice of Constraint Programming*. Springer, 1994.

[War83] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.