An Efficient Abstract Machine for Curry

Wolfgang Lux, Herbert Kuchen

Universität Münster
{wlux,kuchen}@uni-muenster.de

Abstract. The functional logic programming language Curry integrates features from functional, logic, and concurrent programming. It combines lazy evaluation, higher-order functions, logic variables, partial data structures, built-in search and the concurrent evaluation of (equational) constraints. Curry amalgamates the most important operational principles developed in the area of integrated functional logic languages: residuation and narrowing. The effects of non-determinism stemming from the narrowing semantics can be encapsulated using the primitive `try` operator. This is necessary to implement declarative I/O in the presence of non-deterministic computations and allows to implement different search strategies.
In the present paper we develop an abstract machine for the implementation of Curry. The focus in this paper will be on the efficient implementation of encapsulated search in a lazy functional logic language.

1 Introduction

The integrated functional logic language Curry[HKM95,Han99] combines features from functional languages, logic languages, and concurrent programming. Using basically the same syntax as the functional language Haskell [HPW92], Curry also implements many of the features of Haskell. It provides higher order functions, nested expressions, lazy evaluation of functions, monadic I/O [PW93] and a polymorphic type system with Hindley-Milner style type inference [Mil78]. Curry extends Haskell by supporting logic variables and allows computations with partial data structures as in logic languages like Prolog [SS86]. Curry also supports the concurrent evaluation of expressions and uses logic variables as means of synchronization.

The two most important operational principles developed in the area of functional logic programming are narrowing and residuation (see [Han94] for a survey on functional logic programming). Narrowing [Red85] combines unification and reduction, allowing the non-deterministic instantiation of logic variables in expressions. The residuation strategy [ALN87] on the other hand delays the evaluation of functions until their arguments have been sufficiently instantiated. Curry smoothly integrates both operational principles and gives the programmer the chance to select the appropriate strategy for every function based on his/her needs.

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. Also global non-determinism conflicts with the single-threaded interaction with the world imposed by the monadic I/O concept. Therefore Curry requires all I/O operations to occur only in the deterministic parts of the program. As a remedy to both problems, Curry offers a primitive operator `try`, that encapsulates the non-deterministic parts of the program and allows the user to define his/her own search strategies.

In this paper we develop an abstract machine for the efficient implementation of Curry. Due to the lack of space we cannot describe the machine in full detail in this paper and we restrict the presentation to the implementation of the encapsulated search, which is the main contribution of our work. In the rest of the paper we will assume some familiarity with graph reduction machines for functional and functional logic languages [Joh87,Loo93].

The rest of this paper is organized as follows. The next section gives an introduction to the computation model of Curry. Section 3 describes the encapsulated search operator `try` in more detail. Section 4 introduces the abstract machine. Section 5 then presents some runtime results for our prototypical implementation. The sixth section presents related work and the final section concludes.

2 The Computation Model of Curry

The basic computational domain of Curry is a set of data terms. A data term t is either a variable x or the application $c\,t_1 \ldots t_n$ of an n-ary data constructor c to n argument terms. 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 an application $e_1\,e_2$ of two expressions.

Constraint expressions are checked for satisfiability. The predefined nullary function `success` reduces to a constraint that is trivially 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 the definition to provide additional local definitions, whose scope is the guard expression and the expression e. Unbound logic variables can be introduced by the special syntax[1] `where` $x_1, \ldots, x_n$ `free`

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

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 = Succ (add m n)
```

An expression is reduced to a value and a set of bindings for the free variables in the expression (the answer). Due to non-determinism, there may be several such pairs of a result value and an answer substitution. Thus the computational domain consists of disjunctions of such pairs. For instance, if the function `f` is defined by the two equations `f 0 = 1` and `f 1 = 2`, the solution to the goal `f x`, where `x` is a free variable, is the disjunction[2] `{x=0} 1 | {x=1} 2`.

A single computation step performs a reduction in the outer-most redex of exactly one disjunct. How this disjunct is chosen is left to the implementation. The reduction may either yield a single new expression (deterministic step), or a disjunction of several new expressions together with their corresponding bindings (non-deterministic step), or may fail, depending on the number of applicable equations.

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

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 disjunction, comprising a disjunct for each possible binding of the variable. In the example, the reduction would yield the disjunction `{n=Zero} success | {n=Succ m} nat m`, 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 strategy, 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.

3 Encapsulated Search

The use of monadic I/O, which enforces a single-threaded interaction with the world, conflicts with the non-deterministic instantiation of unbound variables. For that reason Curry provides the primitive search operator `try`, that allows

[2] Here `|` denotes a disjunction and `{x=0}` denotes a substitution. `{x=0} 1` denotes a pair of an expression and a substitution.

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 `try` operator expects a search goal as argument, which must be a unary function that returns a constraint. 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 a least two elements, if the goal can only be reduced 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 nat s & add s Zero=:=Succ Zero)
```

yields the list

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

4 The Abstract Machine

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 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 a 9-tuple $\langle c, ds, es, hp, H, rq, bs, scs, tr\rangle$, 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.

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

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

Fig. 1. State space

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, therefore it is not 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 local variables, and the arguments that have been supplied.

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. Finally two kinds of search continuation nodes are used to implement (partially) solved search goals. They will be described in more detail below.

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 backtrack stack *bs* manages the choicepoints, that are created to handle global non-determinism by a backtracking strategy. In each choicepoint the alternate continuation address, the data and environment stacks, the run queue, the trail, and the allocation pointer are saved. The search context stack *scs* manages search contexts, which are used during an encapsulated search. The saved machine state in a search context is similar to a choicepoint, except that the instruction pointer denotes the continuation into which the `try` operator returns, and that instead of the allocation pointer a

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

Fig. 2. Instruction set

pointer to a goal variable is saved. The final register, tr, holds a pointer to the trail, which is used to save the values of nodes that are 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 the instructions operate similarly to the G-machine [Joh87] and the Babel abstract machine [KLMR96] and are not described in this paper due to the lack of space.

Representation of search goals The encapsulated search operator `try` returns a list of closures 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 and c represents the yet unsolved part of the search goal. These closures are represented by search continuation (`SrchCont1`) nodes, that capture the current state of the machine registers together with the bindings for the logic variables.

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 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*), and those which must be restored, when the encapsulated search returns (the *trail*). In addition the search space also contains the address of a goal variable.

Invocation of search goals A new encapsulated search is started by the `Solve` instruction. This instruction saves the current machine state in a search context on the search context stack. If the argument passed to `Solve` is a closure node, i.e. the search goal is invoked for the first time, a fresh, unbound variable is allocated and the goal is applied to it. 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. Also no new goal variable needs to be allocated in this case.

$$\langle \texttt{Solve} : c, ds_1 : ds, es, hp, H, rq, bs, scs, tr\rangle \Longrightarrow$$
$$\langle c', hp : \epsilon, \epsilon, hp+1, H[hp/(\texttt{Var}, \epsilon)], \epsilon, bs, (hp, c, ds, es, rq, tr) : scs, \epsilon\rangle$$
$$\text{where } H[ds_1] = (\texttt{Clos}, c'', ar, l, a_1, \ldots, a_{ar-1})$$
$$\quad \text{and } c' = \texttt{Apply 1} : \texttt{Suspend} : \texttt{Eval} : \texttt{Succeed} : \epsilon$$
$$\langle \texttt{Solve} : c, ds_1 : ds, es, hp, H, rq, bs, scs, tr\rangle \Longrightarrow$$
$$\langle c', ds', es', hp, restore(H, scr), rq', bs, (g, c, ds, es, rq, tr) : scs, tr'\rangle$$
$$\text{where } H[ds_1] = (\texttt{SrchCont1}, (c', ds', es'), rq', (g, scr, tr'), \texttt{?})$$

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.

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

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.

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

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 non-empty list must be returned. It will contain the search continuations for the alternative computations. Due to the lazy evaluation semantics of Curry, the tail of this list has to be a suspended application. We use a second kind of search continuation nodes (`SrchCont0`), that do not take arguments, for that purpose.[3]

[3] The `RetryMeElse` and `TrustMe` instructions are handled similarly.

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

Unpacking the result In order to access the computed solution for a solved search goal or to constrain a solution further, the search goal must be applied to an argument. The current binding for the goal variable computed in the search goal is then unified with the argument. The function `unpack` can be used to extract the solution from a solved goal:

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

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 goal. If the search goal is already solved, it will immediately execute the `Succeed` instruction, which unifies the argument and the value, that was bound to the goal variable and then returns into the context where `Eval` was invoked.[4]

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

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" [Pey92,HCS95].

In to order test the efficiency of our abstract machine, we have run some well known benchmarks. *Tak* computes the Takeuchi function, *nrev* performs 100 times a naive reversal of a list of 250 elements, and *8-puzzle* tries to find a solution to the 8-puzzle. We have compared two versions of the program, one written in the purely functional Haskell subset of Curry, the other using logical style,

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

which employs narrowing and the encapsulated search. For a comparison we also include the results for Sicstus Prolog and Glasgow Haskell. All benchmarks were run on an otherwise unloaded Ultra Sparc 1 equipped with 128 MBytes of RAM. The execution times are given in seconds.

Benchmark	Functional	Logical	Sicstus	ghc
tak	14.6	37.9	14.2	0.22
nrev	13.7	33.2	22.7	1.7
8-puzzle	1.7	4.1	14.4	0.13

Fig. 3. 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 narrowing machines developed for the functional logic language Babel [KLMR96] were a basis for the design of our abstract machine. These abstract machines in turn incorporate ideas from the G-machine [Joh87] and the WAM [War83].

Encapsulated search was first introduced in 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.

Other implementations of local search spaces are known [Gup94]. E.g. the Java based abstract machine in [HS97] uses binding tables for each variable, which are indexed by a unique identifier assigned to each search space.

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 and 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. In *Proc. 4th ILPS*, pages 17–23, 1987.

[Gup94] G. Gupta. *Multiprocessor Execution of Logic Programs.* Kluwer, 1994.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[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. In *Proc. of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages*, pages 1–15, 1995.

[HKM95] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

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

[HS97] M. Hanus and R. Sadre. A concurrent implementation of Curry in Java. In *Proc. ILPS'97 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1997.

[HS98] M. Hanus and F. Steiner. Controlling search in functional logic programs. In *Proc. PLILP'98*, 1998.

[Joh87] T. Johnsson. *Compiling lazy functional languages.* PhD thesis, Chalmers Univ. of Technology, 1987.

[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.

[Loo93] R. Loogen. Relating the implementation techniques of functional and functional logic languages. *New Generation Computing*, 11:179–215, 1993.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer System Science*, 17(3), 1978.

[MSS95] M. Mehl, R. Scheidhauer, and Ch. Schulte. An abstract machine for Oz. In *Proc. PLILP'95*, pages 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. In *Proc. 20th POPL'93*, pages 123–137, 1993.

[Red85] U. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. ILPS'85*, pages 138–151, 1985.

[Smo95] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Current Trends in Computer Science.* Springer LNCS 1000, 1995.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[War83] D. Warren. An abstract Prolog instruction set. TR 309, SRI, 1983.