Using Demandedness Information for the Compilation of Lazy Narrowing [§]

(Extended Abstract)

Juan José Moreno-Navarro[*] Herbert Kuchen[†] Julio Mariño-Carballo[*]

Stephan Winkler[†] Werner Hans[†]

Motivation and Preliminars

Functional logic programming languages have a functional syntax and use narrowing as operational semantics. Here we consider the efficient implementation of lazy narrowing, a strategy which only evaluates the arguments of a function application, if their evaluation is really demanded. For an efficient implementation of lazy narrowing it is crucial to evaluate the arguments as early as possible. Otherwise the arguments are frequently reevaluated. A demandedness analysis is used to detect which parts of the arguments can safely be evaluated before the call to the function. Several approaches (e.g. [HLW92, JMM92]) also use this idea, but they sacrify laziness in order to avoid inefficiency. Our approach is more lazy than the previous approaches, and it uses a more powerful notion of demandedness, which allows to express infinite demand patterns like e.g. spine normal form. Moreover, in contrast to the previous approaches, we take into account dependencies between the arguments of a function in order to improve the efficiency.

We will use the language BABEL [MR92] along the paper, but it is clear that the techniques are independent of the chosen functional logic language. The language BABEL integrates functional and logic programming in a flexible and mathematically well-founded way. It is based on a constructor discipline, has a polymorphic type system, and uses narrowing as evaluation mechanism [Re85]. Rather than giving a precise description of the syntax and semantics of BABEL we will show a small example program.

datatype	nat := 0 | s nat
fun	plus: nat → nat → nat.
	plus 0 M := M.
	plus (s N) M := plus N (s M).

solve plus (s X) (s Y).

A possible result for this goal is $(s\ (s\ Y))$ provided that X is bound to 0.

The use of narrowing as the operational semantics of functional logic languages was first proposed by Reddy [Re85]. In the present paper, we consider a lazy narrowing strategy [MKLR90]. This strategy narrows the leftmost outermost redex, unless an inner redex needs to be narrowed first in order to decide whether the currently considered rule (following the textual order of rules) can be applied or not.

[§]This work is partially supported by the spanish PRONTIC project TIC/93-0737-C02-02

[*]Universidad Politécnica de Madrid, Departamento LSIIS, Facultad de Informática, Campus de Montegancedo, Boadilla del Monte, 28060 Madrid, Spain, email:{jjmoreno, jmarino}@fi.upm.es, fax: +34 1 336 74 12

[†]RWTH Aachen, Lehrstuhl für Informatik II, Ahornstr. 55, W-5100 Aachen, Germany, {herbert, winkler,hans}@zeus.informatik.rwth-aachen.de

Demandedness Analysis

The notion of strictness is used in lazy functional languages in order to decide which arguments are needed anyway and may hence be evaluated in advance, avoiding the construction of some intermediate representation of unevaluated expressions. In purely functional programming, this is valid since the whole computation fails, if some subexpression fails. In functional logic programming, a failing subexpression causes backtracking, and some alternative computation will be tried. Hence, the notion "strictness" is replaced by the notion "demandedness". In fact, only one (part of an) argument can be demanded at a time. If the evaluation of this argument fails, all remaining arguments are not evaluated. If, however, the evaluation of the argument succeeds and the values of the currently evaluated arguments are unifiable with the corresponding patterns on the left hand side of a rule, then the next argument may be demanded and can be evaluated. Hence, this stepwise notion of demandedness allows several arguments to be evaluated in advance, and it takes into account the patterns on the left hand side and especially their dependencies.

The inefficiency due to frequent reevaluations of the arguments can be avoided, if the arguments are evaluated before to function call, more precisely: before a choice point for the function is created. Here, the notion of demandedness is needed in order to decide which arguments may be evaluated in advance without sacrificing laziness. In fact, this idea was already followed in several approaches [MKLR90, HLW92, JMM92]. Note that this preliminary evaluation of the arguments is much more important for functional logic languages, where reevaluations can be avoided, than for purely functional languages, where just the construction of an intermediate representation of unevaluated expressions is saved. However, the mentioned approaches have the drawback that they do not take into account dependencies between the arguments, but consider each argument separately. This leads to a blind evaluation of arguments even if it is already clear that no rule will be applicable, since no left hand side of a rule is unifiable with the values of the arguments computed up to now.

In order to be able to compute the most accurate evaluation degree without loss of laziness we formalize the notion of degree of evaluation in the so called *demand patterns* and then we will sketch an abstract interpreter to compute this degree.

Intuitively, a pattern tree can be used to indicate how much an argument needs to be evaluated. An or-pattern enumerates different alternatives. **any** means that no evaluation is needed. **hnf** asks for evaluation to *head normal form*, i.e. evaluation to an expression which is either a free variable or has a constructor on the top. **nf** (for *normal form*) demands evaluation to a term. These last two components are introduced to simplify the patterns. For instance, the pattern $or(c_1\ (pt_1), \ldots, c_m\ (pt_m))$ could be simplified to $or(\mathbf{hnf})$ if each pt_i's is of the form (**any**, ..., **any**) and $c_1, \ldots, c_m$ are all the constructors of the type. Labels are introduced to allow regular trees to represent infinite shapes (as spine normal form for lists). In case of finite patterns we will skip the labels. Pattern trees are able to represent the demandedness information of *distributive functions* A distributive function is a function with only one distributive demand argument. An argument is demanded if there is a rule with a non variable formal parameter for it. We say that an argument of f is distributive if their corresponding different formal parameters of the rules for f differ only in a unique position.

Example: Consider the following function:

```
fun  length: list T → nat.
     length [] := 0.
     length [H|T] := 1 + length T.
```

Apart from the fact that we have not explained how we have got them, the reader can intuitively accept that the following pattern trees are correct: or([],[**any**|**any**]), or([],[**any**|or([], [**any**|**any**])]), or([],[**any**|or([],[**any**|or([],[**any**|**any**])])]) ... The most general (and safe) pattern tree is a regular one: l1 : $or([], [\mathbf{any}|l1])$

Consider now the *plus* function used in section 2. A correct pattern tree for the first argument of *plus* is $l1 \; : \; or \; (0, \; s \; (l1))$ which could be simplified to or (**nf**) or, by abuse of notation, to **nf**.

Pattern trees for a given function can be ordered into a lattice $\mathcal{L}$(CS). The bottom of the lattice is $\perp$ (to represent inconsistent patterns, although **nf** is the least useful information we can get), while the top is **any**. We will use *lub* for the least upper bound and *glb* for the greatest lower bound in the lattice. The ordering $\preceq$ induces an equivalence relation into $\mathcal{L}$, which is the basis for a canonical representation of demand pattern.

The abstract interpreter abstracts a program by a set of equations. The equations reflect the demandedness information that the program rules contain. Variables represent the demandedness information required for the arguments (an also for any argument position and rule). The operation "+", represents the *lub* operation of the lattice. Intuitively, the solution for a variable is the *glb* of all the pattern trees that fulfil the equations for such a variable.

The resulting equation system for the *plus* example is:

$$\begin{array}{lcll} A & = & A^1 + A^2 & \text{General equation mixing both rules} \\ A^1 & = & 0 & \text{Equation from first rule} \\ A^2 & = & s\ A^2_1 & \text{Equation from second rule} \\ A^2_1 & = & A & \text{Equation from the body of the second rule} \end{array}$$

Equations are solved by applying two algorithms: 1) unification between pattern trees, and 2) *lub* of pattern trees. These algorithms behave similar to those for regular trees, see [Co84, Co83]. The equation solver proceeds bottom-up from simple equations to more complicate ones applying these algorithms. In summary, the solution for this equation system is the following:

$$\begin{array}{lcl} A & = & or(0,\ s\ A) \\ A^1 & = & 0 \end{array} \qquad \text{or} \quad \begin{array}{lcl} (A & = & l1 : or(0,\ s\ l1) \\ A^2 & = & l2 : or(0,\ s\ l2) \end{array}$$

The abstract interpreter sketched above is safe. Informally, this means that every lazy computation of f will evaluate the arguments at least as the pattern tree inferred by the abstract interpreter. More formally:

Definition: A pattern tree p is safe for the i-th argument of a function f if for every well typed expressions $e_1, \ldots, e_n$ and every narrowing sequence $f e_1 \ldots e_i \ldots e_n \stackrel{*}{\Longrightarrow}_{\sigma_{out}} e$, e in head normal form, then e_i is narrowed inside the narrowing sequence to a expression lesser than p.

We prove the following theorem:

Theorem: If the abstract interpreter infers a pattern tree p for the i-th argument of a function f then p is safe.

Now, we need to extend the notion of demandedness for non distributive functions. Notice that the extension is not trivial. One cannot simply consider a tuple of pattern trees because we loose the safety property. For instance, consider the next simple function:

```
f 0 1 := 1.
f 1 0 := 0.
```

If we apply the previous method we get the following pair of pattern trees $(or(0, 1),\ or(1, 0))$, but the pair (0,0) has the shape of this and it is not correct. The problem is that ordinary tuples are not distributive. Instead of having only a kind of tuples of pattern trees we will have another possibility. We will treat some tuples as a special constructor *tup* with an overloading property. Different tuples into an or-pattern are treated as different constructors and no factorization is done. New tuples are distributive. We will call the resulting pattern trees *demand patterns*. In the previous example, the correct demand pattern is: $(or(tup(0,1), tup(1,0))$, and the pair (0, 0) has not its shape.

Consider the following non distributive modification of the *plus* function of section 2.

```
fun  plus : nat → nat → nat → nat.
     plus 0 Y 0 := Y.
     plus 0 Y 1 := Y.
     plus (s X) Y 0 := (plus X (s Y) 1).
     plus (s X) Y 1 := (plus X (s Y) 0).
```

The first and the third arguments are not distributive. The solution trick is to consider a specialization of the function depending on a non distributive argument. In our case, we select the third one. Each function is replaced by several different functions, one for each all the possible cases of the selected argument. In the example, we generate functions $plus_0$ and $plus_1$. If one rule has a variable as formal parameter all the possible specializations are done for this rule.

The new functions for the *plus* function are:

```
plus₀ 0 Y := Y.                     plus₁ 0 Y := Y.
plus₀ (s X) Y := (plus₁ X (s Y)).   plus₁ (s X) Y := (plus₀ X (s Y) 0).
```

Now, the equations (for the first argument only) are the following, where $A0$, $A1$ are the variable names for $plus_0$ and $plus_1$ respectively.

$$\begin{array}{lllllllllll} A0 & = & A0^1 + A0^2 & A0^1 & = & 0 & A1 & = & A1^1 + A1^2 & A1^1 & = & 0 \\ A0^2 & = & sA0^2_1 & A0^2_1 & = & A1 & A1^2 & = & sA1^2_1 & A1^2_1 & = & A0 \end{array}$$

which gives the following results: $A0 = l1 : or(0, s(l1)) \quad A1 = l2 : or(0, s(l2))$

The solution is now mixed with the specialization to get tuples and the final demand pattern. It consists in an or-pattern with distributive tuples as component. In the example we get the most accurate demand pattern as possible: $or(tup(\mathbf{nf}, any, 0), tup(\mathbf{nf}, any, 1))$ that is equivalent to: $(or(tup(\mathbf{nf}, any, or(0, 1))$.

The safety theorem can be extended to demand patterns.

Theorem: If the transformation and the abstract interpreter infers a demand pattern p for a function f then p is safe.

A Translation Scheme and Experimental Results

Next, we want to describe how we translate BABEL programs into abstract machine code. For simplicity of presentation, we will assume that the machine has a data stack for auxiliary computations.

This leads to the following code for a BABEL function. Let p_i be the inferred demand pattern for the arguments of the *ith* rule:

$ca(f)$:	$demandtrans(p_1)$
	JMP r_1
l_2:	UNDO
	$demandtrans(p_2)$
	JMP r_2
	⋮
l_m:	UNDO
	$demandtrans(p_m)$
	JMP r_m
l_{m+1}:	UNDO
	FAIL
r_1:	$ruletrans(\mathrm{lhs}_1, \mathrm{rhs}_1, l_2)$
r_2:	$ruletrans(\mathrm{lhs}_2, \mathrm{rhs}_2, l_3)$
	⋮
r_m:	$ruletrans(\mathrm{lhs}_m, \mathrm{rhs}_m, l_{m+1})$

First, there is a part, the *selection section* (from ($ca(f)$ to FAIL), which tries to find an applicable rule by checking whether the left hand side of some rule is unifiable with the arguments of the considered function call. If this check requires the evaluation of parts of some arguments, this evaluation is performed. The arguments are evaluated from left to right if they are demanded. However, during the evaluation they are checked to have the shape of the corresponding demand pattern, so dependencies between the arguments are taken into account. If rule i is found to be applicable, a jump to the main code for the rule, generated by *ruletrans*(lhs_i,rhs_i,l_{i+1}), is performed. We will not give details about the compilation of rules.

The selection section uses the *demandtrans* scheme in order to check whether the considered part p of an argument has the same shape as the demand pattern.

In the following, we will represent a demand pattern by a graph g and a pointer l to the currently considered node of this graph. Since the graphs possibly contain cycles, we will represent them by a mapping $g : Addresses \rightarrow Nodes$.

Of course, general regular patterns requires some kind of recursion. We will manage them by using new function calls, i.e. the control stack is used to simulate recursive patterns.

The *demandtrans* scheme has to make some case analysis on the type of the currently considered node of the demand pattern.

If the current node is an or-node, we first evaluate (by EVAL) the considered parts of the argument to **hnf**, if necessary. Then we use a SWITCH instruction to jump depending on the topmost constructor c of the considered part of an argument. If $c = c_i$ (for some $i \in \{1, \ldots, k\}$, then SWITCH first loads the arguments of c to the data stack and jumps then to label l_i, where the code starts which handles the sub-patterns starting with constructor c. The code is generated by a recursive call $demandtrans(g[p/\mathrm{label}\ l], p_i)$. Note that in this recursive call, we replace the currently considered node $g(p)$ by a new *label node* containing the address l of the code which handles the currently considered or-pattern.

If the topmost constructor c of the considered part of the argument is not contained in $\{c_1, \ldots, c_k\}$, this indicates that c conflicts with the topmost constructors demanded by the right hand side of the rule. The computation will fail in this case and SWITCH directly backtracks to the last alternative computation.

If the considered part of the argument does not start with a constructor but is just an unbound variable, no further evaluation of the argument can be done and SWITCH just behaves as a JUMP.

If the currently considered node is a label node, this indicates that we have detected a cycle in the pattern graph and that we have already produced code for this node.

If the considered demand pattern is *any*, then we just remove the considered argument from the data stack.

Let us now discuss the case that the pattern is an and-pattern with a constructor application. A previous SWITCH instruction has checked that the expression starts with the correct topmost constructor. So, the arguments are loaded into the data stack and the sub-patterns are handled by recursive calls of the *demandtrans* scheme.

Distributive as well as non distributive tuples are handled by simply making recursive calls to the *demandtrans* scheme.

Finally, the **hnf** pattern is handled by an EVAL instruction.

We have investigated the following example programs: 1) the computation of all the sublists of revers $[1, \ldots, n]$, and 2) the n-queens problem (using a simple generate and test approach). Both examples have the property that a lot of reevaluations are needed of demanded arguments are not evaluated in advance. The example programs were executed on a stack-based narrowing machine [Lo91]. Based on the implementation of the presented ideas on the stack-based narrowing machines, we have tried some example programs and measured their runtimes with the naive lazy approach and with our new approach. Additionally, we have measured the runtimes for an eager implementation of BABEL. The runtimes are depicted in Table 1. The examples show that the runtimes can be considerably improved, if

example	n	Eager	Naive Lazy	New Approach
sublists	10	0.470	1.000	0.620
	11	0.910	1.980	1.240
	12	1.860	3.960	2.420
	13	3.680	7.940	4.800
	14	7.420	15.820	9.600
	15	15.050	31.750	19.230
n queens	4	1060	890	210
	5	14500	10550	2600
	6	247030	174300	39300

Table 1: Runtimes of the example programs in seconds.

the demanded arguments are evaluated in advance. Notice that in examples like n-queens, the lazy strategy is even better that the eager one. Moreover, bigger real examples have a lot of nested function call, what means a considerable risk of reevaluation.

References

[Co84] A. Colmerauer, Equations and Inequations on Finite and Infinite Trees, Proc. International Conference on Fifth Generation Computer Systems, ICOT 1984, pp. 85-99.

[Co83] B. Courcelle, Fundamental Properties of Infinite Trees, Theoretical Computer Science 25, 1983, pp. 95-169.

[HLW92] W. Hans, R. Loogen, S. Winkler: On the Interaction of Lazy Evaluation and Backtracking, Procs. PLILP'92, LNCS 631, 1992, 355-369.

[Hu76] G. Huet, Resolution d'équations dans les languages d'ordre 1, 2, $\ldots, \omega$, Thèse de doctorat d'etat, Université Paris VI.

[JMM92] J.A. Jiménez-Martín, J. Mariño-Carballo, J.J. Moreno-Navarro: Some Techniques for the Efficient Compilation of Lazy Narrowing into Prolog, Procs. LOPSTR'92, Manchester, 1992.

[Lo91] R. Loogen: From Reduction Machines to Narrowing Machines, TAPSOFT'91, LNCS 494, 1991, 438-457.

[MKLR90] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: Lazy Narrowing in a Graph Machine, Procs. ALP'90, LNCS 463, 1990, 298-317.

[MM93] J. Mariño-Carballo, J.J. Moreno-Navarro, Abstract Interpretation for Demandedness Analysis, Technical Report, Facultad de Informatica de Madrid, in preparation.

[MR92] J.J.Moreno-Navarro, M. Rodríguez-Artalejo: Logic Programming with Functions and Predicates: The Language BABEL, J. Logic Programming, 12, 189-223 (1992).

[Re85] U.S. Reddy: Narrowing as the Operational Semantics of Functional Languages, Procs. Int. Symp. on Logic Programming, IEEE Comp. Soc. Press 1985, 138-151.