Functional Logic Languages:

Basic Concepts and Operational Semantics

Michael Hanus

RWTH Aachen

Declarative Programming

- no coding of algorithms
  (how can we solve a problem?)

- description of logical relationships
  (what is the problem?)
  $\Rightarrow$ computer has to find a solution from this description

- mathematical formalisms for the description:
  - *functional languages:*
    functions and equations
  - *logic languages:*
    predicates and logical formulas

- more expressive power, higher programming level

- smaller and more readable programs

- efficient implementation techniques $\sim$ imperative languages

Functional Programming

- program = function: *input* → *output*

  func `fac(N) = if N=0 then 1 else N*fac(N-1)`

- define functions by equations (patterns):

  datatype *nat* = { `0` ; `s`(*nat*) }

  func `fac:` *nat* → *nat*

  ```
  fac(0)     = s(0)
  fac(s(N))  = s(N)*fac(N)
  ```

- computation = evaluation of functional expressions:

  `fac(6)` ⇝ `720`

  `compile(`*`source-program`*`)` ⇝ *`object-code`*

- Execution principle:
  - computation of *values*
  - reduction (function application)
  - deterministic process

Logic Programming

- based on first-order predicate logic
- define relations by Horn clauses (implications)

```
father(joe,bill)
father(bill,tom)
grandfather(X,Z) ⇐ father(X,Y), father(Y,Z)
```

- computation = infer answers from the rules:

  `grandfather(G,tom)` $\rightsquigarrow$ `G=joe`

  `codegen(`*statements*`,C)` $\rightsquigarrow$ `C=`⟨*possible object-code*⟩

- Execution principle:
  - computation of *solutions*
  - resolution (non-deterministic)
  - no input/output arguments

Advantages...

...of functional w.r.t. logic programming:

- higher-order functions:
  - better abstraction
  - better modularization
- efficient evaluation:
  - deterministic reduction
  - lazy evaluation

...of logic w.r.t. functional programming:

- computation with partial information
- search strategies
- more expressive power
  (conditional equations with extra variables)
- applications in operations research and artificial intelligence

(Expected) Advantages of Functional Logic Languages

- expressive power of logic languages
- efficient execution principles of functional languages
- combine applications of both fields

Necessary ingredients:

- logical variables
- constraint solving (unification)
- search (e.g., simulated by backtracking)
- efficient evaluation strategies
- higher-order functions

Equational Logic Programming

- Integration of functional and logic programming languages

- Logic programming:
  predicates and Horn clauses

- Functional programming:
  functions and equations

- Integration:
  - predicates in conditions of equations
  - functional expressions in clauses
  - inverting (solving) functional expressions and equations

Equational Logic Programming

Program:

datatype $nat \ = \ \{\ \mathtt{0} \ ; \ \mathtt{s}(nat)\ \}$

func `+` : $nat, \ nat \ \rightarrow \ nat$

$\mathtt{0 + N} \rightarrow \mathtt{N}$

$\mathtt{s(M) + N} \rightarrow \mathtt{s(M + N)}$

pred $\leq$: $nat, \ nat$

$\mathtt{0} \leq \mathtt{N}$

$\mathtt{s(M)} \leq \mathtt{s(N)} \ \Leftarrow \ \mathtt{M} \leq \mathtt{N}$

Goals:

```
?- A + s(0) = s(s(0))
A = s(0)
```

$\mathtt{?-\ A+B} \leq \mathtt{s(s(0))}$

```
A = 0      B =  0        ;
A = 0      B =  s(0)     ;
A = 0      B =  s(s(0))  ;
A = s(0)   B =  0        ;
...
```

Equational Logic Programming: Semantics

Declarative Semantics:

- Horn clause logic with equality:
  "=" is always interpreted as identity
- In the presence of infinite objects (Babel, K-Leaf):

  Domain-theoretic interpretation (Scott domains):
  "=" is identity on *finite* objects

Operational Semantics:

- Flattening (translation into Prolog)
- Narrowing (reduction with unification)
- Residuation (delay uninstantiated function calls)

Flattening: Translation into Prolog

1. Decompose nested functional expressions into several equations
2. Transform each function into a predicate
3. Apply SLD-resolution to the transformed program

Example: Transform

```
   0 + N → N
s(M) + N → s(M + N)
   0 ≤ N
s(M) ≤ s(N)  ⇐  M ≤ N
?- A+B ≤ s(s(0))
```

into

```
add(0,N,N)
add(s(M),N,s(X))  ⇐  add(M,N,X)
   0 ≤ N
s(M) ≤ s(N)  ⇐  M ≤ N
?- add(A,B,C), C ≤ s(s(0))
```

Flattening

+ easy to implement (preprocessor)

– functions are translated into *nondeterministic* predicate definitions

– loss of information (functional dependencies)

– logic programs are less efficient than equivalent functional programs

Example: Functional `max` program:

$$\begin{array}{lcl} \texttt{max(X,Y)} \rightarrow \texttt{Y} & \Leftarrow & \texttt{X} \leq \texttt{Y} \\ \texttt{max(X,Y)} \rightarrow \texttt{X} & \Leftarrow & \texttt{Y} \leq \texttt{X} \end{array}$$

`?- max(3,3)=Z`

$\leadsto$ *deterministic, one solution*

Example: Relational `max` program:

$$\begin{array}{lcl} \texttt{max(X,Y,Y)} & \Leftarrow & \texttt{X} \leq \texttt{Y} \\ \texttt{max(X,Y,X)} & \Leftarrow & \texttt{Y} \leq \texttt{X} \end{array}$$

`?- max(3,3,Z)`

$\leadsto$ *nondeterministic, two (identical) solutions*

Execution Principle: Narrowing

(ALF, Babel, K-Leaf, LPG, SLOG,...)

Compute with functional expressions (as in functional programming) but replace *matching* by *unification*

Definition of the narrowing relation:

1. p non-variable position in equation e
2. $l \to r$ function rule (with new variables)
3. σ unifier for l and subterm $e|_p$

$\Longrightarrow$ e narrows to $\sigma(e[r]_p)$

Example: Addition of natural numbers:

```
   0+X  →  X
s(X)+Y  →  s(X+Y)
```

$\texttt{Z+s(0) = s(s(0))} \leadsto_{\{\texttt{Z}\mapsto\texttt{s(X)}\}} \texttt{s(X+s(0)) = s(s(0))}$

$\leadsto_{\{\texttt{X}\mapsto\texttt{0}\}} \texttt{s(s(0)) = s(s(0))}$

Computed answer: Z↦s(0)

Narrowing: Properties

- complete (w.r.t. normalized answers) if function rules are confluent

- *each* rule must be unified with *each* non-variable subterm

- huge and infinite search space even for simple programs

- more nondeterminism than equivalent pure logic programs

$\Longrightarrow$ reduce possible narrowing derivations

Confluence

- distinct reduction sequences leads to a same result
- ensures well-definedness of functions
- if the rewrite system is *terminating* (i.e., no infinite reduction derivations): decidable by checking joinability of critical pairs
- if the rewrite system is not terminating: syntactic criteria to ensure confluence:

  1. left-linearity: no multiple variable occurrences in left-hand sides of rules
  2. free variables: $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ for all $l \to r \Leftarrow c$
  3. non-ambiguity: for all $l_i \to r_i \Leftarrow c_i$ $(i = 1, 2)$
     (a) l_1 and l_2 not unifiable, or
     (b) $\sigma(r_1) = \sigma(r_2)$ for $\sigma = mgu(l_1, l_2)$, or
     (c) c_1 and c_2 "syntactically incompatible" (e.g., $p \in c_1$ and $(\neg p) \in c_2$)

Basic Narrowing

Narrowing frequently generates infinite derivations

$$\texttt{rev(rev(L))} \to \texttt{L}$$

$$\begin{aligned}\texttt{rev(X)} = \texttt{X} \quad &\leadsto_{\{\texttt{X}\mapsto\texttt{rev(X1)}\}} \quad \texttt{X1} = \texttt{rev(X1)} \\ &\leadsto_{\{\texttt{X1}\mapsto\texttt{rev(X2)}\}} \quad \texttt{rev(X2)} = \texttt{X2} \\ &\ldots\end{aligned}$$

`rev(X1)` was introduced by unification

$\Longrightarrow$ forbid narrowing steps in instances of variables

$\Longrightarrow$ **Basic Narrowing** [Hullot 80]

- complete for confluent and terminating systems
- finite search space in this example
- further restrictions:
  - left-to-right basic narrowing [Herold 86]
  - LSE narrowing [Bockmayr et al. 92]

Narrowing Strategies

(Basic) narrowing:
set of positions for next narrowing step

Reduction in functional languages:
a *unique position* is selected for next reduction step:

- *call-by-value*: innermost position
- *call-by-name*: outermost position

Narrowing strategy:
selection of a position for the next narrowing step

Implementation: **constructor-based programs**:

- distinguish between constructors and defined functions
- $f(t_1, \ldots, t_n) \to r$ must satisfy:
  f defined function and t_i constructor terms
- required in functional programming

Innermost Narrowing

- select innermost defined function symbol for next narrowing step
- corresponds to *eager* (*call-by-value*) evaluation
- efficiently implementable
- complete for totally defined functions [Fribourg 85]

$$\begin{array}{rcl} \texttt{f(a,Z)} & \rightarrow & \texttt{a} \\ \texttt{g(b)} & \rightarrow & \texttt{b} \end{array}$$

successful (general) narrowing derivation:

$\texttt{f(X,g(X))=a} \quad \leadsto_{\{\texttt{X}\mapsto\texttt{a}\}} \quad \texttt{a=a}$

exactly one innermost narrowing derivation:

$\texttt{f(X,g(X))=a} \quad \leadsto_{\{\texttt{X}\mapsto\texttt{b}\}} \quad \texttt{f(b,b)=a}$

Solution: "skip" innermost function calls
(**innermost basic narrowing** [Hölldobler 89])

$\leadsto$ additional non-determinism

Innermost Narrowing: Problems

- infinite narrowing derivations even for terminating programs
- infinite computations at innermost positions

```
   0 + N → N
s(M) + N → s(M + N)
```

$$\texttt{(X+Y)+Z = 0}$$

$$\leadsto_{\{\texttt{X}\mapsto\texttt{s(X1)}\}} \quad \texttt{s(X1+Y)+Z = 0}$$

$$\leadsto_{\{\texttt{X1}\mapsto\texttt{s(X2)}\}} \quad \texttt{s(s(X2+Y))+Z = 0}$$

$$\leadsto_{\{\texttt{X2}\mapsto\texttt{s(X3)}\}} \quad \texttt{s(s(s(X3+Y)))+Z = 0}$$

...

infinite search space!

Normalizing Innermost Narrowing

Reduce terms to *normal form*
(application of reduction steps)
before each narrowing step

[Fay 79, Fribourg 85]

$$\texttt{0 + N} \rightarrow \texttt{N}$$
$$\texttt{s(M) + N} \rightarrow \texttt{s(M + N)}$$

`(X+Y)+Z = 0`

Terms are in normal form: narrowing step

$\leadsto_{\{\texttt{X}\mapsto\texttt{s(X1)}\}}$ `s(X1+Y)+Z = 0`

Reduce left-hand side to normal form:

$\rightarrow$ `s((X1+Y)+Z) = 0`

Fail: different constructors 0 *and* s

Normalizing Innermost Narrowing

- normalization is a deterministic process
  (uniqueness of normal forms)

- priority for deterministic computations
  $\Longrightarrow$ less time and space for execution

- search space reduction by detecting useless narrowing derivations

- innermost narrowing $\approx$ SLD-resolution
  [Bosco et al. 88]
  $\Longrightarrow$ normalizing innermost narrowing has smaller search space than SLD-resolution

- functional logic programs are more efficient than pure logic programs [Fribourg 85, Hanus 92]

- efficiently implementable on conventional hardware
  (*innermost strategy*)
  (ALF System [Hanus 90,91,92])

Normalizing Innermost Narrowing $\Updownarrow$ SLD-Resolution

$$\begin{aligned} \mathtt{0 + N} &\to \mathtt{N} \\ \mathtt{s(M) + N} &\to \mathtt{s(M + N)} \end{aligned}$$

$$\begin{aligned} \mathtt{(X+Y)+Z = 0} \quad \rightsquigarrow_{\{\mathtt{X}\mapsto\mathtt{s(X1)}\}} \quad & \mathtt{s(X1+Y)+Z = 0} \\ \to \quad & \mathtt{s((X1+Y)+Z) = 0} \end{aligned}$$

Failure!

Equivalent (flattened) logic program:

```
add(0,N,N)
add(s(M),N,s(Z))  ⇐  add(M,N,Z)
```

$\mathtt{add(X,Y,R),\ add(R,Z,0)}$

$\rightsquigarrow_{\{\mathtt{X}\mapsto\mathtt{s(X1)},\mathtt{R}\mapsto\mathtt{s(R1)}\}}$ $\mathtt{add(X1,Y,R1),add(s(R1),Z,0)}$

$\rightsquigarrow_{\{\mathtt{X1}\mapsto\mathtt{s(X2)},\mathtt{R1}\mapsto\mathtt{s(R2)}\}}$ $\mathtt{add(X2,Y,R2),}$
$\mathtt{add(s(s(R1)),Z,0)}$

...

Infinite search space!

Termination

Requirement for eager narrowing strategies (basic, innermost, normalizing narrowing):

Function rules must be *terminating*

- undecidable property of programs
- does not permit infinite data structures

```
            from(N)  →  [N|from(s(N))]
         first(0,L)  →  []
first(s(N),[E|L])  →  [E|first(N,L)]
```

`from(0)` $\rightarrow^*$ `[0,s(0),s(s(0)),s(s(s(0))),...`

`first(s(s(0)),from(0))` $\rightarrow^*$ `[0,s(0)]`

Useful programming technique in non-strict functional languages (*stream-oriented programming*)

Programs with non-terminating rules:

- require *lazy evaluation*:
  compute expressions only if demanded
- require *strict equality*

Strict Equality

Problems with equality in the presence of non-terminating rules:

1. Equality on infinite objects undecidable:

   `f` $\rightarrow$ `[0|f]` `g` $\rightarrow$ `[0|g]`

   Is `f=g` valid?

2. Semantics of non-terminating functions:

   `f(X)` $\rightarrow$ `f(s(X))` `g(X)` $\rightarrow$ `g(s(X))`

   Is `f(0)=g(0)` valid?

Avoided by **strict equality**:
"$\equiv$" is identity on *finite* objects
(both sides reducible to same ground constructor term)

Definable by rewrite rules:
(c and d are 0- and n-ary constructors):

$$
\begin{aligned}
c \equiv c &\rightarrow \texttt{true} \\
d(x_1, \ldots, x_n) \equiv d(y_1, \ldots, y_n) &\rightarrow x_1 \equiv y_1 \wedge \ldots \wedge x_n \equiv y_n \\
\texttt{true} \wedge x &\rightarrow x
\end{aligned}
$$

Lazy Narrowing

- apply narrowing steps at outermost positions
- evaluate arguments if necessary

$$\texttt{0*X} \quad \to \quad \texttt{0}$$

$$\texttt{0*(s(0)+s(z))} \equiv \texttt{0} \quad \leadsto_{\{\}} \quad \texttt{0} \equiv \texttt{0}$$

Permits computations with infinite data structures:

$$\begin{aligned}
\texttt{from(N)} &\quad\to\quad \texttt{[N|from(s(N))]}\\
\texttt{first(0,L)} &\quad\to\quad \texttt{[]}\\
\texttt{first(s(N),[E|L])} &\quad\to\quad \texttt{[E|first(N,L)]}
\end{aligned}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{first(X,from(Y))} \equiv \texttt{[0]}}\\
\leadsto_{\{\}} & \texttt{first(X,[Y|from(s(Y))])} \equiv \texttt{[0]}\\
\leadsto_{\{\texttt{X}\mapsto\texttt{s(X1)}\}} & \texttt{[Y|first(X1,from(s(Y)))]} \equiv \texttt{[0]}\\
\leadsto_{\{\texttt{Y}\mapsto\texttt{0}\}} & \texttt{first(X1,from(s(Y)))} \equiv \texttt{[]}\\
\leadsto_{\{\texttt{X1}\mapsto\texttt{0}\}} & \texttt{[]} \equiv \texttt{[]}
\end{array}$$

Computed answer: $\{\texttt{X} \mapsto \texttt{s(0)}, \texttt{Y} \mapsto \texttt{0}\}$

Eager narrowing strategies run into infinite loop!

Lazy Narrowing is not Lazy

- redex selection depends on rule selection
- interaction with variable instantiation

$$
\begin{array}{rlrl}
\mathtt{0} \leq \mathtt{X} & \rightarrow \mathtt{true} & \mathtt{0+X} & \rightarrow \mathtt{X} \\
\mathtt{s(X)} \leq \mathtt{0} & \rightarrow \mathtt{false} & \mathtt{s(X)+Y} & \rightarrow \mathtt{s(X+Y)} \\
\mathtt{s(X)} \leq \mathtt{s(Y)} & \rightarrow \mathtt{X} \leq \mathtt{Y} & &
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{X} \leq \mathtt{Y+Z} & \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} & \mathtt{true} \\
 & & \\
\mathtt{X} \leq \mathtt{Y+Z} & \leadsto_{\{\mathtt{Y}\mapsto\mathtt{0}\}} & \mathtt{X} \leq \mathtt{Z} \\
 & \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(X1)},\mathtt{Z}\mapsto\mathtt{0}\}} & \mathtt{false} \\
 & & \\
\mathtt{X} \leq \mathtt{Y+Z} & \leadsto_{\{\mathtt{Y}\mapsto\mathtt{0}\}} & \mathtt{X} \leq \mathtt{Z} \\
 & \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} & \mathtt{true}
\end{array}
$$

First step is superfluous in the last derivation!

Solution:

1. Bind X to a constructor term
2. Decide which step is needed for this binding

$\Longrightarrow$ Needed narrowing [Antoy/Echahed/Hanus 94]

Needed Narrowing Steps

Idea: **Evaluate only needed subterms**

1. Determine some needed argument position (from the left-hand sides of the rules)
2. If actual argument is a
   - constructor: proceed with matching
   - function call: evaluate it
   - logical variable: (nondeterministically) instantiate it

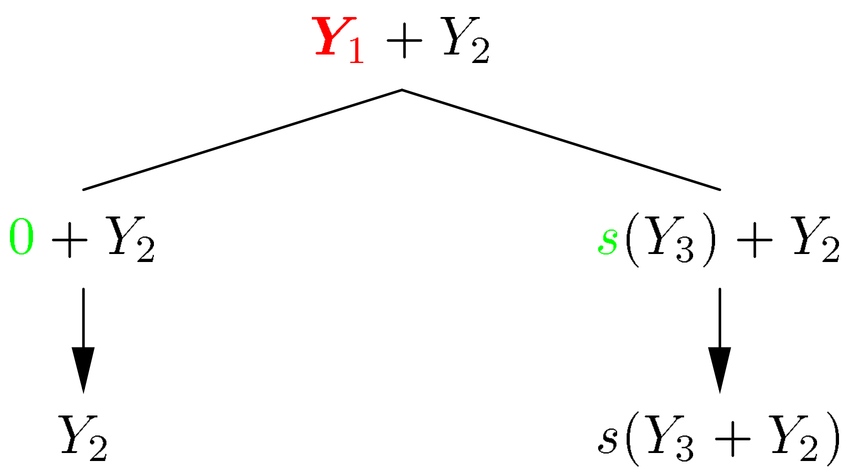
$$\begin{array}{rcl} \mathtt{0} \leq \mathtt{X} & \to & \mathtt{true} \\ \mathtt{s(X)} \leq \mathtt{0} & \to & \mathtt{false} \\ \mathtt{s(X)} \leq \mathtt{s(Y)} & \to & \mathtt{X} \leq \mathtt{Y} \end{array} \qquad \begin{array}{rcl} \mathtt{0+X} & \to & \mathtt{X} \\ \mathtt{s(X)+Y} & \to & \mathtt{s(X+Y)} \end{array}$$

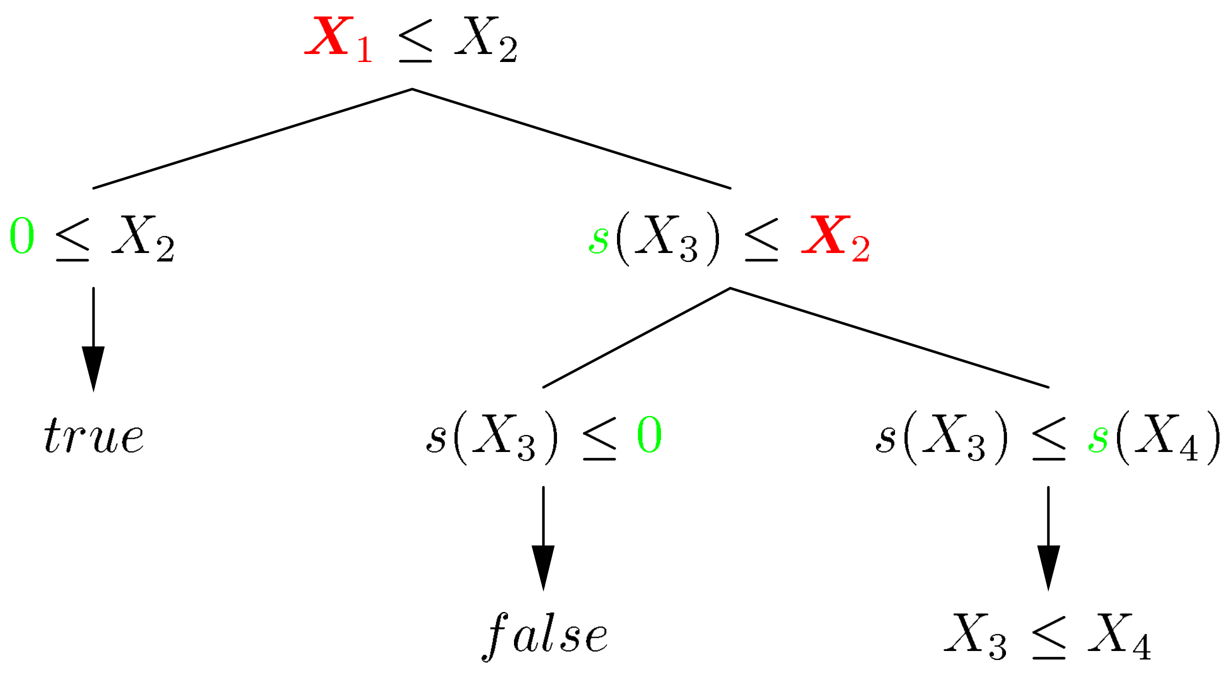
$$\begin{array}{rlll} \mathtt{X} \leq \mathtt{Y+Z} & \mapsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} & \mathtt{0} \leq \mathtt{Y+Z} & \to \quad \mathtt{true} \\ \\ \mathtt{X} \leq \mathtt{Y+Z} & \mapsto_{\{\mathtt{X}\mapsto\mathtt{s(X1)}\}} & \mathtt{s(X1)} \leq \mathtt{Y+Z} & \\ & \mapsto_{\{\mathtt{Y}\mapsto\mathtt{0}\}} & \mathtt{s(X1)} \leq \mathtt{0+Z} & \to \\ \mathtt{s(X1)} \leq \mathtt{Z} & \mapsto_{\{\mathtt{Z}\mapsto\mathtt{0}\}} & \mathtt{s(X1)} \leq \mathtt{0} & \to \quad \mathtt{false} \end{array}$$

Crucial: do not compute *most general* unifiers!

Definitional Trees [Antoy 92]

- data structure to organize the rules of an operation
- each node has a distinct *pattern*
- *branch* nodes (case distinction), *rule* nodes





Properties of Needed Narrowing

[Antoy/Echahed/Hanus POPL'94]

- **Sound** and **complete** w.r.t. strict equality
- **Optimality:**
  1. **No unnecessary steps:**
     Each narrowing step is needed, i.e., it cannot be avoided if a solution should be computed.
  2. **Shortest derivations:**
     If common subterms are shared, needed narrowing derivations have minimal length.
  3. **Independence of solutions:**
     Two solutions σ and σ' computed by two distinct derivations are independent (i.e., $\sigma(x)$ and $\sigma'(x)$ are not unifiable for some variable x).
- **Determinism:**
  No nondeterministic step during the evaluation of ground expressions
- **Restriction:** Needed narrowing defined for
  Inductively Sequential Rewrite Systems:
  The rules of each operation can be organized in a *definitional tree*
  (in particular, non-overlapping left-hand sides)

Overlapping Rules: Problems

$$\begin{array}{rcl} \texttt{true} \vee \texttt{X} & \rightarrow & \texttt{true} \\ \texttt{X} \vee \texttt{true} & \rightarrow & \texttt{true} \\ \texttt{false} \vee \texttt{false} & \rightarrow & \texttt{false} \end{array} \qquad \begin{array}{rcl} \texttt{ev(0)} & \rightarrow & \texttt{true} \\ \texttt{ev(s(0))} & \rightarrow & \texttt{false} \\ \texttt{ev(s(s(N)))} & \rightarrow & \texttt{ev(N)} \end{array}$$

- no efficient strategy:

  Evaluate $t_1 \vee t_2$: evaluate t_1 or evaluate t_2?

- "demanded" arguments depend on applied rule

- unnecessary evaluation due to "wrong" rule

$$\texttt{true} \vee \texttt{ev(X)} \leadsto_{\{\}} \texttt{true}$$

$$\begin{array}{lll} \texttt{true} \vee \texttt{ev(X)} & \leadsto_{\{\texttt{X}\mapsto\texttt{s(s(X1))}\}} & \texttt{true} \vee \texttt{ev(X1)} \\ & \leadsto_{\{\texttt{X1}\mapsto\texttt{s(s(X2))}\}} & \texttt{true} \vee \texttt{ev(X2)} \\ & \leadsto_{\{\texttt{X2}\mapsto\texttt{s(s(X3))}\}} & \texttt{true} \vee \texttt{ev(X3)} \\ & \ldots & \end{array}$$

$\boxed{\text{Infinite search space!}}$

Dynamic Cut

[Loogen/Winkler PLILP'91]

- Ignore alternative rules if no goal variable has been bound during rule application
- Implementation: check bound variables (trail) and (conditionally) delete choice point
  $\Rightarrow$ *dynamic* cut

$$
\begin{array}{rcl}
\texttt{true} \vee \texttt{X} & \to & \texttt{true} \\
\texttt{X} \vee \texttt{true} & \to & \texttt{true} \\
\texttt{false} \vee \texttt{false} & \to & \texttt{false}
\end{array}
\qquad
\begin{array}{rcl}
\texttt{ev(0)} & \to & \texttt{true} \\
\texttt{ev(s(0))} & \to & \texttt{false} \\
\texttt{ev(s(s(N)))} & \to & \texttt{ev(N)}
\end{array}
$$

$$\texttt{true} \vee \texttt{ev(X)} \quad \rightsquigarrow_{\{\}} \quad \texttt{true}$$

(delete choice point)

No alternative rule applicable $\rightsquigarrow$ finite search space!

Disadvantage: depends on the order of rules:

$$
\begin{array}{ll}
\texttt{ev(X)} \vee \texttt{true} & \rightsquigarrow_{\{\texttt{X}\mapsto\texttt{s(s(X1))}\}} \quad \texttt{ev(X1)} \vee \texttt{true} \\
& \rightsquigarrow_{\{\texttt{X1}\mapsto\texttt{s(s(X2))}\}} \quad \texttt{ev(X2)} \vee \texttt{true} \\
& \ldots
\end{array}
$$

Lazy Narrowing with Simplification

- combine advantage of normalizing and lazy narrowing
- apply deterministic reduction steps before nondeterministic narrowing steps
- use a terminating subset of the rules for simplification
- completeness similar to lazy narrowing [Hanus PLILP'94]

$$
\begin{array}{rcl@{\qquad}rcl}
\texttt{true} \vee \texttt{X} & \to & \texttt{true} & \texttt{ev(0)} & \to & \texttt{true} \\
\texttt{X} \vee \texttt{true} & \to & \texttt{true} & \texttt{ev(s(0))} & \to & \texttt{false} \\
\texttt{false} \vee \texttt{false} & \to & \texttt{false} & \texttt{ev(s(s(N)))} & \to & \texttt{ev(N)}
\end{array}
$$

$$\texttt{ev(X)} \ \vee \ \texttt{true} \quad \to \quad \texttt{true}$$

only one derivation possible $\leadsto$ finite search space!

Preference to deterministic reductions

Overlapping Rules: Problems

$$\begin{array}{rcl} \texttt{true} \vee \texttt{X} & \to & \texttt{true} \\ \texttt{X} \vee \texttt{true} & \to & \texttt{true} \\ \texttt{false} \vee \texttt{false} & \to & \texttt{false} \end{array}$$

Evaluate `f(X)` $\vee$ `g(X)`: narrow `f(X)` or narrow `g(X)`?

Solution in current languages:
Backtracking: narrow `f(X)`, if not successful:
narrow `g(X)`

Disadvantage:

- unnecessary evaluations
- incompleteness because of infinite narrowing in `f(X)`
- non-determinism even for ground terms!
  (Backtracking on ground terms:
  Functional programming?)

Alternative solution:
Apply evaluation steps to `f(X)` and `g(X)` in parallel

Parallel Narrowing Steps

1. Compute all narrowing substitutions (for all overlapping cases)
2. Minimize substitution set
3. Perform parallel rewriting for each substitution

$$\begin{array}{rclcl} \texttt{true} & \vee & \texttt{X} & \to & \texttt{true} \\ \texttt{X} & \vee & \texttt{true} & \to & \texttt{true} \\ \texttt{false} & \vee & \texttt{false} & \to & \texttt{false} \end{array} \qquad \texttt{f(a)} \to \texttt{true}$$

$\texttt{f(X)} \vee \texttt{f(X)}$:

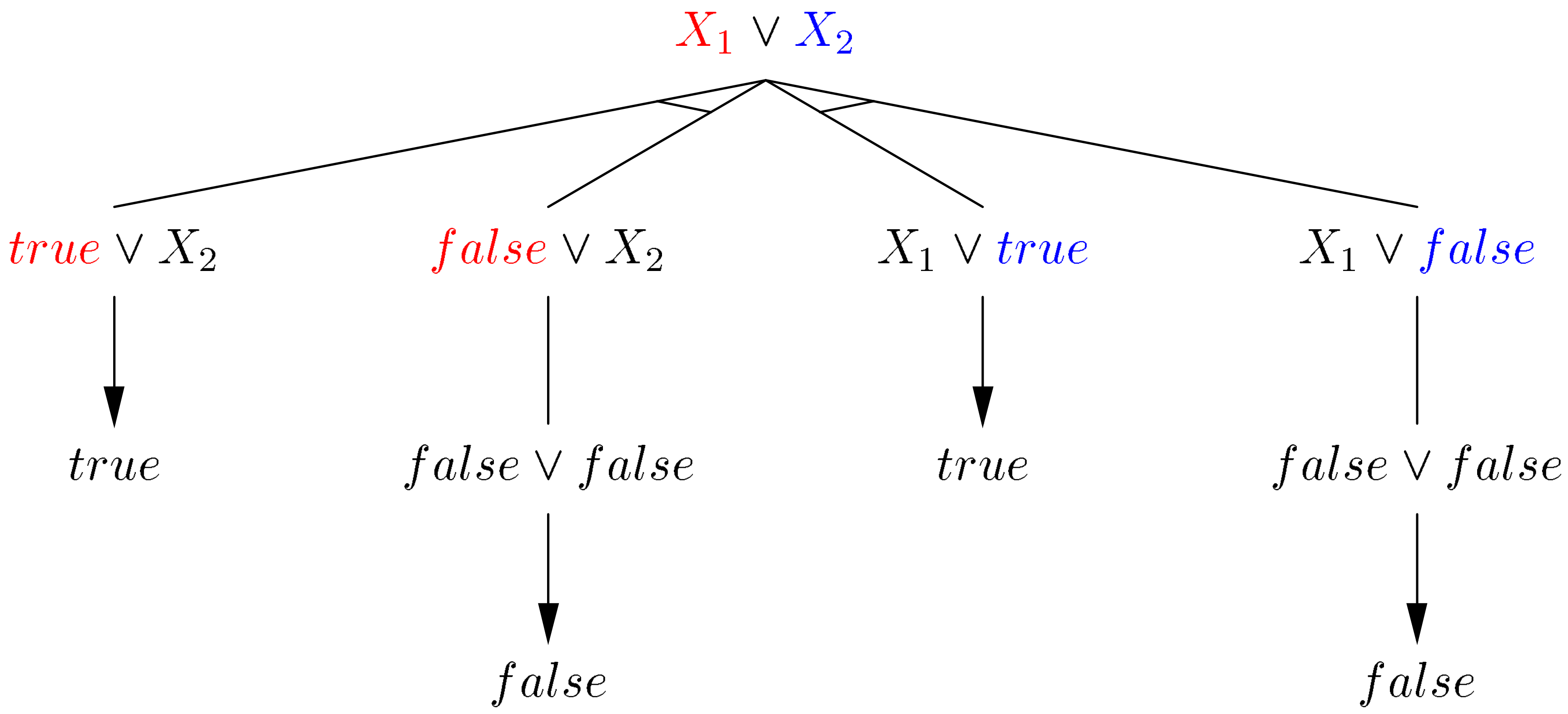
1. Computed substitutions: $\{\texttt{X} \mapsto \texttt{a}\}$ and $\{\texttt{X} \mapsto \texttt{a}\}$
2. Minimized substitution set: $\{\ \{\texttt{X} \mapsto \texttt{a}\}\ \}$
3. Apply substitution + parallel rewriting:

   $\texttt{f(X)} \vee \texttt{f(X)} \;\; \approx\!\!\!>_{\{\texttt{X}\mapsto\texttt{a}\}} \;\; \texttt{true} \vee \texttt{true} \;\; \approx\!\!\!>_{\{\}} \;\; \texttt{true}$

Standard lazy narrowing computes four distinct derivations!

Parallel Definitional Trees

May contain branch nodes with more than one branching argument:



Parallel Narrowing: Minimize Substitutions

- Prefer substitutions belonging to outer subterms
- Take minimal substitutions among the non-identity computed substitutions

$$
\begin{array}{rclcrcl}
\texttt{f(X,s(Y))} & \to & \texttt{s(0)} & \quad & \texttt{h(s(s(X)))} & \to & \texttt{s(0)} \\
\texttt{f(0,Y)} & \to & \texttt{s(0)} & & \texttt{g(X)} & \to & \texttt{s(g(X))}
\end{array}
$$

`f(g(X),f(h(Y),f(0,h(s(Y))))):`

1. Computed position/substitution pairs:
   $\{(1, \{\}), (2 \cdot 1, \{\texttt{Y} \mapsto \texttt{s(s(Y1))}\}), (2 \cdot 2, \{\}),$
   $(2 \cdot 2 \cdot 2, \{\texttt{Y} \mapsto \texttt{s(Y2)}\})\}$
2. Minimized substitution set: $\{ \; \{\} \; \}$

Consider non-identity substitutions in the presence of non-terminating rules:

$$
\begin{array}{rclcrcl}
\texttt{f(X,0)} & \to & \texttt{s(0)} & \quad & \texttt{h(0)} & \to & \texttt{0} \\
\texttt{f(0,X)} & \to & \texttt{s(0)} & & \texttt{g(X)} & \to & \texttt{g(s(X))}
\end{array}
$$

$\texttt{f(g(X),h(Y))} \;\; \approx\!\!\!>_{\{\texttt{Y}\mapsto\texttt{0}\}} \;\; \texttt{f(g(s(X)),0)} \;\; \approx\!\!\!>_{\{\}} \;\; \texttt{0}$

Properties of Parallel Narrowing

[Antoy/Echahed/Hanus 96]

- **Soundness** and **Completeness**
  w.r.t. strict equality

- **Deterministic evaluation** of ground terms

- Computes always a normal form if it exists

- **Optimality:**

  – Equivalent to *needed narrowing* if the occurring functions are inductively sequential
  ($\approx$ optimal on non-overlapping parts)

  – Equivalent to *rewriting with necessary sets* [Sekar/Ramakrishnan 93] on ground terms
  ($\approx$ optimal if rhs are not considered)

Narrowing Strategies

- Completeness w.r.t. standard or strict equality
- Extension of rewrite strategies to functional logic programs
- Optimality results from term rewriting
- Function inversion: guess values for unknown arguments (driven by left-hand sides of rules)

  $\leadsto$ Function evaluation may cause non-determinism

Alternative evaluation principle:

- Evaluate functions always in a deterministic way
- Express non-determinism by predicates or disjunctions
- Delay insufficiently instantiated function calls

  $\Longrightarrow$ **Residuation**

Residuation

[Aït-Kaci et al. 87]

(Escher, Le Fun, Life, NUE-Prolog, Oz,...)

Delay functions calls until deterministic reduction is possible

```
  0+X → X               nat(0)
s(X)+Y → s(X+Y)         nat(s(X)) ⇐ nat(X)
```

$$
\begin{array}{ll}
 & \texttt{X+0 = s(0), nat(X)} \\
\leadsto_{\{\texttt{X}\mapsto\texttt{s(X1)}\}} & \texttt{s(X1)+0 = s(0), nat(X1)} \\
\leadsto_{\{\}} & \texttt{s(X1+0) = s(0), nat(X1)} \\
\leadsto_{\{\texttt{X1}\mapsto\texttt{0}\}} & \texttt{s(0+0) = s(0)} \\
\leadsto_{\{\}} & \texttt{s(0) = s(0)}
\end{array}
$$

Computed answer: $\{\texttt{X} \mapsto \texttt{s(0)}\}$

Functions: Deterministic evaluation

Predicates: Nondeterministic evaluation

(in Escher [Lloyd 94]: mode declarations)

Properties of Residuation

- preserves the deterministic nature of functions
- **incomplete**:
  compute the last element `E` of a list `L`:
  solve equation `append(_,[E])=L` $\leadsto$ *fail*
- Residuation may have an infinite search space in contrast to narrowing:

Reversing a list:

```
rev([],[])
rev(L,[E|LR]) ⇐ append(LE,[E])=L, rev(LE,LR)
?- rev([a,b,c],R)
```

$\Rightarrow$ infinite derivation by applying the second rule

[Hanus JLP'95]: conditions for the completeness of residuation based on program analysis techniques (abstract interpretation)

- Which residuations may occur at run time?
- Which variables are bound in these residuations?

Residuation vs. Narrowing

Residuation:

+ deterministic evaluation of functions
+ provide concurrent computations
+ simple connection of external functions (arithmetic,. . .)

– incomplete (no solution if functions not evaluable)

– unclear relation to Prolog's resolution strategy (infinite residuation derivations)

Narrowing:

+ completeness w.r.t. functional and logic programming
+ clear relation to Prolog's resolution strategy (always better or at least equal)

– expensive function inversion (nondeterminism)

– infinite loops

– combination with external functions unclear

Summary

Three kinds of evaluation strategies:

- **Eager narrowing** (basic, innermost,...)
  - require terminating rules
  - complete for standard equality
  - application areas: theorem proving, specification languages
- **Lazy narrowing** (needed, parallel,...)
  - syntactic criteria to ensure confluence
  - complete for strict equality
  - optimal for inductively sequential systems
- **Residuation**
  - incomplete
  - allows connection of external functions

Conclusions

Why integration of declarative paradigms?

- more expressive than pure functional languages (compute with partial information)
- more structural information than in pure logic programs (functional dependencies)
- more efficient than logic programs (determinism, laziness)
- functions: declarative notion to improve control in logic programming
- avoid impure features of Prolog

Future of Functional Logic Programming

Design and implement a declarative language with

- operational semantics:
  combination of narrowing and residuation
- types (parametric polymorphism)
- modules
- higher-order features (higher-order unification?)
- declarative (monadic) I/O
- encapsulated search (avoid global search)
- constraints

$\leadsto$ Join research efforts in functional and logic programming

$\leadsto$ Do not teach two paradigms, but one:

Declarative Programming