Evaluation Strategies
for
Functional Logic Programming

WRS'01, Utrecht, May 26

Sergio Antoy

`antoy@cs.pdx.edu`

Portland State University

Outline

Whats, hows and whys of Functional Logic Programming.

Constructor-based rewrite systems as programs.

The role of strategies.

The classes of rewrite systems.

A strategy for each class.

What is FLP

FLP studies programming languages that integrate functional and logic programming.

A program in these languages is a constructor-based, possibly conditional, rewrite system.

A computation is a rewrite or narrowing derivation of a term to a constructor term.

A strategy is the most critical component of a FLP language.

FLP Features

- Logic variables (partial structures)
- Inversion of computations
- Non-determinism
- Infinite structures
- Functional nesting
- Strategies

FLP Advantages

- Expressiveness (compute more with less code)
- Transparency (code is not cryptic/opaque)
- Economy (save code)
- $\Rightarrow$ Improved entire software cycle

Example

N-queens puzzle

```
queens X -> Y :- Y=permute X, void (capture Y)
permute [] -> []
permute [X|Xs] -> U++[X]++V :- U++V=permute Xs
capture Y :- _++[Y1]++K++[Y2]++_=Y, abs(Y1-Y2)=length K+1
```

This program is (for the most part) a constructor-based conditional rewrite system (with several liberties concerning the traditional notation).

`queens [1,2,3,4]` evaluates (among others) to `[2,4,1,3]`

Execution

The execution of an FLP program is by narrowing and/or residuation.

Ground (sub)terms are simply rewritten.

Non-ground (sub)terms are either instantiated enough to be rewritten or suspended until they become instantiated enough.

A strategy schedules the (sub)terms to evaluate, decides whether to suspend or to instantiate and if the latter, computes the instantiation.

Different classes of rewrite systems are better executed by different strategies.

Strategies

A strategy is a mapping from terms to set of steps.

When it is applied to a term it computes both a reduct (computed value) and an instantiation (computed answer) of some variables of the term.

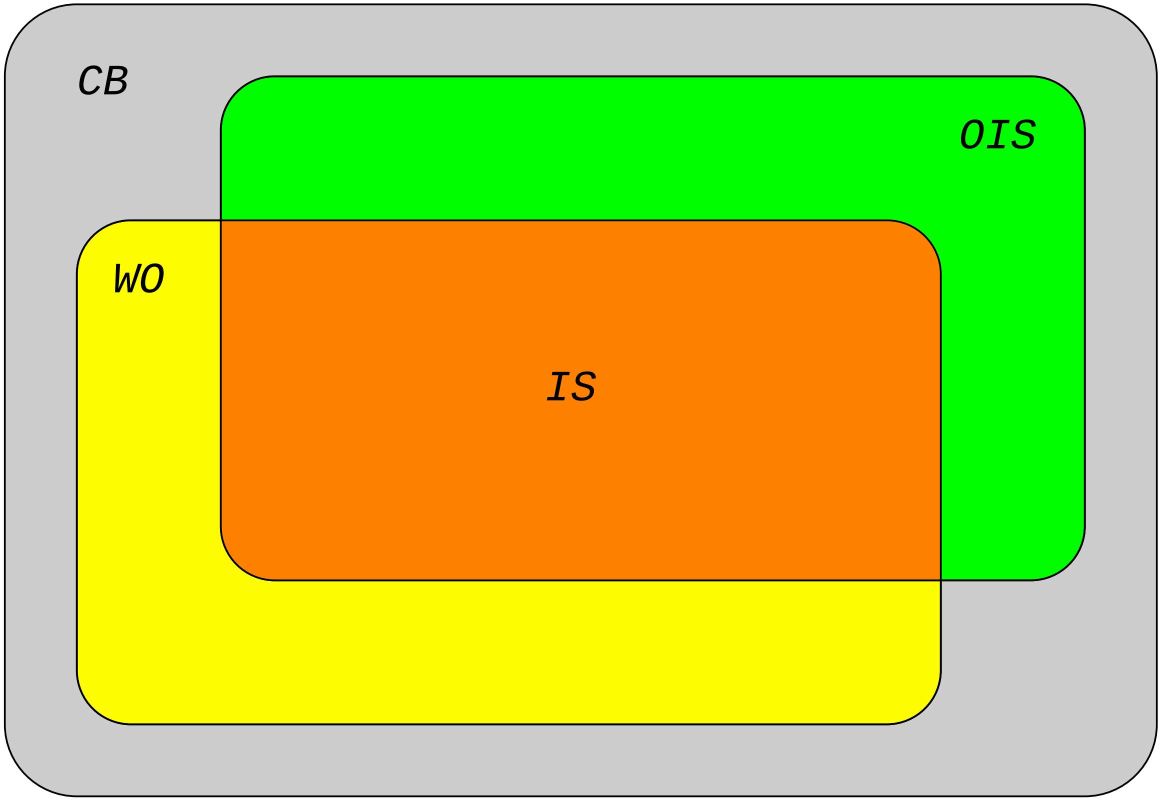
A useful strategy must have a number of desirable properties that are better explained when the term is an equation with free variables.

Soundness: every computed answer is a solution of the equation.

Completeness: every solution of the equation is computed.

Efficiency: computational resources are not wasted.

Classes of Rewrite Systems

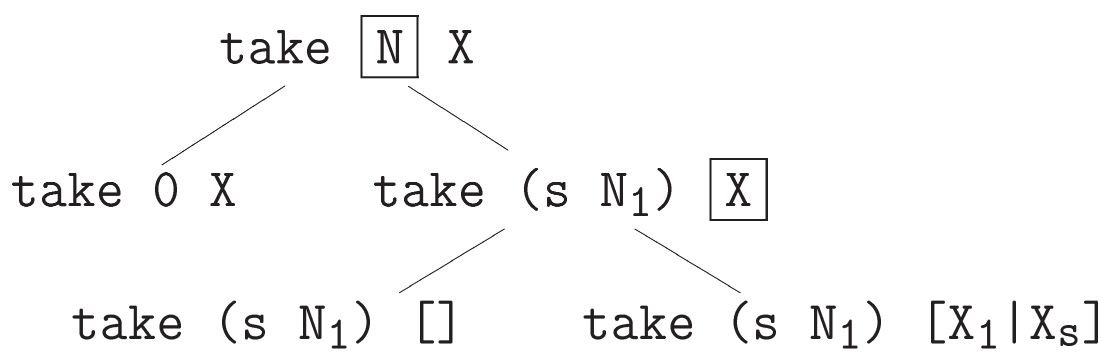


Inductively Sequential TRSs

- Constructor-based subclass of the strongly sequential TRSs

```
take 0 _ -> []
take (s N) [] -> []
take (s N) [X|Xs] -> [X | take N Xs]
```

- The left-hand sides of the rules of each operation can be organized a in hierarchical structure called a definitional tree



Needed Narrowing

The strategy for the inductively sequential TRSs is needed narrowing.

Needed narrowing is easily implemented using a definitional tree as an automaton to compute steps.

Needed narrowing satisfies two important optimality criteria.

Only needed steps are computed: derivations have minimal length.

Only needed derivations are computed: substitutions are disjoint.

Compute a Needed Step

The task is to evaluate t = `take N` u, where `N` is an unbound variable and u is some operation-rooted term.

Unify t with a maximal (lowest) element of the definitional tree. There are two such elements: `take 0 X` (a leaf) and `take (s` $\mathtt{N}_1$`) X` (a branch).

Because the first is a leaf, instantiate t with $\{N \mapsto 0\}$ and narrow. The result is []

Because the second is a branch, instantiate t with $\{N \mapsto (s\ N_1)\}$ and recursively evaluate the match of the inductive variable. An instance of u is evaluated.

Weakly Orthogonal TRSs

- Rewrite rules's lhs may overlap, but only if critical pairs are trivial.
- Interesting TRSs because they capture parallelism.

```
true ∨ _ -> true
_ ∨ true -> true
false ∨ false -> false
```

- There is no definitional tree of operation "$\vee$".
- There are terms that have no needed redex.
- In practice, to evaluate a term of the form $u \vee v$ both u and v must be evaluated concurrently.

Parallel Narrowing

The strategy for the weakly orthogonal TRSs is parallel narrowing.

The rules of a weakly orthogonal operation can be partioned into subsets so that every subset has a definitional tree.

The set of steps obtained by computing a step for each subset of the partition is a necessary set of steps.

One step of a necessary set of a term t is executed in any computation of t to a constructor term.

Parallel narrowing is a conservative extension of both needed narrowing and weakly needed rewriting.

Overlapping Inductively Sequential TRSs

- Rewrite rules's lhs may overlap, but only if they are equal modulo a renaming of variables.
- Interesting TRSs because they capture non-determinism.

```
regexp X -> X
          | "(" ++ regexp X ++ ")"
          | regexp X ++ regexp X
          | regexp X ++ "*"
          | regexp X ++ "|" ++ regexp X
```

- To recognize whether a string s is a well-formed regular expression over $\{0, 1\}$ one evaluates $\texttt{regex}\ ("0"|"1") = s$.

INS

The strategy for the overlapping inductively sequential TRSs is *INS*.

Overlapping inductively sequential operations have a definitional tree.

INS steps are computed as needed steps, except that several alternative replacements may be available for a narrex.

The optimality results of needed narrowing holds for *INS,* but only modulo non-deterministic choices.

Needed and possibly non-needed steps and/or derivations are computed by *INS*.

Constructor-Based TRSs

- No specific restrictions on the rewrite rules except for the constructor discipline and left linearity.
- Greatest expressive power:

```
permute [] -> []
permute [X|Xs] -> insert X (permute Xs)
insert X Ys -> [X|Ys]
insert X [Y|Ys] -> [Y|insert X Ys]
```

- No accepted notion of "need." Known strategies are demand-driven and their properties are poorly understood.
- There exists a semantics-preserving transformation into the overlapping inductively sequential TRSs.

Transforming Conditions

- The conditional part of a rule is a sequence of equational constraints:

$$l \to r \Leftarrow u_1 = v_1, \ldots, u_n = v_n$$

  expressed by ordinary operations: conjunction, strict equality, ...

- A new conditional operation is added to the signature:

$$\texttt{if success then X} \to \texttt{X}$$

- Rules are deconditionalized by moving the condition to the rhs in the form of a conditional expression:

$$l \to \texttt{if}\ \ u_1 = v_1,\ \ldots,\ u_n = v_n\ \ \texttt{then}\ \ r$$

Transforming Overlapping

- The set of rules of an operation f is partitioned into overlapping inductively sequential subsets.
- Each subset of a partition defines a new overlapping inductively sequential operation, say $f_1, f_2, \ldots, f_n$.
- Operation f is replaced by

$$f(\bar{X}) \to f_1(\bar{X}) \mid \cdots \mid f_n(\bar{X})$$

- The choice to evaluate an argument of an operation is transformed into a choice of one of several rhs's.

Transformation Example

The following operation:

$$\texttt{insert X Y}_\texttt{s}\ \texttt{->}\ \texttt{[X|Y}_\texttt{s}\texttt{]}$$
$$\texttt{insert X [Y|Y}_\texttt{s}\texttt{] -> [Y|insert X Y}_\texttt{s}\texttt{]}$$

is transformed into:

$$\texttt{insert X Y}_\texttt{s}\ \texttt{-> insert}_1\ \texttt{X Y}_\texttt{s}\ \texttt{| insert}_2\ \texttt{X Y}_\texttt{s}$$
$$\texttt{insert}_1\ \texttt{X Y}_\texttt{s}\ \texttt{-> [X|Y}_\texttt{s}\texttt{]}$$
$$\texttt{insert}_2\ \texttt{X [Y|Y}_\texttt{s}\texttt{] -> [Y | insert X Y}_\texttt{s}\texttt{]}$$

Some optimizations are possible, e.g., $\texttt{insert}_1$ can be eliminated.

Transformation Properties

- Let $\mathcal{R}$ be a left-linear constructor-based TRS and $\mathcal{S}$ the transformed TRS.
- $\mathcal{S}$ is an overlapping inductively sequential TRS.
- The constructor terms of $\mathcal{R}$ and $\mathcal{S}$ are the same.
- For each computation $t \xrightarrow{*} v$ in $\mathcal{R}$ there exists a computation $t \xrightarrow{*} v$ in $\mathcal{S}$.

Conclusion: all the computations of $\mathcal{R}$ are simulated by computations of $\mathcal{S}$.

Summary

Narrowing and rewriting strategies for the constructor-based TRSs have been extensively investigated and are well understood.

Four subclasses are emerging for functional logic programming.
Each class has at least one effective strategy.

Inductively Sequential: functional computations, needed narrowing.

Weakly Orthogonal: parallel computations, parallel narrowing.

Overlapping Inductively Sequential: non-deterministic computations, *INS*.

Left-Linear: transform into overlapping inductively sequential.

Acknowledgments

The organization of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001) gave me the opportunity me to work on this survey and invited me to present this talk.

The National Science Foundation supported in part this research under grants INT-9981317 and CCR-9406751.