ICLP 2006

Overlapping Rules and Logic Variables
in
Functional Logic Programs

Michael Hanus

Christian-Albrechts-Universität Kiel

(joint work with Sergio Antoy, Portland State University)

FUNCTIONAL LOGIC LANGUAGES

Approach to amalgamate ideas of declarative programming

- efficient execution principles of functional languages
  (determinism, laziness)
- flexibility of logic languages (constraints, built-in search)
- avoid non-declarative features of Prolog (arithmetic, I/O, cut)
- combine best of both worlds in a single model
  (higher-order functions, declarative I/O, concurrent constraints)
- Advantages:
  - ➜ optimal evaluation strategies [JACM'00,ALP'97]
  - ➜ new design patterns [FLOPS'02]
  - ➜ better abstractions for application programming
    (GUI programming [PADL'00], web programming [PADL'01, PPDP'06])

FUNCTIONAL LOGIC LANGUAGES

Approach to amalgamate ideas of declarative programming

- efficient execution principles of functional languages
  (determinism, laziness)

- flexibility of logic languages (constraints, built-in search)

- avoid non-declarative features of Prolog (arithmetic, I/O, cut)

- combine best of both worlds in a single model
  (higher-order functions, declarative I/O, concurrent constraints)

- Advantages:
  - ➜ optimal evaluation strategies [JACM'00,ALP'97]
  - ➜ new design patterns [FLOPS'02]
  - ➜ better abstractions for application programming
    (GUI programming [PADL'00], web programming [PADL'01, PPDP'06])

Minimal kernel language for FLP?

FUNCTIONAL LANGUAGES (E.G., HASKELL)

FUNCTIONAL LANGUAGES (E.G., HASKELL)

```
[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

FUNCTIONAL LANGUAGES (E.G., HASKELL)

+ logic variables (expressive power)

```
[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys

last xs | ys ++ [x] =:= xs  = x      where x,ys free
```

FUNCTIONAL LANGUAGES (E.G., HASKELL)

+ logic variables (expressive power)

```
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys

last xs | ys++[x] =:= xs  = x     where x,ys free
```

+ overlapping rules (demand-driven search)

```
insert e []     = [e]
insert e (x:xs) = e : x : xs           -- overlapping
insert e (x:xs) = x : insert e xs      -- rules
```

FUNCTIONAL LANGUAGES (E.G., HASKELL)

+ logic variables (expressive power)

```
[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys

last xs | ys ++ [x] =:= xs  = x      where x,ys free
```

+ overlapping rules (demand-driven search)

```
insert e []     = [e]
insert e (x:xs) = e : x : xs          -- overlapping
insert e (x:xs) = x : insert e xs     -- rules

perm []     = []
perm (x:xs) = insert x (perm xs)
```

```
perm [1,2,3]  ⇝  [1,2,3]  |  [1,3,2]  |  [2,1,3]  |  ...
```

= **functional logic languages**

Main result of this paper: **only one of these extensions is sufficient!**

TERM REWRITING SYSTEMS (TRS)

Formal model for functional logic languages:

TERM REWRITING SYSTEMS (TRS)

Formal model for functional logic languages:

Datatypes (≈ admissible values): enumerate all data constructors

```
data Bool    =  True   |  False
data Nat     =  O      |  S Nat
data List    =  []     |  Nat : List
```

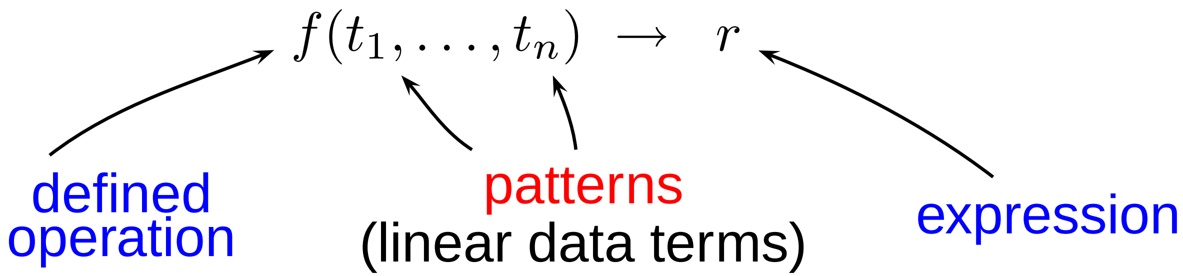
TERM REWRITING SYSTEMS (TRS)

Formal model for functional logic languages:

Datatypes (≈ admissible values): enumerate all data constructors

```
data Bool      =  True     |  False
data Nat       =  O        |  S Nat
data List      =  []       |  Nat : List
```

Rewrite rules: define operations on values



$$f(t_1, \ldots, t_n) \rightarrow r$$

defined operation

patterns
(linear data terms)

expression

```
add(O,y)      →  y                       positive(O)      →  False
add(S(x),y)   →  S(add(x,y))             positive(S(x))   →  True
```

Extra variable: rule variable not occurring in left-hand side

CLASSES OF TERM REWRITING SYSTEMS

CLASSES OF TERM REWRITING SYSTEMS

Inductively sequential term rewriting systems:

- ➜ no overlapping left-hand sides in rules
- ➜ operations inductively defined on datatypes

```
leq(0,x)        → True        cond(True,x) → x
leq(S(x),0)     → False
leq(S(x),S(y))  → leq(x,y)    seven → S(S(S(S(S(S(S(0)))))))
```

CLASSES OF TERM REWRITING SYSTEMS

Inductively sequential term rewriting systems:

- ➜ no overlapping left-hand sides in rules
- ➜ operations inductively defined on datatypes

```
leq(O,x)        → True         cond(True,x) → x
leq(S(x),O)     → False
leq(S(x),S(y))  → leq(x,y)     seven → S(S(S(S(S(S(S(O)))))))
```

ISX: inductively sequential TRS with extra variables

```
smallnum → cond(leq(x,seven),x)
```

CLASSES OF TERM REWRITING SYSTEMS

Inductively sequential term rewriting systems:

- ➜ no overlapping left-hand sides in rules
- ➜ operations inductively defined on datatypes

```
leq(O,x)         →  True        cond(True,x)  →  x
leq(S(x),O)      →  False
leq(S(x),S(y))   →  leq(x,y)    seven  →  S(S(S(S(S(S(S(O)))))))
```

ISX: inductively sequential TRS with extra variables

```
smallnum  →  cond(leq(x,seven),x)
```

OIS: overlapping inductively sequential TRS:

allow rules with multiple right-hand sides: $l \rightarrow r_1 \, ? \cdots ? \, r_k$

```
parent(x)  →  mother(x) ? father(x)
```

CLASSES OF TERM REWRITING SYSTEMS

Inductively sequential term rewriting systems:

- ➜ no overlapping left-hand sides in rules
- ➜ operations inductively defined on datatypes

```
leq(O,x)        →  True        cond(True,x)  →  x
leq(S(x),O)     →  False
leq(S(x),S(y))  →  leq(x,y)    seven  →  S(S(S(S(S(S(S(O)))))))
```

ISX: inductively sequential TRS with extra variables

```
smallnum  →  cond(leq(x,seven),x)
```

OIS: overlapping inductively sequential TRS:
allow rules with multiple right-hand sides: $l \to r_1 \; ? \cdots ? \; r_k$

```
parent(x)  →  mother(x)  ?  father(x)
```

Main result: OIS with rewriting $\Longleftrightarrow$ ISX with narrowing

EVALUATION: REWRITING VS. NARROWING

Functional evaluation: (lazy) rewriting

$\texttt{add(S(0),S(0))} \quad \rightarrow \quad \texttt{S(add(0,S(0)))} \quad \rightarrow \quad \texttt{S(S(0))}$

EVALUATION: REWRITING VS. NARROWING

Functional evaluation: (lazy) rewriting

`add(S(0),S(0))` → `S(add(0,S(0)))` → `S(S(0))`

Functional logic evaluation: narrowing: guess values + rewriting

`add(x,S(0)) =:= S(S(S(0)))` $\leadsto$ $\{$`x`$\mapsto$`S(S(0))`$\}$

EVALUATION: REWRITING VS. NARROWING

Functional evaluation: (lazy) rewriting

$\texttt{add(S(0),S(0))} \quad \rightarrow \quad \texttt{S(add(0,S(0)))} \quad \rightarrow \quad \texttt{S(S(0))}$

Functional logic evaluation: narrowing: guess values + rewriting

$\texttt{add(x,S(0)) =:= S(S(S(0)))} \quad \leadsto \quad \{\texttt{x} \mapsto \texttt{S(S(0))}\}$

Needed narrowing: demand-driven variable instantiation and rewriting

$\texttt{leq(x,add(0,S(0)))} \quad \leadsto_{\{\texttt{x} \mapsto \texttt{0}\}} \quad \texttt{True}$

Sound, complete, optimal evaluation strategy [JACM'00]

ELIMINATING OVERLAPPING RULES

Transformation OE: OIS$\longrightarrow$ISX:

Replace overlapping rule $f(\overline{t_n}) \rightarrow r_1 \; \texttt{?} \cdots \texttt{?} \; r_k$ by:

$f(\overline{t_n}) \rightarrow f'(y, \overline{x_l})$ (where $\mathcal{V}ar(\overline{t_n}) = \{x_1, \ldots, x_l\}$, y fresh, f' new)

$$f'(I_1, \overline{x_l}) \quad \rightarrow \quad r_1$$

$$\vdots$$

$$f'(I_k, \overline{x_l}) \quad \rightarrow \quad r_k$$

`data Ix =` I_1 `|` $\cdots$ `|` I_k (index type, e.g., natural numbers)

ELIMINATING OVERLAPPING RULES

Transformation OE: OIS$\longrightarrow$ISX:

Replace overlapping rule $f(\overline{t_n}) \to r_1 \,?\, \cdots \,?\, r_k$ by:

$f(\overline{t_n}) \to f'(y, \overline{x_l})$ (where $\mathcal{V}ar(\overline{t_n}) = \{x_1, \ldots, x_l\}$, y fresh, f' new)

$$\begin{array}{lll} f'(I_1, \overline{x_l}) & \to & r_1 \\ \vdots & & \\ f'(I_k, \overline{x_l}) & \to & r_k \end{array}$$

`data Ix =` I_1 `|` $\cdots$ `|` I_k (index type, e.g., natural numbers)

Example: `parent(x)` → `mother(x) ? father(x)`

OE ↦

```
data Iparent = I0 | I1

parent(x) → parent'(y,x)
parent'(I0,x) → mother(x)
parent'(I1,x) → father(x)
```

ELIMINATING OVERLAPPING RULES: RESULTS

Proposition: $\mathcal{R} \in \mathrm{OIS} \quad \Rightarrow \quad OE(\mathcal{R}) \in \mathrm{ISX}$

ELIMINATING OVERLAPPING RULES: RESULTS

Proposition: $\mathcal{R} \in \mathrm{OIS} \quad \Rightarrow \quad OE(\mathcal{R}) \in \mathrm{ISX}$

Correctness w.r.t. results computed by needed narrowing:

Theorem: $\mathcal{R} \in \mathrm{OIS}$, $\mathcal{R}' = OE(\mathcal{R})$, t, s terms of $\mathcal{R}$:

Soundness: $t \overset{\text{NN}\,*}{\leadsto}_{\sigma'} s$ w.r.t. $\mathcal{R}'$ $\;\Rightarrow\; \exists\, t \overset{\text{NN}\,*}{\leadsto}_{\sigma} s$ w.r.t. $\mathcal{R}$ with $\sigma =_{\mathcal{V}ar(t)} \sigma'$

Completeness: $t \overset{\text{NN}\,*}{\leadsto}_{\sigma} s$ w.r.t. $\mathcal{R}$ $\;\Rightarrow\; \exists\, t \overset{\text{NN}\,*}{\leadsto}_{\sigma'} s$ w.r.t. $\mathcal{R}'$ with $\sigma =_{\mathcal{V}ar(t)} \sigma'$

ELIMINATING OVERLAPPING RULES: RESULTS

Proposition: $\mathcal{R} \in \mathrm{OIS} \quad \Rightarrow \quad OE(\mathcal{R}) \in \mathrm{ISX}$

Correctness w.r.t. results computed by needed narrowing:

Theorem: $\mathcal{R} \in \mathrm{OIS}$, $\mathcal{R}' = OE(\mathcal{R})$, t, s terms of $\mathcal{R}$:

Soundness: $t \overset{\mathsf{NN}\,*}{\rightsquigarrow}_{\sigma'} s$ w.r.t. $\mathcal{R}' \;\Rightarrow\; \exists\, t \overset{\mathsf{NN}\,*}{\rightsquigarrow}_{\sigma} s$ w.r.t. $\mathcal{R}$ with $\sigma =_{\mathcal{V}ar(t)} \sigma'$

Completeness: $t \overset{\mathsf{NN}\,*}{\rightsquigarrow}_{\sigma} s$ w.r.t. $\mathcal{R} \;\Rightarrow\; \exists\, t \overset{\mathsf{NN}\,*}{\rightsquigarrow}_{\sigma'} s$ w.r.t. $\mathcal{R}'$ with $\sigma =_{\mathcal{V}ar(t)} \sigma'$

$\Rightarrow$ Implementations need not implement overlapping rules
(done in several implementations but without formal justification)

ELIMINATING LOGIC VARIABLES

Transformation XE: ISX$\longrightarrow$OIS$^-$
(extra variable elimination)

OIS$^-$: overlapping inductively sequential *without extra variables*

Evaluation in OIS$^-$: rewriting (not narrowing)

Thus: programs transformed by XE
$\leadsto$ implementation without handling of logic variables and substitutions

ELIMINATING LOGIC VARIABLES

Transformation XE: ISX$\longrightarrow$OIS$^-$
(extra variable elimination)

OIS$^-$: overlapping inductively sequential *without extra variables*

Evaluation in OIS$^-$: rewriting (not narrowing)

Thus: programs transformed by XE
$\leadsto$ implementation without handling of logic variables and substitutions

Basic idea of XE: replace extra variables in rules by value generators

VALUE GENERATORS

S: sort defined by datatype declaration

$$\texttt{data}\ S\ \texttt{=}\ C_1\ t_{11}\ \ldots\ t_{1a_1}\ \texttt{|}\ \ldots\ \texttt{|}\ C_n\ t_{n1}\ \ldots\ t_{na_n}$$

Value generator operation $\texttt{instOf}S$ defined by (overlapping) rules

$$\begin{aligned}\texttt{instOf}S \;\to\; & C_1\texttt{(instOf}t_{11}\texttt{,...,instOf}t_{1a_1}\texttt{)}\\ \texttt{?}\; & \ldots\\ \texttt{?}\; & C_n\texttt{(instOf}t_{n1}\texttt{,...,instOf}t_{na_n}\texttt{)}\end{aligned}$$

VALUE GENERATORS

S: sort defined by datatype declaration

$$\texttt{data}\ S\ \texttt{=}\ C_1\ t_{11}\ \ldots\ t_{1a_1}\ \texttt{|}\ \ldots\ \texttt{|}\ C_n\ t_{n1}\ \ldots\ t_{na_n}$$

Value generator operation $\texttt{instOf}S$ defined by (overlapping) rules

$$\begin{array}{lll} \texttt{instOf}S & \to & C_1\texttt{(instOf}t_{11}\texttt{,}\ldots\texttt{,instOf}t_{1a_1}\texttt{)} \\ & \texttt{?} & \ldots \\ & \texttt{?} & C_n\texttt{(instOf}t_{n1}\texttt{,}\ldots\texttt{,instOf}t_{na_n}\texttt{)} \end{array}$$

Example: `data TreeNat = Leaf | Branch Nat TreeNat TreeNat`

```
instOfTreeNat → Leaf
              ? Branch(instOfNat,instOfTreeNat,instOfTreeNat)

instOfTreeNat → O ? S(instOfNat)
```

VALUE GENERATORS

S: sort defined by datatype declaration

$\texttt{data}\ S\ \texttt{=}\ C_1\ t_{11}\ \ldots\ t_{1a_1}\ \texttt{|}\ \ldots\ \texttt{|}\ C_n\ t_{n1}\ \ldots\ t_{na_n}$

Value generator operation `instOf`S defined by (overlapping) rules

$$\begin{aligned}\texttt{instOf}S &\to C_1\texttt{(instOf}t_{11}\texttt{,}\ldots\texttt{,instOf}t_{1a_1}\texttt{)}\\ &\quad \texttt{?}\ \ldots\\ &\quad \texttt{?}\ C_n\texttt{(instOf}t_{n1}\texttt{,}\ldots\texttt{,instOf}t_{na_n}\texttt{)}\end{aligned}$$

Example: `data TreeNat = Leaf | Branch Nat TreeNat TreeNat`

```
instOfTreeNat → Leaf
              ? Branch(instOfNat,instOfTreeNat,instOfTreeNat)

instOfTreeNat → O ? S(instOfNat)
```

Lemma: ∀ ground constructor terms t of sort S: $\texttt{instOf}S \xrightarrow{*} t$

EXTRA VARIABLE ELIMINATION

Transformation XE: OIS$\longrightarrow$OIS^{-}

replace each extra variable v of sort S in a rule by `instOf`S

EXTRA VARIABLE ELIMINATION

Transformation XE: OIS$\longrightarrow$OIS^{-}

replace each extra variable v of sort S in a rule by `instOf`S

Example: `even` $\rightarrow$ `add(x,x)`

XE $\mapsto$ `even` $\rightarrow$ `add(instOfNat,instOfNat)`

EXTRA VARIABLE ELIMINATION

Transformation XE: OIS$\longrightarrow$OIS^{-}

replace each extra variable v of sort S in a rule by $\texttt{instOf}S$

Example: $\texttt{even} \rightarrow \texttt{add(x,x)}$

$XE \quad \mapsto \quad \texttt{even} \rightarrow \texttt{add(instOfNat,instOfNat)}$

Lemma (Completeness of XE): $t \overset{*}{\rightsquigarrow} u$ w.r.t. $\mathcal{R} \quad \Rightarrow$
$XE(t) \overset{*}{\rightarrow} v$ w.r.t. $XE(\mathcal{R})$ (v: ground constructor instance of u)

EXTRA VARIABLE ELIMINATION

Transformation XE: OIS$\longrightarrow$OIS$^-$

replace each extra variable v of sort S in a rule by `instOf`S

Example: `even` $\rightarrow$ `add(x,x)`

XE $\mapsto$ `even` $\rightarrow$ `add(instOfNat,instOfNat)`

Lemma (Completeness of XE): $t \overset{*}{\leadsto} u$ w.r.t. $\mathcal{R}$ $\Rightarrow$
$XE(t) \xrightarrow{*} v$ w.r.t. $XE(\mathcal{R})$ (v: ground constructor instance of u)

XE is not sound:

`even` $\rightarrow$ `add(instOfNat,instOfNat)` $\xrightarrow{+}$ `add(0,S(0))` $\rightarrow$ `S(0)`

EXTRA VARIABLE ELIMINATION

Transformation XE: OIS$\longrightarrow$OIS$^-$

replace each extra variable v of sort S in a rule by `instOf`S

Example: `even` $\rightarrow$ `add(x,x)`

XE $\mapsto$ `even` $\rightarrow$ `add(instOfNat,instOfNat)`

Lemma (Completeness of XE): $t \overset{*}{\rightsquigarrow} u$ w.r.t. $\mathcal{R}$ $\Rightarrow$
$XE(t) \overset{*}{\rightarrow} v$ w.r.t. $XE(\mathcal{R})$ (v: ground constructor instance of u)

XE is not sound:

`even` $\rightarrow$ `add(instOfNat,instOfNat)` $\overset{+}{\rightarrow}$ `add(0,S(0))` $\rightarrow$ `S(0)`

Solution: identical reductions for extra variable generators

ADMISSIBLE DERIVATIONS

Admissible derivation: apply to occurrences of `instOf` (originating from same extra variable) identical reduction steps

ADMISSIBLE DERIVATIONS

Admissible derivation: apply to occurrences of `instOf` (originating from same extra variable) identical reduction steps

Implementation: use `let` bindings available in many languages

Example: `even` $\to$ `add(x,x)`

XEP ↦ `even` $\to$ `let x = instOfNat in add(x,x)`

ADMISSIBLE DERIVATIONS

Admissible derivation: apply to occurrences of `instOf` (originating from same extra variable) identical reduction steps

Implementation: use `let` bindings available in many languages

Example: `even` $\longrightarrow$ `add(x,x)`

XEP $\mapsto$ `even` $\longrightarrow$ `let x = instOfNat in add(x,x)`

Formalization (details in paper) by transformation XEP:

term/substitution pairs where substitution contains `instOf` operations

ADMISSIBLE DERIVATIONS

Admissible derivation: apply to occurrences of `instOf` (originating from same extra variable) identical reduction steps

Implementation: use `let` bindings available in many languages

Example: `even` $\rightarrow$ `add(x,x)`

XEP $\mapsto$ `even` $\rightarrow$ `let x = instOfNat in add(x,x)`

Formalization (details in paper) by transformation XEP:

term/substitution pairs where substitution contains `instOf` operations

Theorem: transformation XEP is sound and complete

ELIMINATING EXTRA VARIABLES: INITIAL GOALS

Transformation XEP shows:

logic variables can be eliminated if overlapping rules are supported

ELIMINATING EXTRA VARIABLES: INITIAL GOALS

Transformation XEP shows:
logic variables can be eliminated if overlapping rules are supported

Initial goals with logic variables: t with $\mathcal{V}ar(t) = \{x_1, \ldots, x_n\}$

$\leadsto$ start computation with initial term $(t, x_1, \ldots, x_n)$:

Result $(e, b_1, \ldots, b_n)$:

$e \approx$ computed value
$b_1, \ldots, b_n \approx$ computed answer

CONCLUSIONS

Two transformations on functional logic programs:

1. eliminate overlapping rules by extra variables
   - ➜ any functional logic program can be mapped into ISX
   - ➜ ISX $\approx$ core language for functional logic programming
   - ➜ practice: ISX already used as core in some implementations
2. eliminate extra variables by new operations (overlapping rules)
   - ➜ correctness requires admissible derivations
   - ➜ implement admissible derivations by sharing / let expressions

CONCLUSIONS

Two transformations on functional logic programs:

1. eliminate overlapping rules by extra variables
   - ➜ any functional logic program can be mapped into ISX
   - ➜ ISX ≈ core language for functional logic programming
   - ➜ practice: ISX already used as core in some implementations
2. eliminate extra variables by new operations (overlapping rules)
   - ➜ correctness requires admissible derivations
   - ➜ implement admissible derivations by sharing / let expressions

Results useful for

- ➜ better understanding of functional logic programming features
- ➜ simpler core languages (support overlapping rules *or* extra variables)
- ➜ simpler implementations
- ➜ simplify tool support (e.g., current tracers, profilers, slicers, partial evaluators: core language with overlapping rules *and* extra variables)
- ➜ better understanding of the role of logic variables