Recovering Sequentiality in Functional-Logic Programs

Julio Mariño Juan José Moreno-Navarro
Universidad Politécnica de Madrid

Abstract

Efficient code generation in implementations of functional logic languages relies on the sequentiality of the program rules — existence of an optimal evaluation order for arguments. Parallel evaluation of arguments in the presence of free variables is out of the question due to the possibility of backtracking and sharing of these variables among different arguments. In this paper we show that lack of sequentiality is often syntactic rather than semantic and that a clever use of type information and strictness analysis can enable a compiler to generate sequential code from most programs.
Keywords: Sequentiality, Abstract Interpretation, Functional Logic Programming.

1 Introduction

The relationship between *sequential term rewriting systems* [8] and the implementation of lazy functional programming languages is well known — see, for instance [16, 15] for an introduction to the subject. The typical example is *parallel or*, defined by the equations

```
or true  x     = true
or x     true  = true
or false false = false
```

One would expect any system with lazy evaluation to give the answer `true` to the calls

```
or true (loop e)
or (loop e) true
```

where `(loop e)` is any non-terminating expression. But in practice what we get is a hung terminal — at least for one of the calls. The problem is that given a call `(or` e_1 e_2`)` there is no information on which argument has to be reduced first and the compiler favors one of them.

In other cases, there is a "good" order to try the evaluation of arguments, but actual implementations are unable to perform the sequentiality analysis required to generate sequential code. For instance, certain "lazy" Hope interpreter is bound to the left-to-right order and thus the program

LSIIS - Facultad de Informática. Campus de Montegancedo s/n, 28660, Madrid, SPAIN. email: {jmarino|jjmoreno}@fi.upm.es, voice: +34-91-336-7458, fax: +34-91-336-7412. This research was partly supported by the Spanish project TIC96.1012-C02-02.

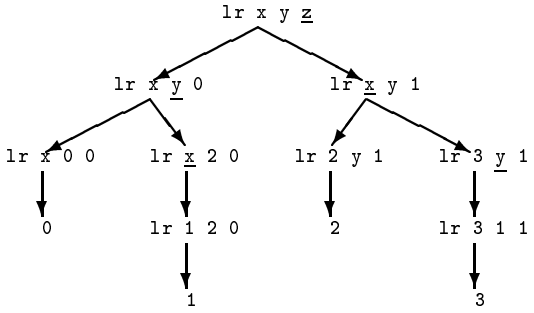
```
dec left: num -> num -> num ;
--- left x 0 <= 0 ;
--- left 1 1 <= 1 ;

dec right: num -> num -> num ;
--- right 0 x <= 0 ;
--- right 1 1 <= 1 ;

dec loop: alpha -> beta ;
--- loop x <= loop x ;
```

loops with `(left (loop 0) 0)` but `(right 0 (loop 0))` gives the desired `0`. This problem is solved in a widely used Haskell interpreter statically reordering the arguments but some sequential programs still cause trouble:

```
lr x 0 0 = 0
lr 1 2 0 = 1
lr 2 y 1 = 2
lr 3 1 1 = 3
```

The call `(lr (loop 0) 0 0)` works fine, but `(lr 2 (loop 0) 1)` loops.

Our own Curry [7] compiler (Sloth [17]) is able to generate sequential code from these two examples by performing sequentiality analysis based on the so called *definitional trees* [1]. For the last example the following decision tree is built:



Depending on the value of the third argument to `lr`, the first or the second argument is evaluated. For a given set of rules there may exist several definitional trees [1]. The class of programs for which a definitional tree exists is that of *inductively sequential programs*[2].

For those cases (as parallel or) where no definitional tree exists, *parallel definitional trees* have been proposed [3]. These imply simultaneous reduction of arguments and, being Curry a functional logic language [6, 12], those arguments can have free variables in them and even share those variables. This leads to a risk of reevaluation and speculative work even more serious than in functional systems. It does not mean that the implementation is not possible (see [5]), but the complexity and inefficiency is very high. For more arguments in favor of sequentialization, see [11].

So it seems that parallel evaluation should be avoided as much as possible and that means that a compiler should generate parallel trees only as a last resort or even allow the user to choose a non-complete evaluation, issuing the opportune warnings at compile time. Fortunately, most functional programs are inductively sequential (see [4, 18] for a large sample) and some of those that are not, can be rewritten into an equivalent sequential one. A typical example is merging two lists of numbers:

[1] Sloth is actually a translator from Curry into Prolog written itself in Prolog and is able to obtain every possible definitional tree via backtracking. Additional criteria to choose among them exist but are beyond the scope of this paper.

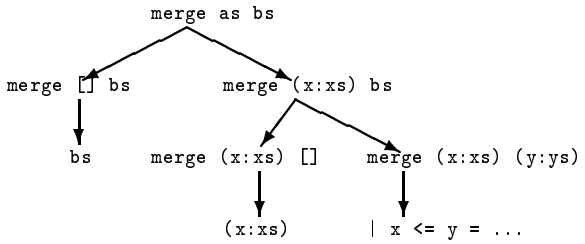
```
merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) | x <= y = x:(merge xs (y:ys))
                    | x >= y = y:(merge (x:xs) ys)
```

If we just look at the left hand sides it looks like a parallel or in disguise, but if we look at the right hand sides we immediately see that both arguments to `merge` are demanded. A programmer could, in fact, have written

```
merge []     ys     = ys
merge (x:xs) []     = x:xs
merge (x:xs) (y:ys) | x <= y = x:(merge xs (y:ys))
                    | x >= y = y:(merge (x:xs) ys)
```

instead. This more or less corresponds with the classic notions of *strong* and *weak* sequentiality [2] [15].

The thesis of this work is that for the vast majority of programs, the information obtained from type inference and strictness analysis can help a compiler to generate sequential definitional trees for programs even when they are not syntactically (strongly) sequential. For the `merge` rules, the following tree can be produced:



Typing information is important here for two reasons. On one hand, if the most general type of a given argument fixes a type constructor, we know all the possible data constructors that can appear at that position of the tree. On the other hand, if the argument is polymorphic the information is that the argument is not "touched" by pattern matching during reduction.

Finally, an example showing that higher order strictness analysis is desirable. Figure 1 shows a higher order generalization of the `merge` function. The (functional) parameters indicate the task to be done when the selected element comes from the first list, the second, or both, and what to do when one of the lists is empty. This function can be used to define list union, intersection and difference. Although `hiMerge` is not sequential, it is reasonable to generate code for its two possible sequential versions — let them be called $hiMerge_1$ and $hiMerge_2$. Then `union` can be executed calling either $hiMerge_1$ or $hiMerge_2$ directly, `difference` can call $hiMerge_1$ and only `intersection` would need to call the parallel evaluation code.

The paper is organized as follows. Section 2 introduces essential notions of narrowing and definitional trees. Section 3 defines a denotational semantics for a purely equational subset of Curry. The framework for abstract interpretation used in this work is presented in section 4 and applied in section 5 to yield a strictness analysis for our language. Code generation based on the information from the analyzer is discussed in section 6. Section 7 concludes.

[2] We are using the fact that for constructor systems *left*- and strong sequentiality are equivalent [20].

```
hiMerge :: (Int->[Int]->[Int]) -> (Int->[Int]->[Int]) -> (Int->[Int]->[Int]) ->
  ([Int]->[Int]) -> ([Int]->[Int]) -> [Int] -> [Int] -> [Int]

hiMerge f g h a b []     ys     = b ys
hiMerge f g h a b xs     []     = a xs
hiMerge f g h a b (x:xs) (y:ys) | x < y  = f x (hiMerge f g h a b xs (y:ys))
                                | x == y = g x (hiMerge f g h a b xs ys)
                                | x > y  = h y (hiMerge f g h a b (x:xs) ys)

union        = hiMerge (:) (:) (:) id id
intersection = hiMerge (curry snd) (:) (curry snd) (const []) (const [])
difference   = hiMerge (:) (curry snd) (curry snd) id (const [])
```

Figure 1: Higher-order merge and operations on sets

2 Operational Semantics and Definitional Trees

In this section we introduce the main concepts needed to understand the operational principles of the subset of Curry considered in this paper. For a complete description of the operational semantics of Curry the reader is referred to the draft [7].

The main operational mechanism we will deal with is *narrowing*. Narrowing can be viewed as a generalisation of term rewriting where matching is replaced by unification. Informally, to *narrow* an expression e means to apply a substitution that makes it reducible, and then reduce it. For instance, a rule

```
tail (h:ts) = ts
```

which could be part of a Haskell program, can be queried in Curry with

```
eval tail [a,b,c]
```

where `a`, `b` and `c` are free variables, i.e. we try to compute with partial information, asking for the tails of lists with 3 elements. In this case a most general answer is obtained by unifying `(h:ts)` with `[a,b,c]` and applying the resulting unifier to `ts`, i.e. `[b,c]` is obtained as result.

The usual situation is to have partial and non-deterministic goals, where a result is defined only for certain values of its free variables. Consider, for instance, asking

```
eval [1,2,3]=(as++bs)
```

Four different substitutions ($\{$`as` $\mapsto$ `[]`, `bs` $\mapsto$ `[1,2,3]`$\}$, $\{$`as` $\mapsto$ `[1]`, `bs` $\mapsto$ `[2,3]`$\}$, etc) satisfy this equation. In general, the outcome of a query is a possibly infinite series of pairs (*result*, *condition*).

Example 1 The following is a possible definition of a predicate to test membership of an element to a list:

```
member x xs = xs=(prefix++[x]++suffix)
```

The query

```
eval member x [1,2,3]
```

succeeds with the bindings $\{\mathtt{x} \mapsto 1\}$, $\{\mathtt{x} \mapsto 2\}$, $\{\mathtt{x} \mapsto 3\}$. □

More formally, e is *narrowable into* e', *at occurrence* u, *using* $l := r$, *via* σ_{out}, iff $e_{|u} \notin Vars$, $l\sigma_{\text{in}} \equiv e_{|u}\sigma_{\text{out}}$ and $e' \equiv e[u \leftarrow r\sigma_{\text{in}}]\sigma_{\text{out}}$, $(Dom(\sigma_{in}) \cap Dom(\sigma_{out}) = \emptyset.)$ This is written

$$e \leadsto_{u,l:=r,\sigma_{\text{out}}} e'$$

or simply $e \leadsto_{\sigma_{\text{out}}} e'$, when the position and rule can be determined from the context. If $\sigma_{\text{in}}\sigma_{\text{out}}$ is a *mgu* of l and $e_{|u}$, this is called a *most general* narrowing step. We write $e_0 \leadsto^*_\sigma e_n$ to denote that there is a narrowing sequence $e_0 \leadsto_{\sigma_1} e_1 \leadsto_{\sigma_2} \cdots \leadsto_{\sigma_n} e_n$ with $\sigma = \sigma_n \cdots \sigma_2\sigma_1$
If $e \leadsto^*_{\sigma_{\text{out}}} e'$ holds and e' is in nf, we speak of a *computation* for the goal e with *result* e' and *answer* σ_{out}, denoted

$$e \Downarrow e'[\![\sigma_{\text{out}}$$

Otherwise, if e' is not in nf and cannot be further narrowed, we speak of a *failed computation.*

Unrestricted application of narrowing is too nondeterministic and many strategies have been proposed to improve this [6]. The one used in Curry is *needed narrowing*[2], a lazy strategy where the program is translated into a set of definitional trees, one for every function symbol being defined.

The following is abstract syntax for definitional trees:

$$\begin{array}{rcl} DT & ::= & \text{branch } (\mathit{Pattern}, \mathit{Occ}\ [, DT]^+) \\ & | & \text{rule } \mathit{Rule} \\ & | & \text{or } (DT\ [, DT]^+) \end{array}$$

Pattern stands for patterns made up of data constructors and different variables as in the left hand sides of program rules, *Occ* are occurrences defined in the standard way and *Rules* are program rules. Trees without "or" nodes are called *sequential*, otherwise they are *parallel* definitional trees. Definitional trees are used like the *case*-expressions in [21]:

Definition 1 (Definitional Tree) T is a *definitional tree* for pattern π iff one of the following cases holds:

1. T is $\mathit{rule}(l = r)$ where $l = r$ is a program rule such that l is a variant of π.
2. T is $\mathit{branch}(\pi, o, T_1, \ldots, T_k)$ where the T_i are definitional trees for $\pi[c\ x_1 \ldots x_{ac}]_o$, being c a data constructor suitable to appear at o and the x_j fresh variables.
3. T is $\mathit{or}(T_1, \ldots, T_k)$ where the T_i are definitional trees for π.

Example 2 Consider the following definition:

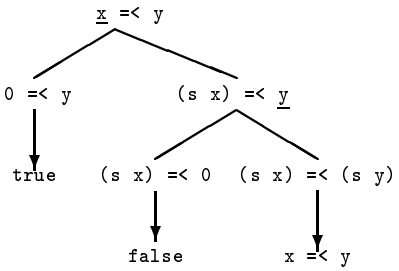
```
0     =< y     = true
(s x) =< 0     = false
(s x) =< (s y) = x =< y
```

The tree[3]

[3]We have not printed the π's as they can be inferred from the leaves of the tree.

branch(1, *rule*(`0 =< y = true`),
branch(2, *rule*(`(s x) =< 0 = false`),
rule(`(s x) =< (s y) = x =< y`)))

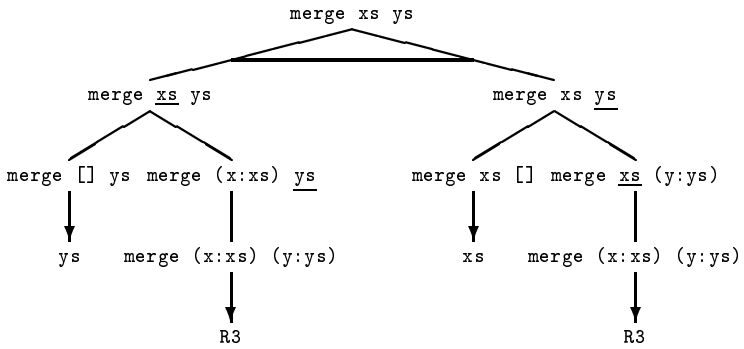
is a definitional tree for `x =< y` that covers all the rules in its definition. It is more graphically displayed as:



Given expression e built with the symbols in program P and a set of definitional trees for the defined symbols in P an occurrence in e can be chosen to apply narrowing. This is done by first looking for an outermost application $f\ e_1 \ldots e_n$ where f is a defined function symbol, and then descending some e_i according to f's definitional tree. A detailed definition of the strategy can be found in [1].

Now an example of a parallel definitional tree:

Example 3 The following is a definitional tree for `merge xs ys` comprising the definition above:



Parallel trees are not as good at specifying control as sequential ones. On one hand, although a generalization of the aforementioned strategy exists to sets of occurrences, this is in general too nondeterministic. For instance, it is clear that as soon as one of the arguments to `merge` is in *hnf* the parallel reduction of the other argument can be stopped and then jump to purely sequential code, but this is not explicit in the tree above. Observe as well that the 3rd. rule is covered by both subtrees.

On the other hand, consider a expression `merge (f xs ys) (g xs zs)` where `xs` is a (free) logical variable shared among the two arguments to `merge`. It is well known from logic programming that parallel evaluation of goals with shared variables is inherently inefficient.

3 Denotational Semantics

A simple denotational semantics for an *equational* subset of Curry follows. This is a purely functional semantics, that is, it does not capture such aspects as computed answers, but suffices for the purposes of this paper.

The approach is similar to that of [13], but has been modified in order to fit into the metalanguage of [9], inspired by [10]. This makes proving the continuity of the interpretation transformer and the safety of the subsequent approximation a straightforward task. Another difference with [13] is that a higher order language is now considered.

We first define some of the semantic domains involved. H is the cpo completion of the Herbrand universe formed with all the (data) constructors in a program. The (higher order) domain of values D is given as the least solution to the equation

$$D \cong H + [D_\perp \to D_\perp] + \sum_i \{(c\ d_1 \ldots d_i) | c \in DC_i,\ \forall k.d_k \in D\}$$

An interpretation for a certain program maps every function symbol to a value in D.

$$Int = FS \to D$$

We regard constructors as free, and thus have a standard denotation.
The semantics can be presented by means of the following semantic functions:

$$\begin{array}{l} \mathbf{F} : Int \\ \mathbf{R} : Rule \to Int \to Int \\ \mathbf{E} : Exp \to Int \to D \end{array}$$

E is just evaluation of expressions according to the semantics of the program, and can be defined as the homomorphic extension of the semantics for function symbols (**F**). It is needed in order to evaluate the right hand sides of rules.

R is the interpretation transformer associated with each rule of the program and represents the amount of information added by every possible application of that rule.

$$\begin{array}{rcl} \mathbf{R}[\![f\ t_1 \ldots t_n := r]\!]i & = & \lambda x_1 \ldots x_n. \bigsqcup_{Var(t_j)} (t_1 \doteq x_1 \wedge \cdots \wedge t_n \doteq x_1) \to \mathbf{E}[\![r]\!]i \\ \mathbf{F} & = & lfp(\bigsqcup_r (\mathbf{R}[\![r]\!])) \end{array}$$

The symbol ($\doteq$) denotes domain equality and ($\cdot \to \cdot$) is shorthand for ($\cdot \to \cdot | \perp$).

Example 4 The meaning of addition, as defined by the rules

```
0     + m = m
(s n) + m = s(n + m)
```

is given by

$$\mathbf{F}[\![+]\!] = \lambda x.\lambda y.lfp(\lambda i.(x \doteq 0 \to \mathbf{E}[\![y]\!]i) \sqcup (\bigsqcup_{N,M} (x \doteq (s\ N) \to \mathbf{E}[\![s(N+M)]\!]i)))$$

after some manipulation of the formula. □

For every equational program, **F**, **R** and **E** are continuous.

4 Theory of Approximation

Due to space limitations, this section has been eliminated from the final copy version. A detailed presentation will appear in [9].

5 A Simple Strictness Analysis for Equational Programs

By "simple" we mean a forward strictness analysis much in the style of [14] (two-point domain), but higher order. The differences are in the presentation, the treatment of pattern matching and of higher-order functions.

A strictness analysis can predict undefinedness in functional expressions. Domain-theoretically, this amounts to detect bottoms as the result of semantic (complete) functions. In our system, only D will be treated as dynamic, and its approximation is obtained by approximating its first-order subdomains by 2-point lattices where $\bot$ represents undefined values and $\top$ represents everything, i.e. the supremum of the powerdomain.

In other words, we have

$$D^\sharp \cong 2 + [D^\sharp \to D^\sharp]$$

as the abstract domain. The idea is then to define an α such that $(D, \alpha, D^\sharp)$ is an approximation. The $(\cdot^\sharp)$ notation stands for $\alpha(\cdot)$, as usual.

Example 5 If we consider our program for addition then we have

$$\bot^\sharp = \bot,\ 0^\sharp = \top,\ s^\sharp = \lambda x.\top$$

which implies, of course, $\bot \propto \bot$, $\top \propto 0$, $\lambda x.\top \propto s$, etc. Notice that due to the notion of approximation relation , functions are abstracted out as functions. □

Having obtained best approximations for constructors, the abstract semantics follows just by computing $\mathbf{F}$ in the new domain.

Before getting back to our example, let us note that the operations present in the metalanguage have well-known behaviours in the abstract domain. Least upper bounds in 2 are just logical disjunction, of course. For higher types we have "higher order disjunction" defined recursively as

$$(f \vee g)\ x = (f\ x) \vee (g\ x)$$

Parametric lub's, like in

$$\bigsqcup_{N,M} (x \doteq (s\ N) \to \mathbf{E}[\![s(N+M)]\!]i)$$

correspond to existential quantification. Guarded expressions $(b \to e)$ behave as conjunction — apply Shannon. Finally, domain equality behaves as coimplication.

Example 6 Back to our example. The strictness analysis of the addition rules is

$$\begin{aligned}
\mathbf{F}^\sharp[\![+]\!] &= \lambda x.\lambda y.\mathit{lfp}(\lambda i.(x \leftrightarrow \top \wedge \mathbf{E}^\sharp[\![y]\!]i) \vee (\exists n, m.\ x \leftrightarrow \top \wedge \mathbf{E}^\sharp[\![s(N+M)]\!]i)) \\
&= \lambda x.\lambda y.\mathit{lfp}(\lambda i.(x \leftrightarrow \top \wedge y) \vee (\exists n, m.\ x \leftrightarrow \top \wedge \top)) \\
&= \lambda x.\lambda y.(x \wedge y) \vee x \\
&= \lambda x.\lambda y.x
\end{aligned}$$

That is, the analysis says that if the first argument is undefined so is the result. □

For every equational program $\mathbf{F}^\sharp \propto \mathbf{F}$, which is straightforward from the metalanguage in [9].

5.1 Computing the Approximation

Effectively computing the abstract denotation for the function symbols in a program implies the ability of manipulating propositional formulae and deciding on their equality — in order to detect fixed points.

Our implementation is based on BDD's (binary decision diagrams) which are normalized representations of propositional formulae. Thanks to normalisation, coimplication reduces to syntactic equality. This more or less fits our needs, whenever the functions tested for equality are of the form

$$\lambda x_1.\lambda x_2 \ldots \lambda x_n.P$$

where P is a propositional formula. That is, in order to test the equality of $\lambda\tilde{x_i}.P$ and $\lambda\tilde{y_i}.Q$ the lists of parameters $[\tilde{x_i}]$ and $[\tilde{y_i}]$ are renamed out with a unifier θ and then $P\theta$ and $Q\theta$ are tested for logical equivalence.

Example 7 We want to check the equality of the boolean functions

$$\lambda x.\lambda y.\ x \to y \quad \text{and} \quad \lambda w.\lambda z.\ z \vee \neg w$$

Renaming formal parameters with the substitution $[x \mapsto p, y \mapsto q, w \mapsto p, z \mapsto q]$ gives propositional formulae

$$p \to q \quad \text{and} \quad q \vee \neg p$$

Both normalize as $(p \wedge ((q \wedge \top) \vee (\neg q \wedge \bot))) \vee (\neg p \wedge \top)$ and thus the original functions were equal. □

5.2 Dealing with Function Spaces

When higher order functions like `map`, `foldr`, etc, occur in the program the former method no longer applies.

Example 8 Consider the definition of the ubiquitous function `foldr`:

```
foldr f b []     = b
foldr f b (h:ts) = f h (foldr f b ts)
```

The strictness semantics gives

$$\begin{aligned}\mathbf{F}^\sharp[\![\text{foldr}]\!] &= \lambda f.\lambda b.\lambda xs.lfp(\lambda i.(xs \leftrightarrow \top \wedge \mathbf{E}^\sharp[\![b]\!]i) \vee (\exists h, ts.\ xs \leftrightarrow \top \wedge \mathbf{E}^\sharp[\![f\ b\ (\text{foldr}\ f\ b\ ts)]\!]i)) \\ &= \lambda f.\lambda b.\lambda xs.lfp(\lambda i.(xs \wedge b) \vee (\exists h, ts.\ xs \wedge f\ b\ ((\mathbf{E}^\sharp[\![\text{foldr}]\!]i)\ f\ b\ ts)))\end{aligned}$$

which is not an abstraction of a propositional formula. □

Unsurprisingly, the abstraction of a higher-order function is a higher-order boolean function. A possible way to overcome this is to translate equality between higher order boolean functions to an isomorphic first order problem.

If we have $f : \text{bool} \to T$ then by Shannon's expansion theorem

$$f(x) \leftrightarrow (x \wedge f(\top)) \vee (\neg x \wedge f(\bot))$$

— where $(\vee, \wedge)$ are really higher order disjunction and conjunction if T is a function type. Making $f_\bot = f(\bot)$, $f_\top = f(\top)$ we obtain an isomorphism

$$(\text{bool} \to T) \leftrightarrow (T \times T)$$

i.e. a function type is translated into a product which in turn can be treated via currying.

Example 9 The higher order function $F : (\text{bool} \to \text{bool}) \to \text{bool} \to \text{bool}$ defined as

$$F = \lambda f.\lambda x.\ f(\top) \wedge x$$

translates to $F' : \text{bool} \to \text{bool} \to \text{bool} \to \text{bool}$ with definition

$$F' = \lambda f_\perp.\lambda f_\top.\lambda x.\ f_\top \wedge x$$

For types $(H \to T)$ with $H \neq \mathit{bool}$ the combinatory explosion seems excessive to apply this method. Fortunately enough, this kind of "very" higher order functions where some arguments are expected to be functions which expect functions as arguments do not occur very frequently in real programs, so we consider this a limitation of our analyzer. The following definition tries to formalize this idea:

Definition 2 (Degree of a type) The degree of a type, notation **Deg**, is defined recursively in the following way:

$\mathbf{Deg}(\alpha)$	$=$	0	with α a type variable
$\mathbf{Deg}(\kappa\ \tau_1 \ldots \tau_n)$	$=$	$max\{\mathbf{Deg}(\tau_{ij})\}$	with κ a type constructor given by `data` $\kappa\ \tau_1 \ldots \tau_k$ $=$ $c_1\ \tau_{11} \ldots \mid c_j\ \tau_{n1}\ \ldots \mid \ldots$
$\mathbf{Deg}(\tau_1 \to \tau_2 \to \ldots \tau)$	$=$	$1 + max\{\mathbf{Deg}(\tau_i)\}$	with τ not a function type

To empirically support that degrees over 2 are rare, the reader is referred to [4, 18], for example.

6 Application to Sequentiality Analysis

Here we show how to apply a strictness analysis to the compilation of equational programs so as to produce sequential code in most cases. In short, the idea is to search for a sequential definitional tree. If found, we are done. If not, look for the causes of the non-sequentiality, and use that information to help the strictness analyzer to find the desired solutions.

Definition 3 (Conflict) Let l_1 `=` r_1 and l_2 `=` r_2 be two rules defining the same function symbol in a given program. It is said that there is a *conflict* between those rules at position o iff $l_{1|o}$ is a variable, $l_{2|o}$ is not, and for every p strict prefix of o, $root(l_{1|p}) = root(l_{2|p})$

Conflicts are a necessary condition — but not sufficient — for the lack of sequentiality in a function's definition. The interesting point in the last definition is the last requirement which makes them less likely to appear in deep positions of a pattern. Due to space limitations we will restrict ourselves to the treatment of "topmost" conflicts.[4]

Let f be a function symbol for which a sequential tree could not be built, and let $f\ \pi_1 \ldots x \ldots \pi_n$ `=` r be a conflicting rule for f. That implies x is of a constructed type instance $\kappa\ \tau_1 \ldots \tau_k$ defined by

$$\texttt{data}\ \kappa\ \tau_1 \ldots \tau_k = c_1\ \tau_{11} \ldots \mid c_j\ \tau_{n1}\ \ldots \mid \ldots$$

[4] Deeper conflicts can be approached in two ways: with a source code transformation, similar to those in [11], that brings the conflicts to the top positions, or using a richer strictness analyzer.

If the analysis says that f is strict in x's position then the conflicting rule can be replaced by the set of rules resulting from applying the substitution $\{x \mapsto c_i\ y_1 \ldots\ y_{ai}\}$ to it, for every constructor in the type declaration, where the y's are fresh variables. After deleting subsumed rules the resulting set can be sequential, otherwise a different position is tried. This transformation is safe w.r.t. the lazy behaviour due to the strictness of f, and although nondeterministic, the number of conflicting positions is usually small, as said above.

We will see with an example that this procedure, while natural, is not entirely satisfactory for functional-*logic* programs.

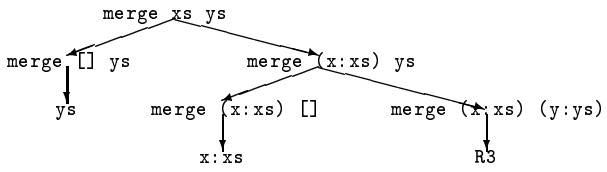
Example 10 Take the rules for the `merge` function:

```
merge []     ys     = ys      (1)
merge xs     []     = xs      (2)
merge (x:xs) (y:ys) = (R3)    (3)
```

the left hand sides are essentially a parallel-or, but the strictness analysis gives $\texttt{merge}^\sharp = \lambda xs.\lambda ys.xs \wedge ys$. Trying the conflict at position 1 we get $\texttt{merge}^\sharp\ \bot\ \top = \bot$, so the second rule is rewritten into:

```
merge []     [] = []      (2a)
merge (x:xs) [] = x:xs    (2b)
```

Rule (2a) is subsumed by (1) and the set of rules is sequential with the following definitional tree:



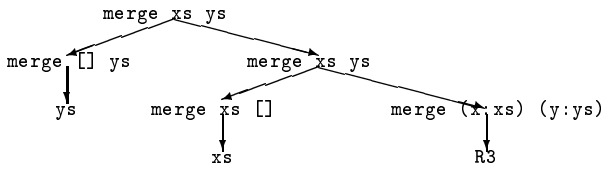
Unfortunately, a query `eval merge any_list []` will give an unexpected outcome with the transformed rules:

$$\texttt{x:xs} \llbracket \{\texttt{any\_list} \mapsto \texttt{x:xs}\}$$

instead of the more natural $\texttt{any\_list} \llbracket \epsilon$

The reason is that in changing the actual rules of the program, while the (functional) denotation of the program does not change, the operational or *computed-answers* semantics is different. One possible solution to this anomaly is to keep the shape of the sequential tree while replacing the leaves with the original rule, when appropriate, and to relax the patterns at every *branch* node of the tree in the path from that leaf upwards accordingly.

For our example the following decision tree would be obtained:



It can be seen that definition 2 is too restrictive to take advantage of strictness information not present syntactically in the left hand sides. More specifically, the property that different branches in a *branch* node are indexed by different constructors at a given position should be separated from the actual substitution applied at that node.

FUNCTION	STRICTNESS SEMANTICS
`id`	$\lambda x.x$
`(:)`	$\lambda x.\lambda y.\top$
`curry snd`	$\lambda x.\lambda y.y$
`[]`	$\top$
`const`	$\lambda x.\lambda y.x$
`const []`	$\lambda y.\top$
`hiMerge`	$\lambda f.\lambda g.\lambda h.\lambda a.\lambda b.\lambda xs.\lambda ys.(xs \wedge b\ ys) \vee (ys \wedge a\ xs)$
`union`	$\lambda xs.\lambda ys.(xs \wedge ys)$
`intersection`	$\lambda xs.\lambda ys.(xs \vee ys)$
`difference`	$\lambda xs.\lambda ys.xs$

Figure 2: Strictness analysis for the hiMerge example

7 Results and Conclusion

Curry, unlike most declarative languages, has included in the current draft the operational behaviour associated with the compilation of patterns as part of the language definition. This is due, in part, to the fact that in the same way as in a functional language with lazy evaluation, sequentiality is more crucial than in an eager one, in a functional-logic language there is a greater risk of speculation. The reason is that parallel strategies are harder to implement in the presence of partial information.

We have shown that although complete, present compilation techniques in Curry are inefficient for non strongly sequential programs, and that it is feasible to produce a more efficient unification code using information from a strictness analyzer. This means taking into account the bodies of the rules, not only the left hand sides.

Even in the presence of higher order functions — a key feature of functional-logic languages — strictness can be used to improve sequentiality. Figure 2 shows the results of applying the method to the example in figure 1. For the three functions `union`, `difference`, `intersection`, only `intersection` would need, in the worst case, parallel code or run-time tests.

It has been shown that the formalism of definitional trees must be enhanced to take full advantage of the analysis. The reason is that current definitional trees assume that all indexing information comes from constructor symbols actually present in the program rules and this is used to produce output substitutions.

Rather than expecting too much from a powerful strictness analyzer we have relied in reasonable approximations to take advantage of the simplest. For instance, the treatment of higher order domains is justified by a measure of the type of the function analyzed, and the applicability of a simple strictness analysis is based on the way sequentiality conflicts tend to appear.

This commitment to the pragmatics of a programming language distinguishes this paper from other work (e.g. [19]) that approaches the issue of (non strong) sequentiality in the field of TRS's. Questions on the use of a programming language and notions like its semantics (and its abstractions) or higher order are usually sidestepped.

As an actual compilation technique this is is still work in progress, i.e. Sloth is not yet able to improve its definitional trees from the strictness information. The method sketched in section 6 is too nondeterministic and we are currently working in a more constructive method and a formulation of definitional trees that distinguishes between indexes and output substitutions.

In this paper we have considered narrowing as the only evaluation mechanism. Curry

allows goals to be evaluated using both narrowing and *residuation* — essentially suspension of goals with free variables. Recently the treatment of residuation, with evaluation annotations, has been integrated into the formalism of definitional trees, making its extension a less trivial question than shown here.

References

[1] Sergio Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic Programming*, number 632 in Springer LNCS, pages 143–157, 1992.

[2] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Proc. 21st. ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[3] Sergio Antoy, Rachid Echahed, and Michael Hanus. Parallel evaluation strategies for functional logic languages. In *International Conference on Logic Programming*, 1997.

[4] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[5] Daniela Genius. Sequential implementation of parallel narrowing. In *Proc. JICSLP'96 Workshop on Multi-Paradigm Logic Programming*, pages 95–104, 1996.

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

[7] Michael Hanus, Sergio Antoy, Herbert Kuchen, Francisco López Fraguas, and Frank Steiner. Curry: an integrated functional logic language. `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/report.html`.

[8] Jan Willem Klop. *Term Rewriting Systems*, volume 1 of *Handbook of Logic in Computer Science*, chapter 6. Oxford University Press, 1991.

[9] Julio Mariño. Denotational abstract interpretation of functional logic programs. To appear. Look under `http://lml.ls.fi.upm.es/~labman/papers`.

[10] Kim Marriot, Harald Søndergaard, and Neil Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, may 1994.

[11] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. ALP'90, pp. 298–317

[12] Juan José Moreno Navarro. Expressivity of functional-logic languages and their implementation. GULP-PRODE'94, Servicio de publicaciones Universidad Politécnica de Valencia.

[13] Juan José Moreno-Navarro. Extending constructive negation for partial functions in lazy functional logic languages. In *Extensions of Logic Programming*, number 1050 in LNAI, pages 213–228. Springer-Verlag, 1996.

[14] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. International Symposium on Programming*, pages 269–281. Springer LNCS 83, 1980.

[15] Michael O'Donnell. *Handbook of Logic in Artificial Intelligence and Logic Programming*, chapter Equational Logic Programming. Oxford University Press, 1994.

[16] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[17] José María Rey and Julio Mariño. Implementación de Curry mediante su traducción a Prolog. Master's thesis, Facultad de Informática, Universidad Politécnica de Madrid, 1997. In Spanish.

[18] Colin Runciman, editor. *Applications of Functional Programming.* UCL Press, 1995.

[19] R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. *Information and Computation*, 1(104):78–109, 1993.

[20] S. Thatte. A refinement of strong sequentiality for term rewriting with constructors. *Information and Computation*, 72:46–65, 1987.

[21] Philip Wadler. Efficient compilation of pattern matching. In Peyton-Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice-Hall, 1987.