ILPS'97

Multi-Paradigm Declarative Programming

Michael Hanus

RWTH Aachen

Declarative Programming

- no coding of algorithms

- description of logical relationships

- more expressive power, higher programming level

- reliable and maintainable programs
  - pointer structures $\Rightarrow$ algebraic data types
  - complex procedures $\Rightarrow$ comprehensible parts (pattern matching, local definitions)

- *Functional programming:*
  – functions and equations
  – deterministic reduction

- *Logic programming:*
  – predicates and logical formulas
  – non-deterministic search for solutions

$\Rightarrow$ **Functional logic languages:**

- expressive power of logic languages

- efficient execution principles of functional languages

Problems with Prolog

- uncontrolled non-determinism:
  inefficient, infinite loops
- pruning operators: non-declarative (red cuts)
- arithmetic (partial predicates!)
- non-declarative I/O: backtracking?

Integration of functions:

- better efficiency (determinism!)
- new programming techniques:
  higher-order functions, laziness, infinite data structures
- declarative I/O, clean arithmetic
- no burden on the programmer:
  most predicates are functions!

Functional logic languages
can avoid impure features of Prolog

First Approach: 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

data nat = 0 | s(nat)

func + : nat, nat $\rightarrow$ nat

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

pred $\leq$: nat, nat

0 $\leq$ N

s(M) $\leq$ s(N) $\Leftarrow$ M $\leq$ N

Goals:

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

?- A+B $\leq$ 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 (strict equality)

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 *non-deterministic* predicate definitions
– loss of information (functional dependencies)
– logic programs are less efficient than equivalent functional programs

Example: Functional `max` program:

$$\texttt{max(X,Y) = Y} \quad \Leftarrow \quad \texttt{X} \leq \texttt{Y}$$
$$\texttt{max(X,Y) = X} \quad \Leftarrow \quad \texttt{Y} \leq \texttt{X}$$

`max(3,3)=Z`

$\leadsto$ *deterministic, one solution*

Example: Relational `max` program:

$$\texttt{max(X,Y,Y)} \quad \Leftarrow \quad \texttt{X} \leq \texttt{Y}$$
$$\texttt{max(X,Y,X)} \quad \Leftarrow \quad \texttt{Y} \leq \texttt{X}$$

`max(3,3,Z)`

$\leadsto$ *non-deterministic, 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. $l = r$ function rule (with new variables)
3. σ (*most general*) unifier for l and subterm $e|_p$

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

Example: Addition of natural numbers:

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

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

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

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

Narrowing: Properties

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

- confluence requirement can be dropped [Gonzales et al. ESOP'96, Antoy ALP'97]

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

- huge and infinite search space even for simple programs

- more non-determinism than equivalent pure logic programs

$\Longrightarrow$ reduce possible narrowing derivations

Confluence

- distinct reduction sequences lead 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 $l = r \Leftarrow c$
  3. non-ambiguity: for all $l_i = r_i \Leftarrow c_i$ $(i = 1, 2)$
     (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) $\sigma(c_1) \wedge \sigma(c_2)$ unsatisfiable for $\sigma = mgu(l_1, l_2)$

Narrowing Strategies [Hanus JLP'94]

simple narrowing [Hullot80,Slagle74]

basic narrowing [Hullot80]

left-to-right basic narrowing [Herold86]

LSE-narrowing [Bockmayr/Krischer/Werner92]

innermost narrowing [Fribourg85]

innermost basic narrowing [Hölldobler89]

selection narrowing [Bosco/Giovannetti/Moiso88]

normalizing narrowing [Fay79]

normalizing innermost narrowing [Fribourg85]

normalizing basic narrowing [Nutt/Réty/Smolka89,Réty87]

normalizing innermost basic narrowing [Hölldobler89]

outermost narrowing [Echahed88]

outer narrowing [You89]

lazy narrowing [Darlington/Guo89,Moreno et al.90,Reddy85]

needed narrowing [Antoy/Echahed/Hanus94]

parallel narrowing [Antoy/Echahed/Hanus97]

simple conditional narrowing [Hussmann85,Kaplan84]

basic conditional narrowing [Middeldorp/Hamoen92]

innermost basic conditional narrowing [Hölldobler89]

Narrowing Strategies

Simple narrowing:
set of possible positions for next narrowing step

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

- *call-by-value* (*eager*): innermost position
- *call-by-name* (*lazy*): outermost position

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

Implementation: **constructor-based programs**:

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

$\leadsto$ **functional logic programs**

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]

```
f(a,Z) = a
  g(b) = b
```

successful (general) narrowing derivation:

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

exactly one innermost narrowing derivation:

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

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)
```

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

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

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

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

...

infinite search space!

Normalizing Narrowing

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

[Fay 79, Fribourg 85]

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

`(X+Y)+Z = 0`

Terms are in normal form: narrowing step

$\leadsto_{\{\mathtt{X} \mapsto \mathtt{s(X1)}\}}$ `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 Narrowing: Properties

- 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

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

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

Failure!

Equivalent (flattened) logic program:

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

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

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

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

$\texttt{add(s(s(R1)),Z,0)}$

...

Infinite search space!

Termination

Requirement for eager narrowing strategies (innermost, basic, 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 = [0|f]          g = [0|g]
   ```

   Is `f=g` valid?

2. Semantics of non-terminating functions:

   ```
   f(X) = f(s(X))        g(X) = g(s(X))
   ```

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

Avoided by **strict equality**:
"$\equiv$": 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):

$$c \equiv c = \texttt{true}$$

$$d(x_1,\ldots,x_n) \equiv d(y_1,\ldots,y_n) = x_1 \equiv y_1 \wedge \ldots \wedge x_n \equiv y_n$$

$$\texttt{true} \wedge x = x$$

Lazy Narrowing

$$\begin{array}{rcll} \mathtt{0} & \leq & \mathtt{X} & = \mathtt{true} \\ \mathtt{s(X)} & \leq & \mathtt{0} & = \mathtt{false} \\ \mathtt{s(X)} & \leq & \mathtt{s(Y)} & = \mathtt{X} \leq \mathtt{Y} \end{array}$$

1. Apply narrowing steps at outermost positions:

$$\mathtt{0} \leq \mathtt{s(0)+s(0)} \quad \leadsto_{\{\}} \quad \mathtt{true}$$

$$\mathtt{X} \leq \mathtt{Y+Y} \quad \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} \quad \mathtt{true}$$

2. Evaluate arguments if necessary:

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

Also possible:

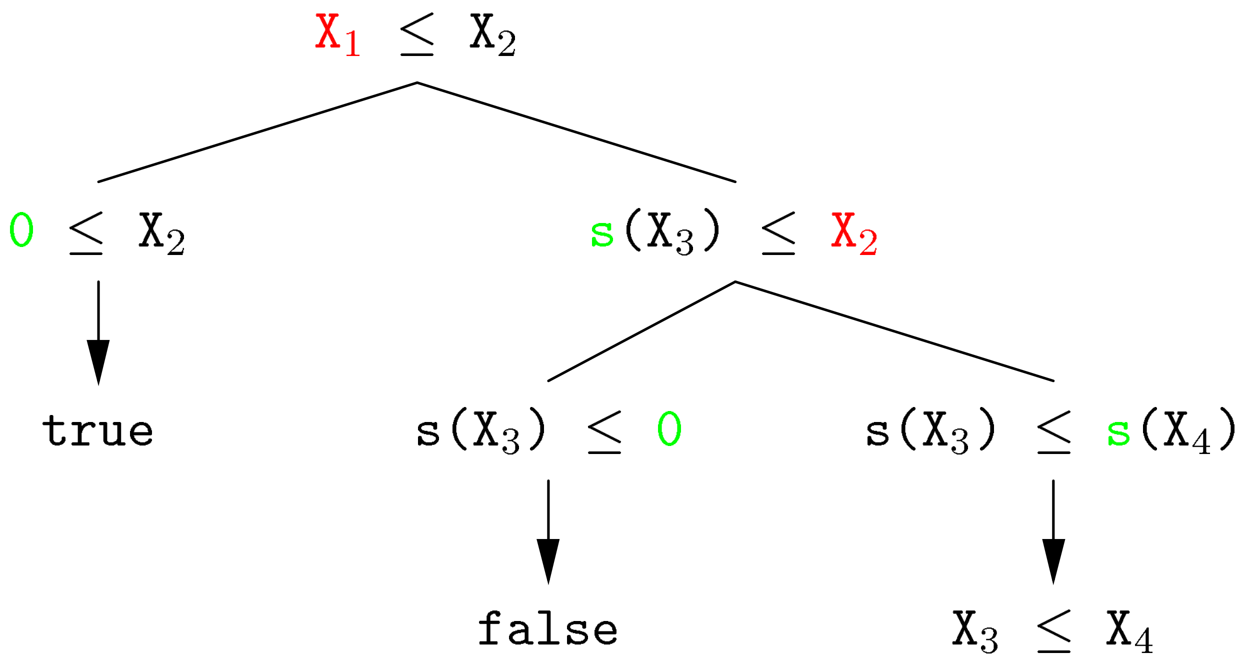
$$\begin{array}{lll} \mathtt{X} \leq \mathtt{Y+Y} & \leadsto_{\{\mathtt{Y}\mapsto\mathtt{0}\}} & \mathtt{X} \leq \mathtt{0} \\ & \leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}} & \mathtt{true} \end{array}$$

Superfluous step + specialized answer!

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

```
   0  ≤ X    = true
s(X) ≤ 0    = false
s(X) ≤ s(Y) = X ≤ Y
```



Function call: $t_1 \leq t_2$

1. Reduce t_1 to head normal form
2. If $t_1 = \texttt{0}$: apply rule
3. If $t_1 = \texttt{s}(\cdots)$: reduce t_2 to head normal form
4. If t_1 variable: bind t_1 to `0` or `s(X)` and proceed

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 non-deterministic 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

$$\begin{array}{rcl} \texttt{true} \lor \texttt{X} & = & \texttt{true} \\ \texttt{X} \lor \texttt{true} & = & \texttt{true} \\ \texttt{false} \lor \texttt{false} & = & \texttt{false} \end{array}$$

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

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

Disadvantages:

- 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

$$\begin{array}{rclcrcl} \mathtt{true} \lor \mathtt{X} & = & \mathtt{true} & & & & \\ \mathtt{X} \lor \mathtt{true} & = & \mathtt{true} & \qquad & \mathtt{f(a)} & = & \mathtt{true} \\ \mathtt{false} \lor \mathtt{false} & = & \mathtt{false} & & & & \end{array}$$

$\mathtt{f(X)} \lor \mathtt{f(X)}$:

1. Computed substitutions: $\{\mathtt{X} \mapsto \mathtt{a}\}$ and $\{\mathtt{X} \mapsto \mathtt{a}\}$

2. Minimized substitution set: $\{\ \{\mathtt{X} \mapsto \mathtt{a}\}\ \}$

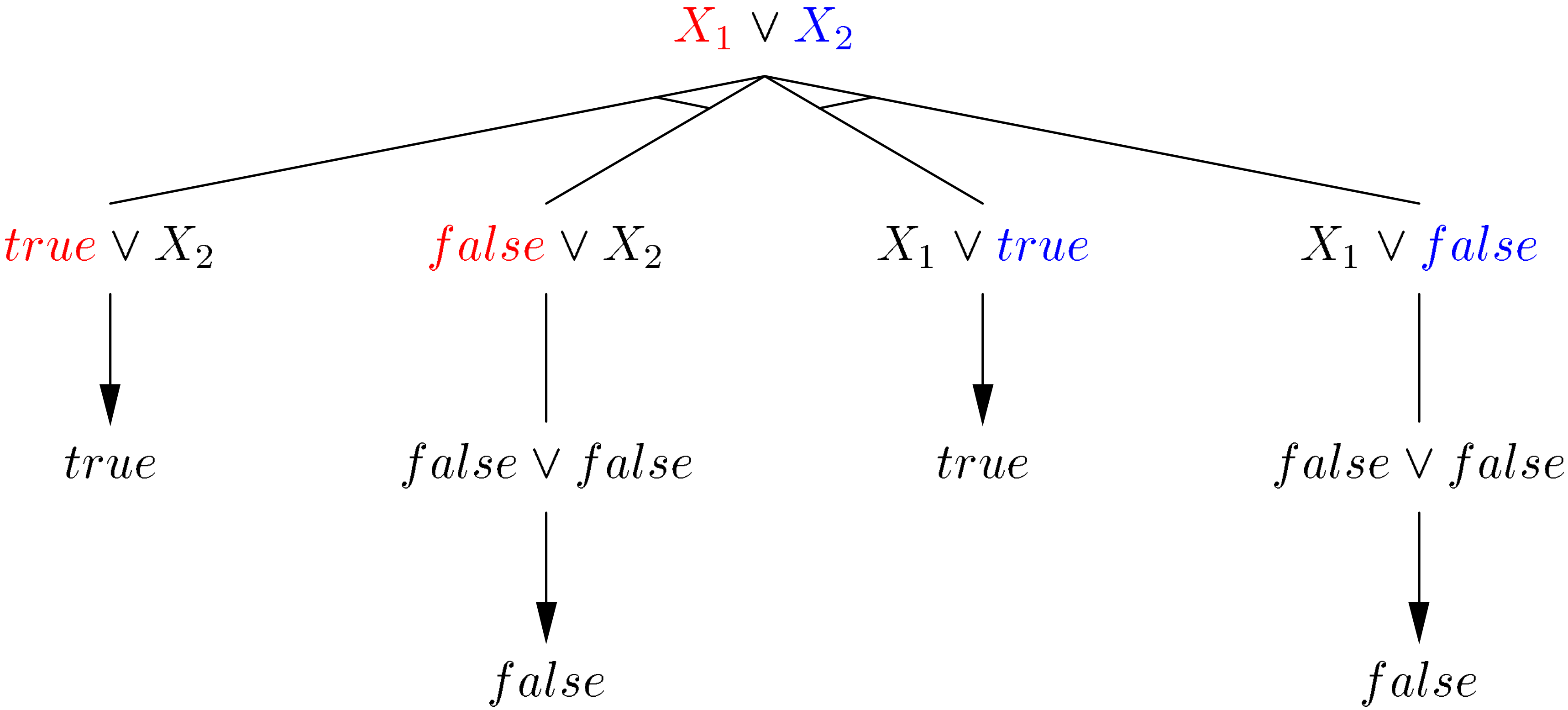
3. Apply substitution + parallel rewriting:

   $\mathtt{f(X)} \lor \mathtt{f(X)} \ \ \rightsquigarrow_{\{\mathtt{X}\mapsto\mathtt{a}\}}\ \ \mathtt{true} \lor \mathtt{true} \ \ \rightsquigarrow_{\{\}}\ \ \mathtt{true}$

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

```
X * 0 = 0        f(s(s(X))) = 0
0 * X = 0        g(X) = g(s(X))
```

`g(X) * (f(Y) * (0 * f(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:

$$\texttt{g(X)*X} \rightsquigarrow_{\{\texttt{X}\mapsto\texttt{0}\}} \texttt{0}$$

Commitment to the identity substitution:

$$\begin{aligned} \texttt{g(X)*X} &\rightsquigarrow_{\{\}} \texttt{g(s(X))*X} \\ &\rightsquigarrow_{\{\}} \texttt{g(s(s(X)))*X} \ \cdots \end{aligned}$$

Properties of Parallel Narrowing

[Antoy/Echahed/Hanus ICLP'97]

- **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)

  $\rightsquigarrow$ Function evaluation may cause non-determinism

Disadvantage of narrowing:

- function inversion not reasonable for all functions
- functions on recursive data structures
  $\rightsquigarrow$ narrowing may not terminate
- all rules must be explicitly known
  $\rightsquigarrow$ combination with external functions unclear (basic arithmetic,...)

Alternative principle:

Evaluate only sufficiently instantiated function calls

$\Longrightarrow$ **Residuation**

Execution Principle: Residuation

[Aït-Kaci et al. 87]

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

Delay functions calls until deterministic reduction is possible

$$
\begin{array}{rcl@{\qquad}l}
\texttt{0+X} & = & \texttt{X} & \texttt{nat(0)} \\
\texttt{s(X)+Y} & = & \texttt{s(X+Y)} & \texttt{nat(s(X))} \Leftarrow \texttt{nat(X)}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{X+0 = s(0)} \wedge \texttt{nat(X)}} \\
\leadsto_{\{\texttt{X}\mapsto\texttt{s(X1)}\}} & \texttt{s(X1)+0 = s(0)} \wedge \texttt{nat(X1)} \\
\leadsto_{\{\}} & \texttt{s(X1+0) = s(0)} \wedge \texttt{nat(X1)} \\
\leadsto_{\{\texttt{X1}\mapsto\texttt{0}\}} & \texttt{s(0+0) = s(0)} \\
\leadsto_{\{\}} & \texttt{s(0) = s(0)}
\end{array}
$$

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

Functions: Deterministic evaluation

Predicates: Non-deterministic evaluation

Properties of Residuation

- evaluate functions always in a deterministic way
- express non-determinism by predicates (Le Fun, Life, NUE-Prolog) or disjunctions (Escher, Oz)
- **incomplete**:
  compute the last element `E` of a list `L`:
  solve equation `append(_,[E])=L` $\rightsquigarrow$ *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)
```

⇒ 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)

- non-determinism in predicates or disjunctions

Narrowing:

+ completeness w.r.t. functional and logic programming
+ optimal evaluation (for typical programs)

– expensive function inversion (non-determinism)

– combination with external functions unclear

- implicit non-determinism ($\sim$ logic programming)

$\Longrightarrow$ **Combine residuation and narrowing**

$\Longrightarrow$ Operational principle of **Curry**

Disjunctive Expressions

Functional programming: *values* of expressions

Logic programming: *computed answers*

Answer expression: $\{\sigma\}\, e$ (bindings + expression)

$\{\sigma\}\, e$ solved: e does not contain defined functions

Disjunctive expression: $\{\sigma_1\}\, e_1 \mid \ldots \mid \{\sigma_n\}\, e_n$

(omit identity substitutions)

Computation step: reduce an unsolved expression

Deterministic step: replace expression by new expression

Non-deterministic step: replace expression by disjunction of new expressions

```
f(a) = c
f(b) = d
```

f(b) $\leadsto$ d

f(X) $\leadsto$ {X=a} c | {X=b} d

Compute necessary bindings with definitional trees

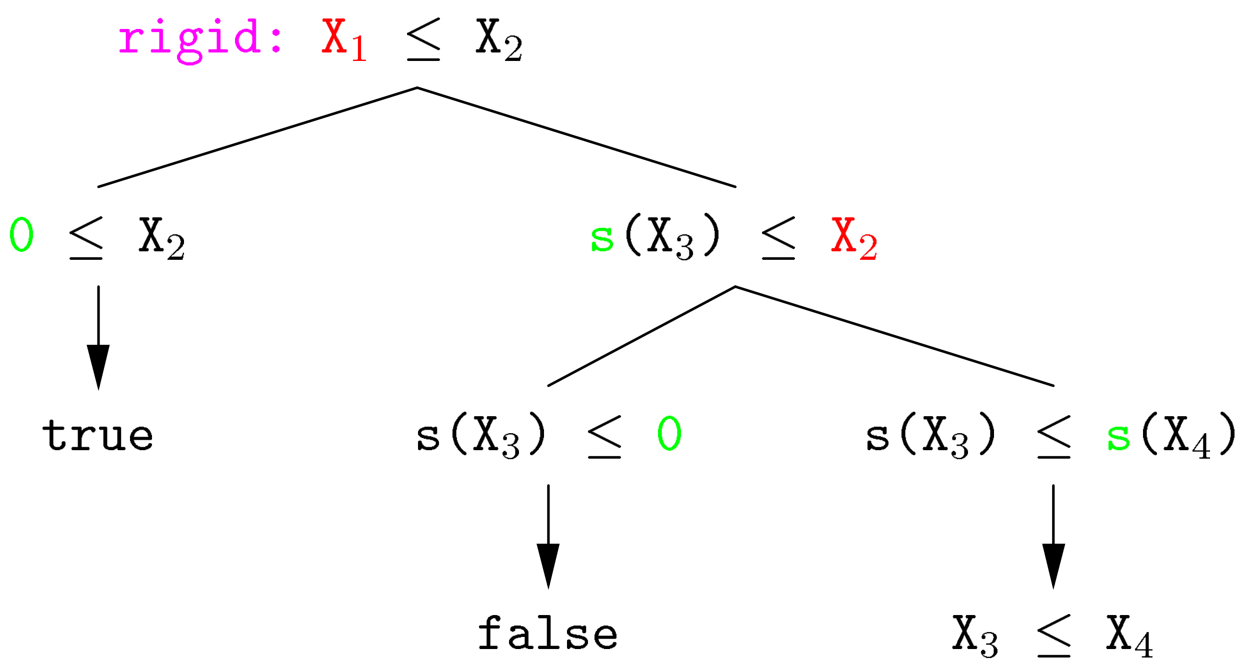
$\leadsto$ *needed narrowing*

Mark *branch* nodes as *rigid* $\leadsto$ *residuation*

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

```
   0  ≤ X    = true
s(X) ≤ 0    = false
s(X) ≤ s(Y) = X ≤ Y
```



Function call: $t_1 \le t_2$

1. Reduce t_1 to head normal form
2. If $t_1 = 0$: apply rule
3. If $t_1 = s(\cdots)$: reduce t_2 to head normal form
4. If t_1 variable: [redacted]
   do not evaluate function call

Constraints and Concurrency

Equational constraint e_1=e_2:

satisfied if both sides reducible to unifiable data terms (*strict equality*)

Constraints

- occur in conditions of rules: $l = r \Leftarrow c$
- are solved when this rule is applied
- are connected by concurrent conjunction $\wedge$:
  To evaluate $c_1 \wedge c_2$, evaluate c_1
  If evaluation of c_1 suspends, evaluate c_2

  Notation: $\{c_1, \ldots, c_n\}$ instead of $c_1 \wedge \ldots \wedge c_n$
- are first-class objects:

Transform a list of constraints into a single constraint:

```
conj([])      = {}
conj([C|Cs]) = {C, conj(Cs)}
```

$\leadsto$ provide functions to generate flexible process architectures (Eden, Goffin)

Soundness and Completeness

Relate derivations to standard rewriting $\rightarrow_{\mathcal{R}}$ ($\rightarrow_{\mathcal{R}}$ sound and complete w.r.t. model-theoretic semantics)

Soundness: If

$$e \quad \leadsto^* \quad \{\sigma_1\}\, e_1 \mid \ldots \mid \{\sigma_n\}\, e_n$$

then $\sigma_i(e) \rightarrow^*_{\mathcal{R}} e_i$ for $i = 1, \ldots, n$

Completeness: If $\sigma(e) \rightarrow^*_{\mathcal{R}} c$ and

$$e \quad \leadsto^* \quad \{\sigma_1\}\, e_1 \mid \ldots \mid \{\sigma_n\}\, e_n$$

then $\exists \varphi, i$ with $\sigma = \varphi \circ \sigma_i$ and $\varphi(e_i) \rightarrow^*_{\mathcal{R}} c$

Completeness w.r.t. flexible functions:
All *branch* nodes *flexible*: If $\sigma(e) \rightarrow^*_{\mathcal{R}} c$, then $\exists$

$$e \quad \leadsto^* \quad \{\sigma_1\}\, e_1 \mid \ldots \mid \{\sigma_n\}\, e_n$$

with $e_i = c$ and $\sigma = \varphi \circ \sigma_i$ for some i and φ

Definitional Trees: Unification of Computation Models

Strategy	Restrictions on definitional trees
Needed narrowing [POPL'94]	only *rule* and flexible *branch* nodes; optimal w.r.t. length of derivations and number of computed solutions
Weakly needed narrowing (∼Babel)	only *rule*, flexible *branch*, and *or* nodes
Simple lazy narrowing and resolution (∼Prolog)	particular trees with flexible *branch* nodes (one *branch*/*rule* tree for each left-hand side, all rules connected by *or* nodes)
Lazy functional languages (∼Haskell)	trees with left-to-right pattern matching; initial expression has no free variable
Residuation (∼Life)	rigid branches for non-Boolean functions; flexible branches for predicates

Curry: A Truly Integrated Functional Logic Language

[ILPS'95, Dagstuhl'96, POPL'97]

- combines most important features of existing
  - functional languages
  - (concurrent) logic languages
  - functional logic languages
- definitional trees are automatically generated:
  - left-to-right pattern matching
  - *rigid* branches for functions
  - *flexible* branches for predicates
- conditional rules
- equational constraints, concurrency
- higher-order functions
- polymorphic type system, modules
- declarative (monadic) I/O
- committed choice, encapsulated search
- external functions and constraint solvers

Programming in Curry

```
append([]    ,Ys) = Ys
append([X|Xs],Ys) = [X|append(Xs,Ys)]
```

Functional programming:

`append([1,2],[3,4])` $\rightsquigarrow$ `[1,2,3,4]`

Logic programming (branch is *flexible*):

`append(L,M)=[1,2]` $\rightsquigarrow$

`{L=[],M=[1,2]} | {L=[1],M=[2]} | {L=[1,2],M=[]}`

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

Lazy functional programming:

`first(s(s(0)),from(0))` $\rightsquigarrow$ `[0,s(0)]`

Lazy functional logic programming:

`first(X,from(Y))=[0]` $\rightsquigarrow$ `{X=s(0),Y=0}`

Conditional Rules

```
member(E,L) = true ⇐ {append(_,[E|_])=L}
```

Conditional rules l = r $\Leftarrow$ c

- condition c: conjunction of equational constraints
- extra variables in conditions
- implement conditional rule as "l = $(c \Rightarrow r)$"
- semantics of *guarded expression* "$c \Rightarrow r$":

```
({} ⇒ X)  =  X
```

```
member(X,[a,b,c])
  ⇝   {X=a} | {X=b} | {X=c}
```

Translation: Logic programs $\hookrightarrow$ Curry:

```
p :- p1,...,pn
↦ p = true ⇐ (p1 &&...&& pn)=true       (sequential)
↦ p = true ⇐ {p1=true,...,pn=true}      (concurrent)
```

Better: define functions instead of predicates

Functions vs. Predicates

rigid branches for functions not always reasonable:

```
append([]    ,Ys) = Ys
append([X|Xs],Ys) = [X|append(Xs,Ys)]
```

Concatenate known lists:

`append([a,b],[c,d])` $\rightsquigarrow$ `[a,b,c,d]`

Splitting a list:

`append(L,[b]) = [a,b]` $\rightsquigarrow$ *not reducible*

Escher [Lloyd 94]: provide additional split predicate (superfluous from a declarative point of view)

Prolog: define `append` always as a predicate
$\Rightarrow$ worse operational behavior than a function:

Curry: `append(append(X,Y),Z) = []`
finite search space (with *flexible* branch node)

Prolog: `append(X,Y,L), append(L,Z,[])`
infinite search space

Functional Logic Programming vs. (Concurrent) Logic Programming

Implementation of functions by flattening
$\leadsto$ loss of functional dependencies:

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

`first(X,from(X)) = []`

$\leadsto$ `{X=0} [] = []` $\mid$ `{X=s(N)}` $\ldots$*failure*$\ldots$

$\leadsto$ `{X=0}`

Translation of functions into predicates by flattening:

```
from(N,[N|R]) :- from(s(N),R).
first(0,L,[]).
first(s(N),[E|L],[E|R]) :- first(N,L,R).
```

`first(X,L,[]), from(X,L)`

$\leadsto_{\{\mathtt{X}\mapsto\mathtt{0}\}}$ `from(0,L)` $\leadsto$ `from(s(0),L1)` $\leadsto$ $\cdots$

Higher-Order Features

Higher-order functions:

```
map :: ((A -> B), [A]) -> [B]
map(F,[]   ) = []
map(F,[E|L]) = [F@E|map(F,L)]
```

`map(add@1,[1,2])` $\leadsto$ `[2,3]`

- higher-order features of functional languages (partial applications, λ-abstractions)
- application function `F@X` $\approx$ `F(X)` (implemented as in [Warren 82])
- first argument of `@` is *rigid*

  $\leadsto$ delay applications with unknown functions
- future extension(?): higher-order unification

Monadic Input/Output

- declarative I/O concept
- I/O: transformation on the outside world
- interactive program: compute **actions**
  (transformation on the *world*)
- type of actions: $\boxed{\texttt{IO(t)} \approx \texttt{World -> (t,World)}}$

  ```
  getChar :: IO(Char)
  getLine :: IO(String)
  putLine :: String -> IO()
  ```

  `getChar` applied to a world
  $\leadsto$ character + new (transformed) world

- compose actions:
  `(>>=) :: IO(a) -> (a -> IO(b)) -> IO(b)`

  `getLine >>= putLine`:
  copies a line from input to output

- no I/O in disjunctions ("cannot copy the world"):
  encapsulate search

Encapsulated Search

Technique to avoid global search (backtracking)
(non-backtrackable I/O, efficiency control,...)

Idea:
Compute until a non-deterministic step occurs,
then give programmer control over this situation
(generalization of Oz's search operator)

Search:

- solve constraint containing search variable
- bind search variable to different solutions
  $\Rightarrow$ abstract search variable: $\lambda x.\{c\}$
- primitive search operator:

```
try :: (A -> Constraint) -> [A -> Constraint]
```

$\texttt{try}(\lambda\texttt{X}.\{\texttt{1=2}\})$	$\leadsto$	`[]`	*failure*
$\texttt{try}(\lambda\texttt{X}.\{\texttt{[X]=[0]}\})$	$\leadsto$	$[\lambda\texttt{X}.\{\texttt{X=0}\}]$	*success*
$\texttt{try}(\lambda\texttt{X}.\{\texttt{X}\leq\texttt{0=true}\})$	$\leadsto$	$[\lambda\texttt{X}.\{\texttt{X=0, 0}\leq\texttt{0=true}\},$ $\lambda\texttt{X}.\{\texttt{X=s(Y), s(Y)}\leq\texttt{0=true}\}]$	*disjunction*

Encapsulated Search: Search Strategies

`try(`$\lambda x.\{c\}$`)`: evaluate, stop after non-deterministic step

Depth-first search: collect all solutions

```
dfs :: (A -> Constraint) -> [A -> Constraint]
dfs(G) = collect(try(G))
where
 collect([])    = []
 collect([S])   = [S]
 collect([G1,G2|Gs]) =
                    concat(map(dfs,[G1,G2|Gs]))
```

`dfs(λL.{append(L,[1])=[0,1]})` $\rightsquigarrow$ `[λL.{L=[0]}]`

Further search strategies:

- compute only first solution:

  `first(G) = head(dfs(G))`

- best solution search
- $\cdots$

External Functions

- infinite set of defining equations

```
0+0 = 0
0+1 = 1
0+2 = 2
...
2+1 = 3
...
```

- definition not accessible
- external implementation (without side effects)
- suspend external function calls until arguments are fully known, i.e., ground
  [Bonnier/Maluszynski 88, Boye 91]
- external function interface
- implementation of basic arithmetic
  (`+`, `-`, `*`,...: external functions)

Not possible in narrowing-based languages!

Arithmetic

$0, 1, 2, \ldots$: constructors

`+`, `-`, `*`,...: external functions

`X=2+3*4` $\rightsquigarrow$ `{X=14}`

`X=2*3+Y` $\rightsquigarrow$ `{} X=6+Y` (*suspend*)

`X+X=Y, X=2` $\rightsquigarrow$ `{X=2} 2+2=Y` (suspend `X+X`)
$\rightsquigarrow$ `{X=2} 4=Y` (evaluate `2+2`)
$\rightsquigarrow$ `{X=2, Y=4}`

$\Rightarrow$ Functions as passive constraints (Life)

```
digit(0) = true
...
digit(9) = true
```

`{X+X=Y, X*X=Y, digit(X)=true}`

$\rightsquigarrow$ `{X=0, Y=0} | {X=2, Y=4}`

Concurrent Objects with State

Modelling objects with state as a constraint function:

- first parameter: stream of messages (wait for input)
- second parameter: current state

Example: **bank account**

```
data messages = deposit(int) | withdraw(int)
                      | balance(int)

account([],_)                = {}
account([deposit(A) |Ms],N)  = account(Ms,N+A)
account([withdraw(A)|Ms],N)  = account(Ms,N-A)
account([balance(B) |Ms],N)  = {B=N,
                                 account(Ms,N)}

make_account(S) = account(S,0)
```

```
{make_account(S),  % create account object
 S=[deposit(200),withdraw(50),balance(B)]}
```

$\leadsto$ {B=150, S=...}

Implementation

- First prototype implementation available (Prolog interpreter)
  `http://www-i2.informatik.rwth-aachen.de/`
  `~hanus/tastecurry`

- Efficient implementation possible based on pattern-matching

- [Hanus LOPSTR'95]: Efficient implementation of needed narrowing by transforming definitional trees into Prolog

- Better implementation of deterministic computations by compilation into abstract machines ($\sim$ ALF, Babel, ...)

- Implementation in Java (threads for concurrency) under work [Hanus/Sadre ILPS'97 Workshop]

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
- combine research efforts in FP and LP

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

Declarative Programming

[Hanus PLILP'97]

Curry: A True Integration of Declarative Paradigms

Functional programming: higher-order functions, lazy evaluation, deterministic evaluation of ground expressions

Logic programming: logical variables, partial data structures, search facilities

Evaluation strategy:

- specified by definitional trees
- lazy reduction of expressions
- necessary instantiation of free variables
- suspension on uninstantiated variables
- concurrent (constraint) programming
- higher-order programming

More infos on Curry:

`http://www-i2.informatik.rwth-aachen.de/~hanus/curry`