A Unified Computation Model for Declarative Programming

María Alpuente
Technical University of Valencia
A Unified Computation Model for Declarative Programming

María Alpuente
Technical University of Valencia
• The goal: *reboost* functional logic languages

• The problem: multiplicity of proposals
  experimental systems
  lack of development tools

• The solution: **unified computation model** (standard)
  semantics-based optimization
  industrial tools
Declarative Programming

vs.

*Imperative* Programming

**Program**

Transcription of an *algorithm*

**Instructions**

*Commands*

**Computation Model**

*States* machine

**Variables**

Pointers to *memory*

```
function length (l: string): nat
b:bool;  b:= is_empty(l);

    case b of
        true: return 0;
        false: x:=prefix(l);
            return 1+length(x)   ...

{m=0, s="ab"}

m := length(s);
{m=2, s="ab"}
```
Declarative Programming ≡
Logic
as a *programming language*

- **Program**: *Specification* of a problem
- **Instructions**: *logical formulae*
- **Computation Model**: *Inferences* machine
- **Variables**: *Logical variables*

*higher level* programming
*automatic* control

more expressive power
greater productivity
smaller programs
easier maintenance

*efficient implementation* ≈
*imperative languages*
Logical formalisms

**clausal logic**

 RELATIONAL
 (Prolog)

**equational logic**

 FUNCTIONAL
 (Haskell)

many sorted logic
order sorted logic
types

modal logic:  dynamic
 temporal  inheritance
 epistemic  objects
 deontic    concurrency

epistemic knowledge

deontic norms
**Functional Programming:**

- **program = function** \( \text{input} \rightarrow \text{output} \)

- **define functions by equations** \( s = t \)

```
data Nat = 0 | s(Nat);
List(A) = nil | A : List(A);

func length: List(Char) \rightarrow Nat;

length(nil) = 0
length(x:xs) = s(length(xs))
```

- **computation = deterministically reduce expressions to values**

```
 eval length(a:a:b:nil)
3

eval compile(program-fuente)
object code
```

- **operational semantics**
  - replacing equals by equals
  - rewriting: reducing a subterm (**redex**) if it matches a **lhs**
Rewriting (Reduction)

\[(l = r) \in E\]

\[t[l\sigma] \rightarrow t[r\sigma]\]

— operational semantics of an expression \(t\)

\[O(t) = t\downarrow\]

\((value\ of\ t)\)

where \(t \rightarrow^* t\downarrow \rightarrow\)

— evaluation strategies:

- selection function for redexes
- eager \((innermost, \ call\ by\ value)\)
- lazy \((outermost\ fair, \ call\ by\ name)\)
**Logic Programming:**

- based on first order predicate logic
- define relations by clauses
  \[ A \leftarrow B_1 \& \ldots \& B_n \] (implications)

\[
\begin{align*}
\text{length}(\text{nil}, 0). \\
\text{length}([X|Xs], M) \leftarrow \\
\quad \text{length}(Xs, N) \& M \text{ is } N+1.
\end{align*}
\]

- computation: **nondeterministic** inference of **solutions** by using resolution

  \[
  \text{goal ?- length}([a, a, b], M). \\
  M=3
  \]

  \[
  \text{goal ?- length}(L, 0). \text{ (no input/output)} \\Leftrightarrow L=\text{nil}
  \]
• operational semantics

— SLD-resolution of goals

\[
\text{SLD - Resolution}
\]

\begin{align*}
(\mathbf{H} & \leftarrow \mathbf{C}) \in \mathbf{P} \quad \& \quad \sigma = \text{mgu}(\mathbf{H}, \mathbf{B}_i) \\
\leftarrow \mathbf{B}_1, \ldots, \mathbf{B}_i, \ldots, \mathbf{B}_n \Rightarrow \leftarrow (\mathbf{B}_1, \ldots, \mathbf{B}_{i-1}, \mathbf{C}, \mathbf{B}_{i+1}, \ldots, \mathbf{B}_n)_{\sigma}
\end{align*}

— operational semantics of a goal \( g \)

\[
O(g) = \{ \emptyset \mid \leftarrow g \Rightarrow^* \leftarrow \text{true} \}
\]

\{answers for \( g \}\}

— evaluation strategies:

• selection function for atoms
  (Prolog: leftmost)
• computation rule
  (Prolog: top-down depth-first)
Main Differences:

<table>
<thead>
<tr>
<th>Functional P.</th>
<th>Logic P.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Equations</strong></td>
<td><strong>Clauses</strong></td>
</tr>
<tr>
<td>defining functions</td>
<td>defining relations</td>
</tr>
<tr>
<td><strong>Rewriting</strong></td>
<td><strong>SLD-resolution</strong></td>
</tr>
<tr>
<td>(pattern-matching)</td>
<td>(unification)</td>
</tr>
<tr>
<td><strong>Initial Model</strong></td>
<td><strong>Minimal Model</strong></td>
</tr>
<tr>
<td>* determinism</td>
<td>* nondeterminism</td>
</tr>
<tr>
<td>no logical variables</td>
<td>logical variables</td>
</tr>
<tr>
<td>directionality</td>
<td>no i/o arguments</td>
</tr>
<tr>
<td>duplicate code</td>
<td>invertibility</td>
</tr>
<tr>
<td>complete data</td>
<td>partial information</td>
</tr>
<tr>
<td>types &amp; polymorph.</td>
<td>no types</td>
</tr>
<tr>
<td>∞ data structures</td>
<td>finite data</td>
</tr>
<tr>
<td>lazy evaluation</td>
<td>eager evaluation</td>
</tr>
<tr>
<td>higher order</td>
<td>first order</td>
</tr>
</tbody>
</table>
Shortages of each paradigm:

**logic** languages don’t have the logic of **equality** and **functions**

**functional** languages don’t have the logic of **unification** and **logical variables**
Advantages...

...of **functional** w.r.t. **logic** languages

* higher order and types
  - better abstraction
  - better modularization

* efficient evaluation
  - deterministic reduction
  - lazy evaluation

...of **logic** w.r.t. **functional** languages

* more expressive power
  invertibility, partial data, extra var’s
* search strategies
* constraint solving
* applications in BD, IO and IA
Functional Logic Languages

thanks to the *functional dimension*:

- problems with Prolog:
  - uncontrolled nondeterminism
    inefficient, infinite loops
  - **pruning**: non-declarative (red cuts)
  - **arithmetics**: (partial predicates)
  - non-declarative I/O (backtracking?)

- integration of functions:
  - **efficiency** (determinism)
  - new programming techniques
    HO, laziness, infinite data structures
  - declarative I/O, clean arithmetic
  - most predicates are functions!

Functional logic languages can avoid impure features of Prolog
thanks to the *logic dimension*:

- expressive power of logic languages
- built-in search (e.g., backtracking)
- constraint solving (unification)
- LP applications
Basic Components:

- **SUPERLANGUAGE:** *predicates + functions*

  ```plaintext
  data Nat = 0 | s(Nat);
  
  func + : Nat → Nat → Nat;
  0 + N = N
  s(M) + N = s(M+N)
  
  pred ≤ : Nat → Nat;
  0 ≤ N
  s(M) ≤ s(N) ← M ≤ N
  
  goal ?- A+B ≤ s(0)
  A=0, B=0;
  A=0, B=s(0);
  A=s(0), B=0;
  
  - **UNIFIED COMPUTATION MODEL:**

    unification (constraint solving)
    search (e.g., backtracking)
    efficient evaluation strategies
    higher order

  - **SOUNDNESS AND COMPLETENESS**
Approaches:

- logic ⇒ **functional**: Add equality to a logic language

- functional ⇒ **logic**: Add logical var’s to a functional lang.
Approach logic $\Rightarrow$ functional

- **SYNTAX:** **Horn Clauses** with equality

\[
p \leftarrow e_1 \& \ldots \& e_n, B_1 \& \ldots \& B_n \\
\text{s=t} \leftarrow e_1 \& \ldots \& e_n, B_1 \& \ldots \& B_n
\]

- **OPERATIONAL SEMANTICS:**
  
  Flattening = translation into Prolog
  
  SLDE-resolution = SLD + equational unification
  
  SLD-resolution + new rule for functions
Approach functional ⇒ logic

- **SYNTAX**: (conditional) **equations**

  \[ s = t \leftarrow e_1 \& \ldots \& e_n \]

  - predicates as boolean functions
  - atoms \( p \) as equations \( p = \text{true} \)

- **OPERATIONAL SEMANTICS**:

  Narrowing = reduction + unification
  
  Residuation = reduction + dynamic scheduling
Approach logic $\Rightarrow$ functional

*Prolog with Equality*

SYNTAX OF PURE PROLOG:

$$p(t_1...t_n) \leftarrow B_1\&...\&B_n$$

where $p$ is not the equality.

*functions* are written as predicates using an extra argument:

\[
\begin{align*}
N + 0 &= N \\
N + s(M) &= s(N + M)
\end{align*}
\]

\[
\begin{align*}
\text{add}(N,0,N) &. \\
\text{add}(N,s(M),s(X)) &\leftarrow \text{add}(N,M,X).
\end{align*}
\]

*equality* in Prolog: **syntactic identity**

- no defined function, since clause heads cannot be equations
- each term is only equal (identical) to itself
- functions are data constructors
* Prolog with Equality *

SUPERLANGUAGE SYNTAX:

Horn Clauses with equations:

\[ p \leftarrow e_1 \& \ldots \& e_n, B_1 \& \ldots \& B_n \]
\[ s = t \leftarrow e_1 \& \ldots \& e_n, B_1 \& \ldots \& B_n \]

OPERATIONAL SEMANTICS:

Naïve Solution: equality axioms as program rules handled by SLD-resolution

\[
\begin{align*}
X &= X. & \text{%Reflexivity} \\
X &= Y & \leftarrow Y = X. & \text{%Symmetry} \\
X &= Z & \leftarrow X = Y, Y = Z. & \text{%Transitivity} \\
\text{f}(X_1, \ldots, X_n) &= f(Y_1, \ldots, Y_n) & \leftarrow \\
& \quad X_1 = Y_1, \ldots, X_n = Y_n. & \text{%(f/n) \in F} \\
p(X_1, \ldots, X_n) & \leftarrow p(Y_1, \ldots, Y_n), \\
& \quad X_1 = Y_1, \ldots, X_n = Y_n. & \text{%(p/n) \in P}
\end{align*}
\]
* SLD-resolution: soundness and completeness!

* huge search space
  - redundant derivations
  - infinite branches

**example**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>( X + 0 = X )</td>
</tr>
<tr>
<td>2)</td>
<td>( X + s(Y) = s(X+Y) )</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
<td>( X = X ) ← true.</td>
</tr>
<tr>
<td>sym</td>
<td>( X = Y ) ← ( Y = X ).</td>
</tr>
<tr>
<td>trans</td>
<td>( X = Z ) ← ( X = Y, Y = Z ).</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s1)</td>
<td>( s(X) = s(Y) ) ← ( X = Y ).</td>
</tr>
<tr>
<td>s2)</td>
<td>( X_1 + Y_1 = X_2 + Y_2 ) ← ( X_1 = X_2, Y_1 = Y_2 ).</td>
</tr>
</tbody>
</table>
Classical solutions:

Components of a theorem prover based on resolution:

* syntactic unification algorithm
* SLD-resolution

1. **Flattening** = translation into Prolog  
   \textit{Brand’75}

2. **SLDE-resolution** = SLD + equational unification  
   \textit{Plotkin’72}
   
   SLD for predicates + new rule for functions
   
   - **Residuation** = delay function calls which are not sufficiently instantiated  
     \textit{Aït-Kaci’86}
   
   - **Narrowing** = reduction + unification  
     \textit{Fay’74}
Flattening: Translation into Prolog

1. Decompose nested functional expressions $C[t]$ using *fresh variables* $Z$:

    * within the body:
      
      
      $A \leftarrow B[t]$
      
      $A \leftarrow t = Z, B[Z]$

    * within the head:
      
      
      $l = c[t] \leftarrow B$
      
      $l = c[Z] \leftarrow t = Z, B$

2. Apply SLD-resolution to the flat program and goal

Example: Tamaki’84/Levi’84 transformation:

\begin{Verbatim}
0 + N = N
s(M) + N = s(M+N)

0 \leq N
s(M) \leq s(N) \leftarrow M \leq N
\end{Verbatim}

goal \texttt{?- A+B \leq s(0)}

\begin{Verbatim}
0 + N = N
s(M) + N = s(Z) \leftarrow M+N = Z

0 \leq N
s(M) \leq s(N) \leftarrow M \leq N
\end{Verbatim}

goal \texttt{?- A+B = C, C \leq s(0)}
**Example:** van Emden’84 transformation:

1. Decompose nested functional expressions
2. Transform each function into a predicate
3. Apply SLD-resolution to the flat program and goal

\[
\begin{align*}
\text{add}(N, 0, N) \\
\text{add}(N, \text{s}(M), \text{s}(X)) & \leftarrow \text{add}(N, M, X) \\
0 & \leq N \\
\text{s}(M) & \leq \text{s}(N) \leftarrow M \leq N
\end{align*}
\]

**goal** \( \text{?- add}(A, B, C), \ C \leq \text{s}(0) \)
Flattening:

+ the equality axioms are subsumed by the transformation
  the equality is an ordinary predicate after the transformation (or it disappears)

+ easy to implement (preprocessing)
  SLD and unification need not be changed

+ both components of the language (functional and logic) are handled by a single rule: SLD-resolution

+ functions and predicates can be mutually recursive
  predicates can appear in the conditions of equations and functional expressions can appear in clauses

+ equivalent to a refined narrowing
  innermost basic narrowing
Flattening:

— functions are translated into *nondeterministic* predicates
— loss of information (funct. dependencies)
— logic programs are less efficient than equivalent functional programs

**Example:** *functional* max program

\[
\begin{align*}
\text{max}(X, Y) &= Y \leftarrow X \leq Y \\
\text{max}(X, Y) &= X \leftarrow Y \leq X
\end{align*}
\]

**goal** ?- max(3,3)=Z

*deterministic, one solution*

**Example:** *relational* max program

\[
\begin{align*}
\text{max}(X, Y, Y) &\leftarrow X \leq Y \\
\text{max}(X, Y, X) &\leftarrow Y \leq X
\end{align*}
\]

**goal** ?- max(3,3,Z)

*nondeterministic, two (identical) solutions*
many opportunities to apply deterministic reduction are lost:

*Nested expressions* within program and goal:

\[
\begin{align*}
\text{f(h(X))} &= \text{f(X)} \\
\text{f(g(X))} &= \text{X} \\
\text{g(X)} &= \text{g(X)}
\end{align*}
\]

\text{goal} \ ?- \ f(g(X)) = 0

*finite search space*

*Flat* program and goal:

\[
\begin{align*}
\text{f(h(X))} &= \text{Z} \leftarrow \text{f(X)}=\text{Z} \\
\text{f(g(X))} &= \text{X} \\
\text{g(X)} &= \text{Z} \leftarrow \text{g(X)}=\text{Z}
\end{align*}
\]

\text{goal} \ ?- \ g(X)=\text{W}, \ f(\text{W})=0

*infinite search space*
SLDE: SLD with equational unification

Use SLD-resolution as inference rule, but replace syntactic unification (=) by equational or semantic unification (=E)

unification in a context where two terms which are not identical are considered equal

Example:

$$\text{max}(3, 8) \neq \text{max}(10, X)$$

whereas

$$\text{max}(3, 8) =_E \text{max}(4, X)$$
SLD-Resolution

\[(H \leftarrow C) \in P \quad & \quad B_i \sigma = H \sigma\]

\[\leftarrow B_1, \ldots, B_i, \ldots, B_n \quad \Rightarrow \quad \leftarrow (B_1, \ldots, B_{i-1}, C, B_{i+1}, \ldots, B_n) \sigma\]

SLDE-Resolution

\[(H \leftarrow C) \in P \quad & \quad B_i \sigma \models E H \sigma\]

\[\leftarrow B_1, \ldots, B_i, \ldots, B_n \quad \Rightarrow \quad \leftarrow (B_1, \ldots, B_{i-1}, C, B_{i+1}, \ldots, B_n) \sigma\]
SLDE-Resolution:

+ both components of the language (functional and logic) are handled by a single rule: SLDE-resolution

— E-unification is only semidecidable
  
  the interpreter can loop

— the set of E-unifiers of two terms may be infinite
  
  the search space may not only be infinitely deep but also infinitely broad

— predicates cannot appear in the conditions of the equations

well-behaved programs:

\[
p \leftarrow e_1 \& \ldots \& e_n, B_1 \& \ldots \& B_n \\
\text{s} = \text{t} \leftarrow e_1 \& \ldots \& e_n
\]
Operational semantics

⇒ LP semantics:

$$SS(P) = \{ A \in B_P \mid \leftarrow A \text{ has a SLD refutation in } P \}$$

where $$B_P = \{ p(d) \mid d \in H_P^n \}$$

⇒ FLP semantics:

$$SS(P, E) = \{ A \in B_{P, E} \mid \leftarrow A \text{ has a SLDE refutation in } P, E \}$$

where $$B_{P, E} = \{ p(d) \mid d \in (H_P/E)^n \}$$

Declarative semantics

⇒ LP semantics:

$$M_P = \{ A \in B_P \mid P \models A \}$$

⇒ FLP semantics:

$$M_{P, E} = \{ A \in B_{P, E} \mid P \models_{H_P/E} A \}$$
Residuation:

*Delay* function calls which are not sufficiently instantiated

\[
X + X = Y \quad \& \quad X = 2 \quad \%\text{delay } X + X
\]

\[
\Rightarrow \{ X = 2 \} \quad 2 + 2 = Y \quad \%\text{evaluate } 2 + 2
\]

\[
\Rightarrow \{ X = 2 \} \quad 4 = Y
\]

\[
\Rightarrow \{ X = 2, Y = 4 \} \quad \text{true}
\]

Computed answer: \{ X = 2, Y = 4 \}
SLD + Residuation:

\[
\begin{align*}
0 + X &= X \\
s(X) + Y &= s(X+Y) \\
nat(0) \\
nat(s(X)) &\leftarrow nat(X)
\end{align*}
\]

\[
\begin{align*}
Z + 0 &= s(0) \land nat(Z) \\
\Rightarrow \{Z/s(X)\} & s(X) + 0 = s(0) \land nat(X) \\
\Rightarrow \{\} & s(X+0) = s(0) \land nat(X) \\
\Rightarrow \{X/0\} & s(0+0) = s(0) \\
\Rightarrow \{\} & s(0) = s(0)
\end{align*}
\]

Computed answer: \{Z/s(0)\}

Advantages = functions: determinism
predicates: nondeterminism

Disadvantages = incompleteness: e.g.
append(_, [E])=L \Rightarrow fail
may have an \(\infty\) search space
in contrast to narrowing
## Operational semantics

*(summary)*

<table>
<thead>
<tr>
<th></th>
<th><strong>L + F</strong></th>
<th><strong>F + L</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SYNTAX</strong></td>
<td>Horn Clauses with equality</td>
<td>equational Horn Clauses (CTRS’s)</td>
</tr>
<tr>
<td></td>
<td>$A \leftarrow B_1 &amp; \ldots &amp; B_n$</td>
<td>$e \leftarrow e_1 &amp; \ldots &amp; e_n$</td>
</tr>
<tr>
<td><strong>SEMANTICS</strong></td>
<td>1. flat-SLD</td>
<td>1. Narrowing</td>
</tr>
<tr>
<td></td>
<td>2. SLDE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. SLD + Residuation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. SLD + Narrowing</td>
<td></td>
</tr>
</tbody>
</table>
**Functional** $\Rightarrow$ **logic** approach

*Functional programming:*
initial expression without free variables

*Logic programming:*
initial expression and conditions with free variables

**Operational extension:**
*Instantiating free variables* before a reduction step

\[
\begin{align*}
f(0) &= b \\
f(1) &= c
\end{align*}
\]

Evaluate $f(X)$:

- Instantiate $X$ to $0$ and reduce $f(0)$ to $b$
  \[
  f(X) \Rightarrow_{\{x/0\}} b
  \]

- Instantiate $X$ to $1$ and reduce $f(1)$ to $c$
  \[
  f(X) \Rightarrow_{\{x/1\}} c
  \]
**Narrowing: reduction with unification**

\[
\begin{align*}
0 + X &= X \\
s(X) + Y &= s(X+Y)
\end{align*}
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Unification</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z+0=s(0)</td>
<td>{Z/s(X),Y/0}</td>
<td>s(0)=s(0)</td>
</tr>
<tr>
<td></td>
<td>{x/0,x'/0}</td>
<td>true</td>
</tr>
<tr>
<td>0=s(0)</td>
<td>fail</td>
<td></td>
</tr>
</tbody>
</table>

Computed answer: \{Z/s(0)\}

**Advantages** = completeness for canonical programs

**Disadvantages** = huge search space
   (nondeterminism: position and rule)
REWRITING RELATION $\rightarrow$
(UNCONDITIONAL)

Let $R$ be a TRS. The term $t$ reduces to $s$, in symbols:

$$t \rightarrow s$$

if $$\exists \ u \in O(t)$$
$$\exists (l \rightarrow r) \ll R$$
$$\exists \ \sigma$$
such that:

$$t|_u = l\sigma$$
$$s = t[r\sigma|_u]$$

• Example: $s(Y + 0) \rightarrow s(Y)$ (rule: $X+0=X$)

NARROWING RELATION $\Rightarrow$
(UNCONDITIONAL)

Let $R$ be a TRS. The term $t$ narrows to $s$ with substitution $\sigma$, in symbols:

$$t \Rightarrow_\sigma s$$

if $$\exists \ u \in O_{nv}(t)$$
$$\exists (l \rightarrow r) \ll R^+ = (R \cup \{X=X \rightarrow \text{true}\})$$
$$\exists \ \sigma = \text{mgu}\{t|_u = l\}$$
such that:

$$(t|_u)\sigma = l\sigma$$
$$s = (t[r|_u])\sigma$$

• Example: $s(s(0)+Y) \Rightarrow_{\{X=s(0), Y/0\}} s(s(0))$ (rule: $X+0=X$)
Narrowing is Nondeterministic:

**THREE SOURCES FOR NONDETERMINISM:**

1. equation within the goal
2. subterm to be reduced
3. program rule

Narrowing is a semidecision algorithm for $\equiv$E
1) $f(0) = 0$
2) $f(s(X)) = f(X)$

$\Rightarrow$ Narrowing strategies
REWRITING RELATION $\rightarrow$  
(CONDITIONAL)

Let $R$ be a TRS. The goal $g$ reduces to $g'$, in symbols:

$$g \rightarrow g'$$

if $\exists \ u \in O(g)$

$$\exists (l \rightarrow r \leftarrow s_1=t_1, ..., s_n=t_n) \ \ll R$$

$$\exists \ \sigma$$

such that:

$$g|_u = l\sigma$$
$$\forall i \ \exists w_i \ \text{t.q.}$$
$$s_i\sigma \rightarrow^* R \ w_i \ R \leftarrow t_i\sigma$$

$$g' = g[r\sigma]_u$$

NARROWING RELATION $\Rightarrow$  
(CONDITIONAL)

Let $R$ be a TRS. The goal $g$ narrows to $g'$ with substitution $\sigma$, in symbols:

$$g \Rightarrow_\sigma g'$$

if $\exists \ u \in O_{nv}(g)$

$$\exists (l \rightarrow r \leftarrow C) \ll R^+ = (R \cup \{X=X \rightarrow \text{true} \leftarrow \})$$

$$\exists \ \sigma = \text{mgu}(\{g|_u = l\})$$

such that:

$$g' = (C, g[r\sigma]_u)\sigma$$
Semantics of $g$

$$O(g) = \{ \theta_{|\text{Var}(g)} \mid g \Rightarrow_{\theta_1} \ldots \Rightarrow_{\theta_n} T^1 \land \theta = \theta_1 \ldots \theta_n \}$$

- By abuse, $\theta \in O(g)$ are called solutions

- It is common to consider only normalized substitutions

---

1 $T$ is a finite sequence of “true”
Different formulations of Conditional Narrowing

• with \(<\text{skeleton } g, \text{ environment } \theta>\)

Narrowing rule\(^2\)

\[
u \in O_{\text{nv}}(g) \& (l \rightarrow r \leftarrow C)\Rightarrow R^+ \& \sigma = \text{mgu}(\{g | u = l\})
\]

\[
<g, \theta> \Rightarrow <(C, g[r]_u)\sigma, \theta\sigma>
\]

Semantics of \(g\)

\[
O(g) = \{ \theta_{|\text{Var}(g)} | \text{ } <g, \varepsilon> \Rightarrow* <\text{T}, \theta> \}
\]

---

\(^2\)We use the same notation \(\Rightarrow\) whenever there is no ambiguity
• with reflexion rule

**Reflexion rule**

\[ \sigma = \text{mgu}\{e\} \]

\[ <(g_1,e,g_2), \theta> \Rightarrow <(g_1,\text{true},g_2), \theta\sigma> \]

**Narrowing rule**

\[ u \in O_{nv}(g) \land (l \rightarrow r \leftarrow C) \Rightarrow R \land \sigma = \text{mgu}\{g_{|u}=l\} \]

\[ <g, \theta> \Rightarrow <(C, g[r]_{u})\sigma, \theta\sigma> \]

**Semantics of g**

\[ O(g) = \{ \theta_{|\text{Var}(g)} \mid <g, \varepsilon> \Rightarrow^* <T, \theta> \} \]
• with unification rule

Unification rule

$$\sigma = \text{mgu}(g)$$

$$\langle g, \theta \rangle \Rightarrow \langle \text{true}, \theta \sigma \rangle$$

Narrowing rule

$$u \in O_{nv}(g) \& (l \rightarrow r \leftarrow C) \ll R \& \sigma = \text{mgu}(\{g|u=l\})$$

$$\langle g, \theta \rangle \Rightarrow \langle (C, g[r]_u)\sigma, \theta \sigma \rangle$$

Semantics of $g$

$$O(g) = \{ \theta_{|\text{Var}(g)} \mid <g, \varepsilon> \Rightarrow^* <\text{true}, \theta> \}$$
Completeness

• Conditional narrowing is a complete E-unification algorithm under certain conditions

\[ \exists \theta \in O(g). \quad \theta \leq E \sigma \]

• i.e., for every E-unifier \( \sigma \) of \( s \) and \( t \) (\( \forall \sigma. \quad s \sigma = E t \sigma \)):

• Conditional narrowing is **complete** for canonical programs without extra variables

• The subscript E can be dropped from \( \leq \) if we only consider normalized substitutions

• For **confluent programs**, narrowing is only complete w.r.t. normalized substitutions
Classification of CTRSs

[Middeldorp-Hamoen'92]

<table>
<thead>
<tr>
<th>Class</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\text{Var}(l) \supseteq \text{Var}(r) \cup \text{Var}(C))</td>
</tr>
<tr>
<td>2</td>
<td>(\text{Var}(l) \supseteq \text{Var}(r))</td>
</tr>
<tr>
<td>3</td>
<td>(\text{Var}(l) \cup \text{Var}(C) \supseteq \text{Var}(r))</td>
</tr>
<tr>
<td>4</td>
<td>none</td>
</tr>
</tbody>
</table>

- Conditional narrowing is complete for canonical programs without extra variables (1-CTRSs)
- and for confluent 1-CTRSs w.r.t. normalized substitutions
- It is not complete for canonical 2-CTRSs (not to speak of 3-CTRSs)
Strong Completeness

- Selection function $S$

  mapping $S$: $Goal \Rightarrow Nat$

  $g=(e_1,\ldots,e_n) \Rightarrow i (e_i)$

- **Strong completeness** = completeness is not lost when
  narrowing is restricted to a single equation in each goal

  (i.e., the choice of the equations is don’t-care
  nondeterministic)

- Conditional narrowing is **strong complete** for
  canonical 1-CTRS’s w.r.t. normalized
  substitutions:

  For every $S$, $D = [\leftarrow g \Rightarrow \theta^* \leftarrow T]$ s.t. $\theta$ is normalized

  there exists $D' = [\leftarrow g \Rightarrow S^\theta^* \leftarrow T]$ respecting $S$
• Narrowing subsumes term rewriting:

If the narrowing step $t \Rightarrow^\sigma s$ can be proven, then $t \sigma \rightarrow s$ is a reduction step

(the opposite direction only holds for nontrivial TRSs)

• Narrowing subsumes SLD-resolution:

If $P \cup \{\leftarrow g\} \Rightarrow^{\theta^*} (SLD) \leftarrow true$

1. transform $P$ into $R$ by replacing each atom $p(t_1, \ldots, t_n)$ by an equation $p(t_1, \ldots, t_n) = true$

2. transform $g$ into $g'$

3. then, $R \cup \{\leftarrow g'\} \Rightarrow^{\theta^*} (Narrowing) \leftarrow T$
• Narrowing is a refinement of a more general inference rule called **Paramodulation**

Consider

\[
E = \{ f(a) = b \quad a = c \}
\]

\[
g = (f(c) = b)
\]

SLD refutation using the equality axioms:

\[
\text{← } f(c) = b
\]

<table>
<thead>
<tr>
<th>TRANSITIVITY</th>
</tr>
</thead>
</table>

\[
\text{← } f(c) = V, \ V = b
\]

<table>
<thead>
<tr>
<th>F-SUSTITUTIVITY</th>
</tr>
</thead>
</table>

\[
\text{← } c = V2, \ f(V2) = b
\]

<table>
<thead>
<tr>
<th>SIMETRY</th>
</tr>
</thead>
</table>

\[
\text{← } V2 = c, \ f(V2) = b
\]

<table>
<thead>
<tr>
<th>RESOLUTION with ( a = c )</th>
</tr>
</thead>
</table>

\[
\text{← } f(a) = b
\]

<table>
<thead>
<tr>
<th>RESOLUTION with ( f(a) = b )</th>
</tr>
</thead>
</table>

\[
\text{← true}
\]

**IDEA:** Embed this mechanism which replaces equals by equals into a specific inference rule:

\[
\begin{align*}
C1: \ & p(t) \\
C2: \ & t = \\
C3: \ & p(s)
\end{align*}
\]
Consider two clauses:

\[ C1: \ (t = t' \ \lor \ C'1) \]
\[ C2: \ (L \ \lor \ C'2) \]

where \( C'1 \) and \( C'2 \) are also clauses, and the literal \( L \) may be an equation

Let be \( s \) a subterm of \( L \) at a position \( u \), i.e. \( s = L|_u \)

Assume that \( t \) and \( s \) (respectively \( t' \) and \( s \)) do unify with mgu \( \sigma \). Then, the clauses:

\[ (C'1 \lor L[t']_u \lor C'2)\sigma \]
\[ (C'1 \lor L[t]_u \lor C'2)\sigma \]

are called \( \text{(binary) paramodulants} \) of \( C1 \) and \( C2 \) at \( u \) using \( \sigma \).

Note that paramodulation exploits the equations in both senses of direction

Example: Consider

\[ C1: \ f(X, h(Y)) = g(X, Y) \ \lor \ p(X) \]
\[ C2: \ q(h(f(h(Z), h(a)))) \]

then a paramodulant of \( C1 \) and \( C2 \) at 1.1 using \( \sigma=\{X/h(Z), Y/a\} \) is:

\[ C3: \ q(h(g(h(Z), a))) \lor p(h(Z)) \]

Completeness of Paramodulation:
• Confluence and termination are not necessary for completeness

• For non-Horn clauses, a **Factorization rule** is needed (this rule is unnecessary for Horn clauses):

**Factorization rule** If two literals $L_1$ and $L_2$ of a clause $C$ unify with mgu $\sigma$

then, a *factor* of $C$ is the clause

$$(C - L_2)\sigma$$
Completeness of Paramodulation (2):

**BINARY RESOLUTION** +
**BINARY PARAMODULATION** +
**FACTORIZATION**

is a **complete** inference system for logic programs with equality

(it infers the *empty clause* from any E-unsatisfiable set of clauses)

**provided that** the following axioms are added to the program:

**REFLEXION:**

\[ x = x \]

**FUNCTIONAL REFLEXION:**

\[ f(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \quad (f/n>0) \in \mathcal{F} \]
Refinements of Paramodulation

• **Directed Paramodulation:**

  Equations are oriented from left to right:

  \[ C_1: \ (t \rightarrow t') \lor C'1 \]

  this refinement is complete for **confluent programs**

• **Narrowing:**

  It is a further refinement of directed paramodulation which is obtained by forbidding paramodulating at variable positions

  this refinement is complete for **canonical (unconditional) programs**, etc
Refinements of Narrowing:
Narrowing strategies

Simple narrowing:
*sets of positions* to be reduced at each step

Reduction in functional languages:
*a single position* is reduced

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

Narrowing strategy = mapping

\[ \varphi: \text{Goal} \rightarrow 2^{(\text{Nat}^*)} \quad \text{(set of positions)} \]

Only selected positions are narrowed

Different *conditions for completeness* depending on \( \varphi \)
Estrategias de Reducción Funcional

Dado un programa canónico y una expresión a evaluar, cualquier estrategia de reducción conduce necesariamente al mismo valor (forma normal) de la expresión.

Pero el número de pasos necesarios para computar este valor puede depender de la estrategia empleada, es decir, del orden en que se seleccionan las (sub-) expresiones a reducir (redexes).

Ejemplo: Dado el siguiente programa:

\[
\begin{align*}
\text{sqr}(X) & = X \times X \\
\text{first}(X,Y) & = X 
\end{align*}
\]

donde la función \(*\) se asume definida en la forma habitual, el término \text{first}(\text{sqr}(4),\text{sqr}(2))\) puede evaluarse en 5 pasos de reescritura: (innermost)

\[
\begin{align*}
\text{first}(\text{sqr}(4),\text{sqr}(2)) \rightarrow \\
\text{first}(4 \times 4,\text{sqr}(2)) \rightarrow \\
\text{first}(16,\text{sqr}(2)) \rightarrow \\
\text{first}(16,2 \times 2) \rightarrow \\
\text{first}(16,4) & \rightarrow 16.
\end{align*}
\]

o en solo 3 pasos: (outermost)

\[
\begin{align*}
\text{first}(\text{sqr}(4),\text{sqr}(2)) \rightarrow \text{sqr}(4) \rightarrow \\
4 \times 4 & \rightarrow 16.
\end{align*}
\]
Modelos de reducción

Estrategia Innermost  \textit{(Call by value)}:
Estrategia en que sólo se reducen expresiones \textit{innermost} (aquellas que no contienen ningún otro redex.)

(i.e. se evalúan completamente los argumentos antes de pasar a evaluar una función más externa.)

Estrategia Outermost  \textit{(Call by name)}:
Estrategia en que sólo se reducen expresiones \textit{outermost} (aquellas no contenidas en ningún otro redex.)

(i.e. se retrasa la evaluación de los argumentos hasta que son necesarios para producir un resultado.)
Eficiencia

En ausencia de la propiedad de terminación del programa, algunas estrategias de reducción pueden terminar con éxito el cálculo del valor de una cierta expresión, mientras que otras no.

Ejemplo: Dado el siguiente programa no terminante:

\[
\begin{align*}
\text{loop} & = \text{succ}(\text{loop}) \\
\text{first}(X,Y) & = X \\
\text{answer} & = \text{first}(0,\text{loop})
\end{align*}
\]

la evaluación con estrategia innermost de la expresión \text{answer} resulta en una derivación infinita:

\[
\begin{align*}
\text{answer} & \rightarrow \text{first}(0,\text{loop}) \\
& \rightarrow \text{first}(0,\text{succ}(\text{loop})) \\
& \rightarrow \text{first}(0,\text{succ}(\text{succ}(\text{loop}))) \\
& \rightarrow \ldots
\end{align*}
\]

mientras que la evaluación con estrategia outermost termina en sólo 2 pasos:

\[
\begin{align*}
\text{answer} & \rightarrow \text{first}(0,\text{loop}) \rightarrow 0.
\end{align*}
\]
Por los ejemplos anteriores, podría llegar a pensarse que la estrategia outermost es siempre más eficiente que la innermost. Pero no es cierto:

**Ejemplo:** Dado el siguiente programa:

\[
double(X) = X + X
\]

la evaluación outermost de la expresión \( double(sqr(3)) \) empieza con:

\[
double(sqr(3)) \rightarrow sqr(3) + sqr(3) \rightarrow
\]

que requerirá la evaluación de la expresión \( sqr(3) \) dos veces!
Resumen

1. Ni la estrategia de reducción innermost es superior en eficiencia a la outermost, ni viceversa.

2. Si el programa es canónico, ambas estrategias son completas y calculan necesariamente el mismo valor (aunque puede variar el número de pasos invertido por cada una.)

3. Si el programa no es terminante, para algunas expresiones la estrategia outermost puede producir una respuesta aún si la estrategia innermost no termina.

4. La estrategia outermost tiene la propiedad importante de que, para cualquier término, si existe una estrategia de reducción que termina calculando la forma normal del término, entonces existe una secuencia de reducciones outermost que termina computando este valor (se dice que es la estrategia outermost es safe).
Estrategia **outermost** = Estrategia **lazy**
(perezosa)
= Orden *normal*

Estrategia **innermost** = Estrategia **eager**
(impatiente, voraz,
    ansiosa)
= Orden *aplicativo*
Narrowing strategies

Two kinds of strategies:

• **Eager narrowing**: (basic, innermost, ...)
  
  - termination is required
  
  - complete w.r.t *standard equality*
  
  - applications: automated deduction, specification languages, ...

• **Lazy narrowing**: (needed, demand-driven, ...)
  
  - syntactic criterion to ensure confluence
  
  - complete w.r.t. *strict equality*
  
  - optimal for inductively sequential programs
# Narrowing strategies

<table>
<thead>
<tr>
<th>Narrowing Strategy</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple narrowing</td>
<td>Fay’75, Slagle’74</td>
</tr>
<tr>
<td>basic narrowing</td>
<td>Hullot’80</td>
</tr>
<tr>
<td>left-to-right basic narrowing</td>
<td>Herold’86</td>
</tr>
<tr>
<td>LSE narrowing</td>
<td>Bockmayr et al’92</td>
</tr>
<tr>
<td>selection narrowing</td>
<td>Bosco et al’88</td>
</tr>
<tr>
<td>innermost narrowing</td>
<td>Fribourg’85</td>
</tr>
<tr>
<td>innermost basic narrowing</td>
<td>Holldobler’89</td>
</tr>
<tr>
<td>normalizing narrowing</td>
<td>Fay’79</td>
</tr>
<tr>
<td>normalizing innermost narrowing</td>
<td>Fribourg’85</td>
</tr>
<tr>
<td>normalizing basic narrowing</td>
<td>Nutt et al’89, Réty’87</td>
</tr>
<tr>
<td>normalizing innermost basic narrowing</td>
<td>Holldobler’89</td>
</tr>
<tr>
<td>outermost narrowing</td>
<td>Echahed’88</td>
</tr>
<tr>
<td>outer narrowing</td>
<td>You’89</td>
</tr>
<tr>
<td>lazy narrowing</td>
<td>Darlington’89, Moreno’90, Reddy’85</td>
</tr>
<tr>
<td>needed narrowing</td>
<td>Antoy et al’94</td>
</tr>
<tr>
<td>conditional narrowing</td>
<td>Hussman’85, Kaplan’84</td>
</tr>
<tr>
<td>basic conditional narrowing</td>
<td>Middeldorp et al’92</td>
</tr>
<tr>
<td>innermost basic conditional narrowing</td>
<td>Holldobler’89</td>
</tr>
<tr>
<td>abstract conditional narrowing</td>
<td>Alpuente et al’92</td>
</tr>
<tr>
<td>compositional basic cond. narrowing</td>
<td>Alpuente et al’94</td>
</tr>
</tbody>
</table>
Basic Conditional Narrowing:

Basic narrowing rule

\[
\begin{align*}
\text{Basic narrowing rule} & : \\
\forall u \in O_{nv}(g) \land (l \rightarrow r \leftarrow C) \bullet \exists R^+ \land \sigma = \text{mgu}([g_u \theta = l]) \\
\langle g, \theta \rangle \Rightarrow \langle (C, g[r]_u), \theta \sigma \rangle
\end{align*}
\]

or

Basic narrowing rule (with Occ)

\[
\begin{align*}
\text{Basic narrowing rule (with Occ)} & : \\
\forall u \in \text{Occ} \land (l \rightarrow r \leftarrow C) \bullet \exists R^+ \land \sigma = \text{mgu}([g_u \theta = l]) \land \\
\text{Occ}' & = O_{nv}(C) \cup ([|C|*(\text{Occ} - \{v \in \text{Occ} \mid u \leq v\}) \cup \{u.v \mid v \in O_{nv}(r)\}) \\
\langle g, \text{Occ} \rangle \Rightarrow_{\sigma} \langle (C, g[r]_u)\sigma, \text{Occ}' \rangle
\end{align*}
\]

This refinement is complete for decreasing and confluent 1-CTRSs

and for level-canonical 2-CTRSs

(but incomplete for level-canonical 3-CTRSs, whereas full conditional narrowing is complete in that case)
Innermost Conditional Narrowing:

Innermost narrowing rule

\[ u \in \varphi(g) \& (l \rightarrow r \leftarrow C) \eweak R^+ \& \sigma = \text{mgu}(\{ g[u=l] \}) \]

\[ \langle g, \theta \rangle \Rightarrow \langle C, g[r_u] \sigma, \theta \sigma \rangle \]

where \( \varphi(g) \) selects an innermost position of \( g \) (e.g. the leftmost innermost position).

This refinement is complete for CB-CD canonical 1-CTRSs w.r.t. ground solutions \( \sigma \) s.t. \( \text{Var}(g) \subseteq \text{Dom}(\sigma) \)

Note that all answers are normalized!!!
(since they are constructor)
Selection Narrowing:

Innermost basic narrowing rule

\[ u \in \varphi(g) \land (l \rightarrow r \leftarrow C) \land \sigma = \text{mgu}(\{g|_u \theta = l\}) \]

\[<g, \theta> \Rightarrow <(C, g[r]_u), \theta \sigma>\]

Innermost reflexion rule

\[ u \in \varphi(g) \land \sigma = \{x/(g|_u \theta)\}\]

\[<g, \theta> \Rightarrow <(C, g[x]_u), \theta \sigma>\]

where \( \varphi(g) \) selects an innermost position of \( g \) (e.g. the leftmost innermost position).

This refinement is complete for decreasing and confluent 1-CTRSs

and for level-canonical 2-CTRSs
Languages:

1. Flattening: EUROPA, K-LEAF
2. SLDE-resolution: CLP(H/E), LPG
3. SLD+Residuation: LeFun, Oz
   SLD+Narrowing: Alf, SLOG
   (innermost basic, innermost)
4. Narrowing: Babel (lazy, demand-driven)
   Residuation: Escher
   Narrowing + Residuation: Curry (needed)
Compositionality
\[ S(c_1 \blacklozenge c_2) = S(c_1) \heartsuit S(c_2) \]

SLD is compositional for:
- ♦ conjunction of subgoals &
- ♥ parallel composition of substitutions \( \uparrow \)

\[
\sigma \uparrow \theta = \text{mgu}(eq(\sigma) \cup eq(\theta))
\]

\[
\Theta_1 \uparrow \Theta_2 = \bigcup_{\theta_1 \in \Theta_1 \& \theta_2 \in \Theta_2} \theta_1 \uparrow \theta_2
\]

[Alpuente, Falaschi & Vidal]:

• Cond. narrowing is not compositional\(^3\)
• Basic cond. narrowing is compositional
  (so innermost, lazy, etc also are)

\[
\begin{align*}
\text{f(0)} &= 0 \\
f(f(0)) &= f(0)
\end{align*}
\]

<table>
<thead>
<tr>
<th>( \text{goal} ) ?- 0=X, f(X)=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(0=X) = { {X/0} } )</td>
</tr>
<tr>
<td>( O(f(X)=0) = { {X/0}, {X/f(0)} } )</td>
</tr>
<tr>
<td>( O(0=X) \uparrow S(f(X)=0) = { {X/0} } \neq )</td>
</tr>
<tr>
<td>( O(0=X &amp; f(X)=0) = { {X/0}, {X/f(0)} } )</td>
</tr>
</tbody>
</table>

\(^3\) Compositionality of conditional narrowing holds w.r.t. normalized substitutions
Compositional basic cond. narrowing

**Simple Basic Narrowing**

\[
\begin{align*}
\forall u \in O_{nv}(e) & \land (l = r \Leftarrow C) \land R \land \sigma = \text{mgu} \{ e_u \theta, l \} \\
\hline \\
\langle e, \theta \rangle & \Rightarrow \langle (C, e[r]_u), \theta \sigma \rangle \\
\end{align*}
\]

**Splitting**

\[
\begin{align*}
\langle g_1, \theta \rangle & \Rightarrow \langle g'_1, \theta \theta_1 \rangle \quad \land \\
\langle g_2, \theta \rangle & \Rightarrow \langle g'_2, \theta \theta_2 \rangle \quad \land \\
\theta_1 \uparrow \theta_2 \neq \text{fail} \\
\hline \\
\langle (g_1, g_2), \theta \rangle & \Rightarrow \langle (g'_1, g'_2), \theta(\theta_1 \uparrow \theta_2) \rangle \\
\end{align*}
\]

**Semantics**

\[
O'(g) = \{ \theta |_{\text{Var}(g)} \mid \langle g, \varepsilon \rangle \Rightarrow^* \langle \text{true}, \theta \rangle \}
\]

- \( O'(g) = O(g) \)
- \( O'(g_1, g_2) = O'(g_1) \uparrow O'(g_2) \)
Needed Narrowing:

- **Sound & complete** w.r.t. the *strict* equality for *inductively sequential* programs:

  the rules for each operation can be organized in a *definitional tree* (e.g. rules with linear patterns which do not overlap)

- **Optimality:**

  1. all steps are *needed*
  2. derivations are shorter
     (in graphs, minimal length)
  3. Independence of solutions (σ y σ’ do not unify)
Curry: a modern functional logic language

- conditional rules
- equational constraints
- concurrency
- higher order functions
- modules
- polymorphic type system
- declarative (monadic) I/O
- encapsulate search
- external functions
- external constraint solvers

---

- **LeFun**: predicates $\rightarrow$ *SLD-resolution*
  functions $\rightarrow$ *residuation*

- **Curry**: predicates (boolean functions) $\rightarrow$ *needed narrowing*
  functions (other functions) $\rightarrow$ *residuation*

---

More Infos on Curry:

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

- Functional programming: values of expressions
- Logic programming: computed answers

Answer expression: $\sigma [] e$ (substitution + expression)
$\sigma [] e$ solved: e does not contain defined functions

Disjunctive expression: $\{\sigma_1 [] e_1, \ldots, \sigma_n [] e_n\}$
also written as: $(\sigma_1 [] e_1) \lor \ldots \lor (\sigma_n [] e_n)$

Computation step: reduce an unsolved expression
Deterministic step: replace expression by new expression
Non-deterministic step: replace expression by disjunction of new expressions

\[
\begin{align*}
p(a) &= \text{true} \\
p(b) &= \text{true}
\end{align*}
\]

\[
p(a) \Rightarrow \text{true}
\]

\[
p(X) \Rightarrow (X=a [] \text{true}) \lor \ldots \lor (X=b [] \text{true})
\]
Extensions:

* **monadic I/O**
  - declarative I/O
  - I/O: transformation of the outside world
  - no I/O within nondeterministic sections of the program: encapsulate search

* **encapsulate search**
  - technique to avoid global search (no backtrackable I/O, efficiency control)
  - select one solution in a disjunction
  - implementation by deep guards and committed choice

* **constraints**
  - interface to specialized constraint solvers
  - arithmetics (built-in constraints)
Why integration of Declarative Paradigms?

• more expressive than pure functional languages (compute with partial information)

• functions: declarative notion to improve control in logic languages
  
  - more structural information (functional dependencies)

  - avoid impure features of Prolog

  - more efficient than logic programs (determinism, lazyness)

• combine research efforts in FP and LP

• a unified paradigm for teaching declarative programming

• optimizations of FLP: analysis and transformation
E-Unification Theory

- equational theory $E$
- equational problem $s = t$

A (possibly infinite) set of substitutions $\Sigma$ is a correct, complete, minimal set of $E$-unifiers w.r.t. $E$ and $s = t$ if:

- correct: $\forall \sigma \in \Sigma, \ s\sigma =_{E} t\sigma$
- complete: $\forall \theta. \ s\theta =_{E} t\theta$, $\exists \sigma \in \Sigma. \ \sigma \leq_{E} \theta$
- minimal: $\forall \sigma, \gamma \in \Sigma. \ (\sigma \neq \gamma) \Rightarrow \neg(\sigma \leq_{E} \gamma) \land \neg(\gamma \leq_{E} \sigma)$

When $\Sigma$ satisfies all conditions, it is called a set of more general $E$-unifiers ($E$-mgus) of $s = t$.

In this case, $\Sigma$ is unique modulo variable renaming.
E-Unification is undecidable

- Are there algorithms for generating sets of E-unifiers for a given problem \( s=t \) which are correct, complete and minimal whenever such a set exists? **No**

- do these algorithms terminate whenever the set of E-mgu’s exists and is finite? **No**
Examples

\[ E = \{ f(X, Y) = f(Y, X) \} \quad \text{Finite n. of E-mgu's} \]

\[ s = f(g(X), Y) \quad \text{E-mgu}_1 = \{ X/a, Y/g(Z) \} \]
\[ t = f(g(a), g(Z)) \quad \text{E-mgu}_2 = \{ X/Z, Y/g(a) \} \]

\[ E = \{ (X.Y).Z = X.(Y.Z) \} \quad \text{∞ n. of E-mgu's} \]

\[ s = X.a \quad \text{E-mgu}_1 = \{ X/a \} \]
\[ t = a.X \quad \text{E-mgu}_2 = \{ X/a.a \} \]
\[ \text{E-mgu}_3 = \{ X/a.a.a \} \quad \text{∞} \]

\[ E = \{ f(1, X) = X \quad \text{A complete and minimal} \]
\[ g(f(X, Y)) = g(Y) \} \quad \text{set of E-mgu's doesn’t exist, even} \quad \text{∞} \]

\[ s = g(X) \quad \theta_1 = \{ X/f(1, a) \} \]
\[ t = g(a) \quad \theta_2 = \{ X/f(Z, f(1, a)) \} \]
\[ \theta_3 = \{ X/f(Z, f(Z', f(1, a))) \} \quad \text{∞} \]

• This set would be complete, but it is not minimal since there are dependent substitutions. For instance \[ \theta_2 \leq_E \theta_1 \] (since \[ \theta_1 =_{E} \theta_2. \{ Z/1 \} \])

• Since the last element of the chain is the more general one, the other ones should be dropped: ∞!

• Unfortunately, the order \[ \leq_E \] is not well-founded
Hierarchy of Equational Theories

1 Unitary
(E.g. \( E = \emptyset \))

\( \omega \) Finitary
(Finite n. of E-mgu's)
(E.g. \( E = \text{CONM} \))

\( \infty \) Infinitary
(\( \infty \) n. of E-mgu's)
(E.g. \( E = \text{ASSOC} \))

0 Nullary
(A set of E-mgu’s doesn’t exist, even \( \infty \))
(E.g. \( E = \text{ASSOC + Id} \))
Summary of (negative) results

- E-unification is **semidecidable**

- **A complete set of E-unifiers:**

  - always exists, although it *might not be finite or minimal*

  - if it is finite  $\Rightarrow$ a complete and **minimal** set of E-mgu’s does exist! (by dropping the redundant elements)

  - if it is not finite  $\Rightarrow$ a complete and **minimal** set of E-mgu’s *might not exist* (even $\infty$)

  - if a complete and minimal set of E-mgu’s exists  
    *(finite or infinite)*  $\Rightarrow$ it is unique (modulo variable renamig)

  - even if a complete and minimal set exists, E-unification algorithms *might not produce a minimal set* (even if a finite set E-mgu’s does exist!)

  - even if a **finite, complete and minimal** set of E-unifiers exists, E-unification algorithms *might not terminate*
Counterexamples

\[
\begin{align*}
  f(0) &= 0 \\
  f(f(X)) &= f(X)
\end{align*}
\]

\textbf{goal} ?- f(X) = 0

\[
\begin{align*}
  X &= 0 \\
  X &= f(0) \\
  X &= f(f(0)) \\
  &\vdots
\end{align*}
\]

\[
\begin{align*}
  f(0) &= 0 \\
  f(f(X)) &= s(f(X))
\end{align*}
\]

\textbf{goal} ?- f(X) = 0

\[
\begin{align*}
  X &= 0 \\
  \infty
\end{align*}
\]
A positive result

Conditional narrowing computes a \textit{complete set of E-unifiers} for programs E which satisfy different conditions.

Unfortunately, the set \textit{might not be minimal}, even if a finite, \textit{complete} and \textit{minimal} set of E-mgu’s exists, see above :-(

Also, the algorithm \textit{may loop} after computing the set of solutions (even if this set is finite), see above:-(

Abstract narrowing

\[ u \in O_{\text{nv}}(g) \land (l = r \leftarrow C) \triangleleft R^A \land \kappa = \text{mgu}^A\{g|u,l\} \]

\[ g \Rightarrow^A_{\kappa} (C, g[r]u)_{\kappa} \]

— abstract semantics of a goal \( g \)

\[ \Delta(g) = \{\kappa \mid g \Rightarrow^A_{\kappa} \text{true} \} \quad \text{(decidable)} \]

— the abstract semantics approximates the concrete semantics

\[ \forall \sigma \in O(g). \exists \kappa \in \Delta(g). \kappa \leq \sigma \]

— unsatisfiability analysis

\[ \Delta(g) = \emptyset \Rightarrow g \text{ es unsatisfiable} \]

— abstract basic narrowing is compositional

\[ \Delta(g_1,g_2) = \Delta(g_1) \uparrow^A \Delta(g_2) \]
Multi-Paradigm Programming

Michael Hanus

Christian-Albrechts-Universität Kiel

Extend functional languages with features for

1. logic (constraint) programming
2. object-oriented programming
3. concurrent programming
4. distributed programming
General idea:

- no coding of algorithms
- description of logical relationships
- powerful abstractions
  - domain specific languages
- higher programming level
- reliable and maintainable programs
  - pointer structures $\Rightarrow$ algebraic data types
  - complex procedures $\Rightarrow$ comprehensible parts
    (pattern matching, local definitions)
DECLARATIVE PROGRAMMING: PARADIGMS

Functional programming:

- functions, $\lambda$-calculus
- equations
- (lazy) deterministic reduction

Logic programming:

- predicates, predicate logic
- logical formulas, Horn clauses
- constraint solving (unification)
- non-deterministic search for solutions
FUNCTIONAL LOGIC LANGUAGES

- efficient execution principles of functional languages
- flexibility of logic languages
- avoid non-declarative features of Prolog (arithmetic, I/O, cut)
- combine best of both worlds in a single model
  ➔ higher-order functions  ~  design patterns
  ➔ declarative I/O
  ➔ concurrent constraints
Readability, safety:

function fac(n: nat): nat =
begin
  z := 1;  p := 1;
  while z<n+1 do
    begin p := p*z;  z := z+1 end;
  return(p)
end

\[
\begin{align*}
  \text{fac 0} & = 1 \\
  \text{fac (n+1)} & = (n+1) \times (\text{fac n})
\end{align*}
\]
Quicksort: Classical imperative version:

```plaintext
procedure qsort(l,r: index);
var i,j: index; x,w: item
begin
    i := l;  j := r;
    x := a[(l+r) div 2];
    repeat
        while a[i] < x do i := i+1;
        while x < a[j] do j := j-1;
        if i <= j then
            begin w := a[i]; a[i] := a[j]; a[j] := w;
                i := i+1; j := j-1
            end
        until i > j;
    if l < j then qsort(l,j);
    if i < r then qsort(i,r);
end
```
Quicksort: Classical imperative version:

```plaintext
procedure qsort(l,r: index);
var i,j: index; x,w: item
begin
  i := 1;  j := r;
  x := a[(l+r) div 2];
  repeat
    while a[i] < x do i := i+1;
    while x < a[j] do j := j-1;
    if i <= j then
      begin w := a[i]; a[i] := a[j]; a[j] := w;
          i := i+1; j := j-1
      end
    until i > j;
  if l < j then qsort(l,j);
  if i < r then qsort(i,r);
end
```

Declarative version:

```plaintext
qsort [] = []
qsort (x:l) =
  qsort (filter (<x) l)
  ++ [x]
  ++ qsort (filter (>=x) l)
```
Program development and maintenance:

function f(n: nat): nat =
begin
    write('Hello');
    return(n*n)
end

... z:=f(3)*f(3) ...

Optimization: ... x:=f(3); z:=x*x ... (?)

side effects complicate program optimization and transformation
As a language for concrete examples, we use Curry:
[Dagstuhl’96, POPL’97]

- multi-paradigm language
- extension of Haskell (non-strict functional language)
- developed by an international initiative
- provide a standard for functional logic languages (research, teaching, application)
- several implementations available
Values in imperative languages: basic types + pointer structures

Declarative languages: **algebraic data types** (Haskell-like syntax)

\[
\begin{align*}
\text{data } \text{Bool} & \quad = \text{True} \mid \text{False} \\
\text{data } \text{Nat} & \quad = \text{Z} \mid \text{S Nat} \\
\text{data } \text{List } a & \quad = [] \mid a : \text{List } a \quad -- \quad [a] \\
\text{data } \text{Tree } a & \quad = \text{Leaf } a \mid \text{Node } [\text{Tree } a] \\
\text{data } \text{Int} & \quad = 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots
\end{align*}
\]

**Value** $\cong$ **data term, constructor term:**
well-formed expression containing variables and data type constructors

\[
\begin{align*}
(S \ Z) & \quad 1 : (2 : []) \quad [1,2] \quad \text{Node } [\text{Leaf } 3, \text{Node } [\text{Leaf } 4, \text{Leaf } 5]]
\end{align*}
\]
**FUNCTIONAL PROGRAMS**

**Functions**: operations on values defined by equations (or rules)

\[ f \ t_1 \ \ldots \ t_n \mid c = r \]

- **defined operation**: \( f \)
- **data terms**: \( t_1, \ldots, t_n \)
- **condition (optional)**: \( c \)
- **expression**: \( r \)

\[
\begin{align*}
Z + y &= y & Z \leq y &= \text{True} \\
(S \ x) + y &= S(x+y) & (S \ x) \leq Z &= \text{False} \\
(S \ x) \leq (S \ y) &= x \leq y \\
\text{[]} + y &= y \\
(x:\text{xs}) + y &= x : (\text{xs} + y) \\
\text{depth (Leaf _)} &= 1 \\
\text{depth (Node [])} &= 1 \\
\text{depth (Node (t:ts))} &= \max (1 + \text{depth } t) (\text{depth (Node ts)})
\end{align*}
\]
Reduce expressions to their values

Replace equals by equals

Apply reduction step to a subterm (redex, reducible expression):
variables in rule’s left-hand side are universally quantified

~ match lhs against subterm (instantiate these variables)

\[
\begin{align*}
Z + y &= y & Z \leq y &= \text{True} \\
(S \ x) + y &= S(x+y) & (S \ x) \leq Z &= \text{False} \\
(S \ x) \leq (S \ y) &= x \leq y \\
\end{align*}
\]

\[
(S \ Z)+(S \ Z) \rightarrow S \ (Z+(S \ Z)) \rightarrow S \ (S \ Z)
\]
**Evaluation Strategies**

Expressions with several redexes: which evaluate first?

**Strict evaluation**: select an innermost redex ($\approx$ call-by-value)

**Lazy evaluation**: select an outermost redex

\[
\begin{align*}
Z + y &= y \\
(S \ x) + y &= S(x+y) \\
Z \leq y &= \text{True} \\
(S \ x) \leq Z &= \text{False} \\
(S \ x) \leq (S \ y) &= x \leq y
\end{align*}
\]

**Strict evaluation:**
\[
Z \leq (S \ Z) + (S \ Z) \rightarrow Z \leq (S \ (Z + (S \ Z))) \rightarrow Z \leq (S \ (S \ Z)) \rightarrow \text{True}
\]

**Lazy evaluation:**
\[
Z \leq (S \ Z) + (S \ Z) \rightarrow \text{True}
\]
Strict evaluation might need more steps, but it can be even worse…

\[
\begin{align*}
Z + y &= y \\
(S \ x) + y &= S(x+y) \\
(S \ x) \leq Z &= \text{False} \\
(S \ x) \leq (S \ y) &= x \leq y
\end{align*}
\]

\[f = f\]

Lazy evaluation:
\[Z + Z \leq f \rightarrow Z \leq f \rightarrow \text{True}\]

Strict evaluation:
\[Z + Z \leq f \rightarrow Z + Z \leq f \rightarrow Z + Z \leq f \rightarrow \cdots\]

Ideal strategy: evaluate only needed redexes
(i.e., redexes necessary to compute a value)

Determine needed redexes with definitional trees
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

\[
\begin{array}{c}
\mathbf{x}_1 \leq x_2 \\
\downarrow \\
\begin{array}{c}
Z \leq x_2 \\
\text{True}
\end{array}
\end{array}
\begin{array}{c}
(S \ x_3) \leq x_2 \\
\downarrow \\
\begin{array}{c}
(S \ x_3) \leq Z \\
\text{False}
\end{array}
\end{array}
\begin{array}{c}
(S \ x_3) \leq (S \ x_4) \\
\downarrow \\
x_3 \leq x_4
\end{array}
\begin{array}{c}
Z \leq y = \text{True} \\
(S \ x) \leq Z = \text{False} \\
(S \ x) \leq (S \ y) = x \leq y
\end{array}
\]
Evaluating function call $t_1 \leq t_2$:

1. Reduce $t_1$ to head normal form (constructor-rooted expression)
2. If $t_1 = Z$: apply rule
3. If $t_1 = (S\ldots)$: reduce $t_2$ to head normal form
Properties of Reduction with Definitional Trees

- **Normalizing strategy**
  i.e., always computes value if it exists \( \approx \) sound and complete

- Independent on the order of rules

- Definitional trees can be automatically generated
  \( \rightarrow \) pattern matching compiler

- Identical to lazy functional languages (e.g., Miranda, Haskell) for the subclass of **uniform** programs
  (i.e., programs with strong left-to-right pattern matching)

- **Optimal strategy:** each reduction step is needed

- Easily extensible to more general classes
Functions are first class citizens

- passing functions as parameters and results
- combinator-oriented programming
- expressing design patterns
- code reuse

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{map} f [] = []
\]
\[
\text{map} f (x:xs) = (f x) : \text{map} f xs
\]

\[
\text{map} (1 +) [2,3,4] \rightarrow [3,4,5]
\]

Partial application: \((1 +)\) is a function of type \(\text{Int}\rightarrow\text{Int}\)

\(\lambda\)-abstraction: \(\lambda x \rightarrow 1 + x\) (anonymous function)
Accumulate list elements with a binary operator:

\[
\begin{align*}
\text{foldr } f \ z \ [] &= z \\
\text{foldr } f \ z \ (x:xs) &= f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}
\]

Multiply all list elements: \( \text{foldr} \ (\ast) \ 1 \ xs \)

Concatenate a list of lists: \( \text{concat} \ xs = \text{foldr} \ (+) \ [] \ xs \)

Tree example: computing list of all leaves in a tree:

\[
\begin{align*}
\text{frontier} :: \text{Tree } a &\rightarrow [a] \\
\text{frontier} \ (\text{Leaf} \ v) &= [v] \\
\text{frontier} \ (\text{Node} \ ns) &= \text{concat} \ (\text{map} \ \text{frontier} \ ns)
\end{align*}
\]
Filter all elements in a list satisfying a given predicate:

\[
\begin{aligned}
\text{filter} &:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\text{filter} \ p \ [] & = [] \\
\text{filter} \ p \ (x:xs) & = \text{if} \ p \ x \ \text{then} \ x : \text{filter} \ p \ xs \\
& \quad \text{else filter} \ p \ xs
\end{aligned}
\]

Now the code for quicksort becomes straightforward:

\[
\begin{aligned}
\text{qsort} \ [] & = [] \\
\text{qsort} \ (x:l) & = \text{qsort} \ (\text{filter} \ (<x) \ l) \\
& \quad ++ \ [x] ++ \text{qsort} \ (\text{filter} \ (> \ =x) \ l)
\end{aligned}
\]
Data type for representing HTML expressions:

```
data HtmExp = HText String
            | HStruct String [(String,String)] HtmExp
```

```
HStruct "A" [("HREF","http://..."),] [HText "click here"]
```

Get all hypertext links in an HTML document:

```
def hrefs [] = []
def hrefs (HText _ : hs) = hrefs hs
def hrefs (HStruct tag attrs shs : hs) =
  (if tag="A" then map snd (filter (\(t,_)-&gt;t="HREF") attrs)
   else [] ) ++ hrefs shs ++ hrefs hs
```
NON-DETERMINISTIC EVALUATION

Previous functions: inductively defined on data structures

Sometimes overlapping rules more natural:

\[
\begin{align*}
\text{True} \lor x &= \text{True} \\
x \lor \text{True} &= \text{True} \\
\text{False} \lor \text{False} &= \text{False}
\end{align*}
\]

First two rules overlap on \( \text{True} \lor \text{True} \)

\( \sim \) Problem: no needed argument: \( e_1 \lor e_2 \) evaluate \( e_1 \) or \( e_2 \)?

Functional languages: backtracking: Evaluate \( e_1 \), if not successful: \( e_2 \)

Disadvantage: not normalizing (\( e_1 \) may not terminate)
NON-DETERMINISTIC EVALUATION

True \lor x = True
x \lor True = True
False \lor False = False

Evaluation of $e_1 \lor e_2$?

1. Parallel reduction of $e_1$ and $e_2$ [Sekar/Ramakrishnan 93]

2. **Non-deterministic reduction**: try *(don’t know)* $e_1$ or $e_2$

Extension to definitional trees / pattern matching:
Introduce *or-nodes* to describe non-deterministic selection of redexes

$\leadsto$ non-deterministic evaluation: $\ e \ \rightarrow \ e_1 \ | \cdots \ | \ e_n \ \\
\hline
$\leadsto$ non-deterministic functions
Functions can have more than one result value:

```
choose x y = x
choose x y = y
```

```
choose 1 2 → 1 | 2
```

Non-deterministic list insertion and permutations:

```
insert x [] = [x]
insert x (y:ys) = choose (x:y:ys) (y:insert x ys)
permute [] = []
permute (x:xs) = insert x (permute xs)
```

```
permute [1,2,3] →
[1,2,3] | [2,1,3] | [2,3,1] | [1,3,2] | [3,1,2] | [3,2,1]
```
**Logic Programming**

Distinguished features:

- compute with partial information *(constraints)*
- deal with *free variables* in expressions
- compute *solutions* to free variables
- built-in search
- non-deterministic evaluation

**Functional programming:** *values*, no free variables

**Logic programming:** *computed answers* for free variables

**Operational extension:** instantiate free variables, if necessary
Evaluate \((f \ x)\): bind \(x\) to 0 and reduce \((f \ 0)\) to 2, or:

- bind \(x\) to 1 and reduce \((f \ 1)\) to 3

**Computation step:** \(\text{bind} \quad \text{reduce} : \quad e \leadsto \{\sigma_1\} \ e_1 \ | \cdots | \{\sigma_n\} \ e_n\)

**Reduce:**

\[
(f \ 0) \leadsto 2
\]

**Bind and reduce:**

\[
(f \ x) \leadsto \{x=0\} \ 2 \ | \ \{x=1\} \ 3
\]

Compute necessary bindings with needed strategy

\[
\leadsto \text{needed narrowing} \quad [\text{Antoy/Echahed/Hanus POPL'94/JACM'00}]
\]
Evaluating function call $t_1 \leq t_2$:

1. Reduce $t_1$ to head normal form
2. If $t_1 = Z$: apply rule
3. If $t_1 = (S \ldots)$: reduce $t_2$ to head normal form
Evaluating function call $t_1 \leq t_2$:

1. Reduce $t_1$ to head normal form
2. If $t_1 = Z$: apply rule
3. If $t_1 = (S \ldots)$: reduce $t_2$ to head normal form
4. If $t_1$ variable: bind $t_1$ to $Z$ or $S$ $x$
Properties of Needed Narrowing

Sound and complete (w.r.t. strict equality, no termination requirement)

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. Minimal set of computed solutions:
   Two solutions $\sigma$ and $\sigma'$ computed by two distinct derivations are independent.
**Properties of Needed Narrowing**

**Determinism:**
No non-deterministic step during the evaluation of ground expressions
\( \approx \) functional programming

**Restriction:** *inductively sequential rules*
(i.e., no overlapping left-hand sides)

Extensible to

- conditional rules [Hanus ICLP’95]
- overlapping left-hand sides [Antoy/Echahed/Hanus ICLP’97]
- multiple right-hand sides [Antoy ALP’97]
- concurrent evaluation [Hanus POPL’97]
Problems with equality in the presence of non-terminating rules:

1. Equality on infinite objects undecidable:

\[
\begin{align*}
\text{f} & = 0: \text{f} \\
\text{g} & = 0: \text{g}
\end{align*}
\]

Is \( \text{f} = \text{g} \) valid?

2. Semantics of non-terminating functions:

\[
\begin{align*}
\text{f} \ x & = \text{f} (x+1) \\
\text{g} \ x & = \text{g} (x+1)
\end{align*}
\]

Is \( \text{f} \ 0 = \text{g} \ 0 \) valid?

Avoided by strict equality: identity on finite objects
(both sides reducible to same ground data term)
Logic programming: solve goals, compute solutions

Functional logic programming: solve equations

Strict equality: only reasonable notion of equality in the presence of non-terminating functions

Equational constraint $e_1 =: e_2$

satisfied if both sides evaluable to unifiable data terms

$\Rightarrow e_1 =: e_2$ does not hold if $e_1$ or $e_2$ undefined or infinite

$\Rightarrow e_1 =: e_2$ and $e_1, e_2$ data terms $\approx$ unification in logic programming
List concatenation:

\[
\begin{align*}
\text{append} &:: [a] \rightarrow [a] \rightarrow [a] \\
\text{append} \ [\] &\ = ys \\
\text{append} \ (x:xs) &\ = x : \text{append} \ xs \ ys
\end{align*}
\]

Functional programming:

\[
\begin{align*}
\text{append} \ [1,2] \ [3,4] &\ \leadsto \ [1,2,3,4]
\end{align*}
\]

Logic programming:

\[
\begin{align*}
\text{append} \ x \ y &\ =: [1,2] \ \leadsto \\
\{ x=[\], y=[1,2] \} &\ | \ \{ x=[1], y=[2] \} \ | \ \{ x=[1,2], y=[] \}
\end{align*}
\]

Last list element:

\[
\begin{align*}
\text{last} \ xs \ | \ \text{append} \ ys \ [x] &\ =: xs = x
\end{align*}
\]
Infinite list of natural numbers:

```
from x = x : from (S x)
first Z   ys = []
first (S x) (y:ys) = y : first x ys
```

Lazy functional programming:

```
first (S (S Z)) (from Z) \rightarrow [Z,(S Z)]
```

Lazy functional logic programming:

```
first x (from y) =: [Z] \rightarrow \{x=(S Z), y=Z\}
```
Non-deterministic functions for generating permutations:

$$\begin{align*}
\text{insert } x \; [] &= [x] \\
\text{insert } x \; (y:ys) &= \text{choose} \; (x:y:ys) \; (y:\text{insert } x \; ys) \\
\text{permute } [] &= [] \\
\text{permute } (x:xs) &= \text{insert } x \; (\text{permute } xs)
\end{align*}$$

Sorting lists with test-of-generate principle:

$$\begin{align*}
\text{sorted } [] &= [] \\
\text{sorted } [x] &= [x] \\
\text{sorted } (x:y:ys) \mid x \leq y &= x : \text{sorted } (y:ys) \\
\text{psort } xs &= \text{sorted } (\text{permute } xs)
\end{align*}$$
Advantages of non-deterministic functions as generators:

- demand-driven generation of solutions (due to laziness)
- modular program structure

\[ \text{psort} \left[ 5, 4, 3, 2, 1 \right] \leadsto \text{sorted} \left( \text{permute} \left[ 5, 4, 3, 2, 1 \right] \right) \]

\[ \leadsto^* \text{sorted} \left( 5 : 4 : \text{permute} \left[ 3, 2, 1 \right] \right) \mid \ldots \]

**undefined**: discard this alternative

**Effect:** Permutations of \([3, 2, 1]\) are not enumerated!

**Permutation sort for** \([n, n-1, \ldots, 2, 1]\): \#or-branches/disjunctions

<table>
<thead>
<tr>
<th>Length of the list:</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate-and-test</td>
<td>24</td>
<td>120</td>
<td>720</td>
<td>40320</td>
<td>3628800</td>
</tr>
<tr>
<td>test-of-generate</td>
<td>19</td>
<td>59</td>
<td>180</td>
<td>1637</td>
<td>14758</td>
</tr>
</tbody>
</table>
How to deal with non-deterministic computation steps?

→ explore alternatives in parallel ↠ parallel architectures
→ explore alternatives by backtracking ↠ Prolog
→ support flexible search strategies ↠ encapsulate search

Disadvantages of fixed search (like backtracking):

→ no application-dependent strategy or efficiency control
→ global search: local search has global effects
→ I/O operations not backtrackable
→ problems with concurrency and backtracking

Solution: provide primitives for user-definable search strategies
(Oz [Schulte/Smolka 94], Curry [Hanus/Steiner 98])
Encapsulated Search

Idea:
Compute until a non-deterministic step occurs, then give programmer control over this situation

Search:

→ solve constraint
→ evaluate until failure, success, or non-determinism
→ return result in a list

First approach to primitive search operator:

\[ \text{try} :: \text{Constraint} \rightarrow [\text{Constraint}] \]
try :: Constraint -> [Constraint]

f 0 = 2
f 1 = 3

try (1::=2) \rightarrow [] \quad \text{failure}

try ([x]::=[0]) \rightarrow [x::=0] \quad \text{success}

try (f x ::= 3) \rightarrow [x::=0 & f 0 ::= 3, x::=1 & f 1 ::= 3] \quad \text{disjunction}

Problem: incompatible bindings for \( x \) in disjunctions!

Solution: abstract search variable in constraints: \( x \mapsto c \)
Search goal: constraint with abstracted search variable

Search operator try: maps search goal into list of search goals

\[\text{try :: (a->Constraint) -> [a->Constraint]}\]

\[f \ 0 = 2\]
\[f \ 1 = 3\]

\[\text{try } \ x -> 1 \ =\ :=\ 2 \quad \leadsto \ [\] \quad \text{failure}\]
\[\text{try } \ x -> [x] \ =\ :=\ [0] \quad \leadsto \ [\ x -> x \ =\ :=\ 0] \quad \text{success}\]
\[\text{try } \ x -> f \ x \ =\ :=\ 3 \quad \leadsto \ [\ x -> x \ =\ :=\ 0 \ & \ f \ 0 \ =\ :=\ 3, \]
\[\ x -> x \ =\ :=\ 1 \ & \ f \ 1 \ =\ :=\ 3] \quad \text{disjunction}\]
try $x \to c$: evaluate $c$, stop after non-deterministic step

**Depth-first search:** collect all solutions in a list

```
all :: (a -> Constraint) -> [a -> Constraint]
all g = collect (try g)
where collect [] = []
collect [g] = [g]
collect (g1:g2:gs) = concat (map all (g1:g2:gs))
```

all ($\langle x s \to append \ x s \ [1] =:= [0,1] \rangle$) $\leadsto$ $[\langle x s \to x s =:= [0] \rangle]$
• compute only the first solution:

\[
\text{once } g = \text{head } (\text{all } g) \quad \text{where } \text{head } (x:xs) = x
\]

Note: \textit{lazy evaluation is important here!}

(strict languages, like Oz, must define new search operator)

\(\leadsto\) \textit{lazy evaluation supports better reuse}

• \texttt{findall}, best solution search, parallel search, …

• negation as failure:

\[
\text{naf } c = (\text{all } \_\rightarrow c) =:=[[]]
\]

\(\leadsto\) \textit{control failures}
Extract value of the search variable by application of search goal:

\[(\text{x} \rightarrow \text{x} = := 1) \text{ freevar } \sim \text{ freevar} = := 1 \sim \{ \text{freevar} = 1 \} \text{ success} \]

Prolog’s findall:

\[
\text{unpack} :: (a \rightarrow \text{Constraint}) \rightarrow a \\
\text{unpack g | g x = x where x free} \\
\text{findall g = map unpack (all g)}
\]

Compute all splittings of a list:

\[
\text{findall } (\text{x}, y) \rightarrow \text{ append x y = := [1,2]} \\
\* \Rightarrow ([[], [1,2]], ([1], [2]), ([1,2], []))
\]
Show a list of search goals, as requested by the user:

```haskell
printloop [] = putStrLn "no\n"
printloop (a:as) = browse a >>= putStrLn "? " >>=
                    getChar >>= evalAnswer as

evalAnswer as ';' = newline >>= printloop as
evalAnswer as '\n' = newline >>= putStrLn "yes\n"
```

Prolog's top-level: `prolog g = printloop (all g)`

```
prolog \(x,y) \rightarrow\ append x y =:= [1,2]

* ([], [1,2]) ? ;
(1, [2]) ? <-
   yes
prolog \x \rightarrow\ 1::=2

* no
```
Laziness easily supports demand-driven encapsulated search

⇒ **Separation of Logic and Control**

⇒ **Modularity:**

- Prolog’s top-level with breadth-first search:
  \[
  \text{prolog\_bfs } g = \text{printloop (bfs } g)\]

- Prolog’s top-level with depth-bounded search:
  \[
  \text{prolog\_bound } g \text{ bd } = \text{printloop (bound } g \text{ bd)}\]
Problem: Handling input/output in a declarative manner?

Solution: Consider the external world as a parameter to all I/O operations (Haskell, Mercury)

I/O actions: transformations on the external world

Interactive program: sequence(!) of actions applied to the external world

Type of I/O actions: \[ \text{IO } a \simeq \text{World } \rightarrow \text{(a,World)} \]

But: the “world” is implicit parameter, not explicitly accessible!
Some primitive I/O actions:

\[
\begin{align*}
\text{getChar} &: \text{IO Char} & \quad & \text{-- read character from stdin} \\
\text{putChar} &: \text{Char} \to \text{IO ()} & \quad & \text{-- write argument to stdout} \\
\text{return} &: \text{a} \to \text{IO a} & \quad & \text{-- do nothing and return argument}
\end{align*}
\]

getChar applied to a world \(\sim\) character + new (transformed) world

**Compose actions:** \((>>=)\) : \(\text{IO a} \to (\text{a} \to \text{IO b}) \to \text{IO b}\)

getChar \(>>=\) putChar: copy character from input to output

**Specialized composition:** ignore result of first action:

\[
(>\>) \quad : \quad \text{IO a} \to \text{IO b} \to \text{IO b}
\]

\[
x >\> y = x >>= \_ \to y
\]
Example: output action for strings (String ≈ [Char])

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (c:cs) = putChar c >> putStr cs
```

Example: read a line

```haskell
getLine :: IO String
getLine = getChar >>= \c ->
    if c=='\n' then return []
    else getline >>= \cs -> return (c:cs)
```
Monadic composition not well readable

\( \leadsto \) syntactic sugar: Haskell’s `do` notation

\[
do \ p \leftarrow a_1 \quad \approx \quad a_1 \gg= \ \backslash p \rightarrow a_2 \\
a_2
\]

Example: read a line (with `do` notation)

\[
\text{getLine} = \ do \ c \leftarrow \text{getChar} \\
\quad \begin{cases} 
\text{if } c == \text{’n’} \text{ then return } [] \\
\text{else do } cs \leftarrow \text{getLine} \\
\quad \text{return } (c : cs)
\end{cases}
\]

Note: no I/O in disjunctions (“cannot copy the world”)

\( \leadsto \) encapsulate search between I/O actions
Logic Programming:

- compute with partial information (constraints)
- data structures (constraint domain): constructor terms
- basic constraint: (strict) equality
- constraint solver: unification

Constraint Programming: generalizes logic programming by

- new specific constraint domains (e.g., reals, finite sets)
- new basic constraints over these domains
- sophisticated constraint solvers for these constraints
**Constraint Programming over Reals**

Constraint domain: real numbers

Basic constraints: equations / inequations over real arithmetic expressions

Constraint solvers: Gaussian elimination, simplex method

Examples:

\[ 5.1 =: x + 3.5 \quad \Rightarrow \quad \{x=1.6\} \]

\[ x \leq 1.5 \quad \& \quad x+1.3 \geq 2.8 \quad \Rightarrow \quad \{x=1.5\} \]
Define relation \( cvi \) between electrical circuit, voltage, and current

Circuits are defined by the data type

```haskell
data Circuit = Resistor Float
  | Series Circuit Circuit
  | Parallel Circuit Circuit
```

Rules for relation \( cvi \):

\[
\begin{align*}
\text{cvi (Resistor r) v i} &= \text{v} =:= \text{i} \ast \text{r} & \text{-- Ohm’s law} \\
\text{cvi (Series c1 c2) v i} &= \text{v} =:= \text{v1 + v2} & \text{& cvi c1 v1 i & cvi c2 v2 i} & \text{-- Kirchhoff’s law} \\
\text{cvi (Parallel c1 c2) v i} &= \text{i} =:= \text{i1 + i2} & \text{& cvi c1 v i1 & cvi c2 v i2} & \text{-- Kirchhoff’s law}
\end{align*}
\]
Querying the circuit specification:

Current in a sequence of resistors:

\[ cvi (\text{Series (Resistor 180.0) (Resistor 470.0)}) \ 5.0 \ i \]
\[ \sim i = 0.007692307692307693 \]

Relation between resistance and voltage in a circuit:

\[ cvi (\text{Series (Series (Resistor r) (Resistor r)) (Resistor r)}) \ v 5.0 \]
\[ \sim v=15.0*r \]

Also synthesis of circuits possible
Constraint domain: finite set of values

Basic constraints: equality / disequality / membership / . . .

Constraint solvers: OR methods (e.g., arc consistency)

Application areas: combinatorial problems
(job scheduling, timetabling, routing,...)

General method:

1. define the domain of the variables (possible values)
2. define the constraints between all variables
3. “labeling”, i.e., non-deterministic instantiation of the variables

Constraint solver reduces the domain of the variables by sophisticated pruning techniques using the given constraints

Usually: finite domain ≈ finite subset of integers
EXAMPLE: A CRYPTO-ARITHMETIC PUZZLE

Assign a different digit to each different letter such that the following calculation is valid:

\[
\begin{align*}
\text{send} + \text{more} &= \text{money} \\
\end{align*}
\]

\[
\text{puzzle send more} =
\]

domain [s,e,n,d,m,o,r,y] 0 9 & -- define domain
s > 0 & m > 0 & -- define constraints
all_different [s,e,n,d,m,o,r,y] &
1000 * s + 100 * e + 10 * n + d
+ 1000 * m + 100 * o + 10 * r + e
= 10000 * m + 1000 * o + 100 * n + 10 * e + y &
labeling [s,e,n,d,m,o,r,y] -- instantiate variables

\[
\text{puzzle send more} \leadsto \{s=9, e=5, n=6, d=7, m=1, o=0, r=8, y=2\}
\]
Disadvantage of narrowing:

- functions on recursive data structures \( \leadsto \) narrowing may not terminate
- all rules must be explicitly known \( \leadsto \) combination with external functions?

Solution: Delay function calls if a needed argument is free

\( \leadsto \textbf{residuation principle} \) [Aït-Kaci et al. 87]
(used in Escher, Le Fun, Life, NUE-Prolog, Oz, \ldots)

Distinguish: 
- \textbf{rigid} (consumer) and \textbf{flexible} (generator) functions

Necessary: Concurrent conjunction of constraints: \( c_1 \& c_2 \)

Meaning: evaluate \( c_1 \) and \( c_2 \) concurrently, if possible
flexible vs. rigid functions

\[
\begin{align*}
f\ 0 &= 2 \\
f\ 1 &= 3
\end{align*}
\]

rigid/flexible status not relevant for ground calls:

\[
f\ 1 \leadsto 3
\]

\[f\] flexible:

\[
f\ x =:= y \leadsto \{x=0, y=2\} \mid \{x=1, y=3\}
\]

\[f\] rigid:

\[
f\ x =:= y \leadsto \text{suspend}
\]

\[
f\ x =:= y \& x =:= 1 \leadsto \{x=1\} f\ 1 =:= y \quad \text{(suspend } f\ x) \\
\leadsto \{x=1\} 3 =:= y \quad \text{(evaluate } f\ 1) \\
\leadsto \{x=1,y=3\}
\]

Default in Curry: constraints are flexible, all others are rigid
Parallel evaluation of arguments:

\[
\text{letpar} \quad x = g \ t1 \\
y = h \ t2 \quad \text{in} \quad k \ x \ y
\]

with concurrent conjunction of equations:

\[
f \ t1 \ t2 \quad \mid \quad x =: g \ t1 \ & \ y = h \ t2 \quad = k \ x \ y
\]

where \( x, y \) free

Skeleton-based parallel programming:

\textbf{farm}: parallel version of \textbf{map}

\[
farm f \ [ ] \quad = \ [ ] \\
farm f \ (x:xs) \quad | \quad r =: f \ x \ & \ rs =: farm f \ xs \\
\quad = r : rs \quad \text{where} \ r, rs \ free
\]
External functions: implemented in another language (e.g., C, Java, ...)

Conceptually definable by an infinite set of equations, e.g.,

\[

def 

0+0 = 0 \quad 1+0 = 1 \quad 2+0 = 2 \quad \ldots \\
0+1 = 1 \quad 1+1 = 2 \quad \ldots \\
0+2 = 2 \quad \ldots \\
\ldots
\]

Definition not accessible, infinite disjunctions

- suspend external function calls until arguments are fully known, i.e., ground
  [Bonnier/Maluszynski 88, Boye 91]
- no extension to presented computation model (external functions are rigid), but
  *not possible in narrowing-based languages*
- reuse of existing libraries
Implementation of standard arithmetic (+, -, *,...) as external functions:

0, 1, 2, ...: constructors

+, −, *, ...: external functions

\[ x := 2 + 3 \times 4 \quad \Rightarrow \quad \{ x = 14 \} \]

\[ x := 2 \times 3 + y \quad \Rightarrow \quad \{ \} \quad x := 6 + y \quad \text{(suspend)} \]

\[ x + x := y \quad \& \quad x := 2 \]

\[ \Rightarrow \quad \{ x = 2 \} \quad 2 + 2 := y \quad \text{(suspend x+x)} \]

\[ \Rightarrow \quad \{ x = 2 \} \quad 4 := y \quad \text{(evaluate 2+2)} \]

\[ \Rightarrow \quad \{ x = 2, \ y = 4 \} \]

⇒ Rigid functions as passive constraints (Life)
External functions as passive constraints:

\[
\begin{align*}
\text{digit } 0 &= \text{success} \\
&\quad \ldots \\
\text{digit } 9 &= \text{success}
\end{align*}
\]

The constraint `digit` acts as a generator:

\[
x + x =: y \land x \times x =: y \land \text{digit } x
\]

\[
\sim \{x=0, \ y=0\} \lor \{x=2, \ y=4\}
\]
Higher-Order Functional Logic Programming

map :: (a -> b) -> [a] -> [b]

map f [] = []

map f (x:xs) = (f x) : map f xs

Functional programming:  \text{map} \ (1 +) \ [2,3,4] \mapsto [3,4,5]

Logic programming:  \text{map} \ f \ [2,3,4] =:= [3,4,5] \mapsto ???

→ consider application function \( f \ x = (f \ x) \) as external

→ consider partial applications as data terms

→ first-order definition of application function \( ($) \) (as in [Warren 82]):

\begin{align*}
\text{(+) \ $} \ x &= (+ \ x) \quad \text{-- right-hand side is data term} \\
\text{(+ x) $} \ y &= x+y \quad \text{-- evaluate right-hand side}
\end{align*}
Reasonable: application function ($) is rigid

⇒ delay applications of unknown functions

⇒ map f [2,3,4] suspends

Other solutions possible but more expensive:

⇒ ($) is flexible  ⇒ guess unknown functions

⇒ solver for higher-order equations
  (higher-order unification, higher-order needed narrowing)
## Unification of Declarative Computation Models

<table>
<thead>
<tr>
<th>Computation model</th>
<th>Restrictions on programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Needed narrowing</td>
<td>inductively sequential rules; optimal strategy</td>
</tr>
<tr>
<td>Weakly needed narrowing ((\sim) Babel)</td>
<td>only flexible functions</td>
</tr>
<tr>
<td>Resolution ((\sim) Prolog)</td>
<td>only (flexible) predicates ((\sim) constraints)</td>
</tr>
<tr>
<td>Lazy functional languages ((\sim) Haskell)</td>
<td>no free variables in expressions</td>
</tr>
<tr>
<td>Parallel functional langs. ((\sim) Goffin, Eden)</td>
<td>only rigid functions, concurrent conjunction</td>
</tr>
<tr>
<td>Residuation ((\sim) Life, Oz)</td>
<td>constraints are flexible; all others are rigid</td>
</tr>
</tbody>
</table>
Modeling objects with state as a (rigid!) constraint function:

- first parameter: current state
- second parameter: message stream (rigid ≈ wait for input)

Example: **Counter object**

```plaintext
data CounterMessage = Set Int | Inc | Get Int

counter :: Int -> [CounterMessage] -> Constraint
counter eval rigid  -- declare as rigid

counter _ (Set v : ms) = counter v ms
counter n (Inc     : ms) = counter (n+1) ms
counter n (Get v : ms) = v:=:=n & counter n ms
counter _ []        = success
```
**CONCURRENT OBJECTS WITH STATE: A COUNTER**

counter _ (Set v : ms) = counter v ms
counter n (Inc : ms) = counter (n+1) ms
counter n (Get v : ms) = v := n & counter n ms
counter _ [] = success

counter 0 s & -- create counter object
   s := [Set 41, Inc, Get x]

≈ \{x=42, s=...\}

Also: incremental instantiation of s (message sending)

Several sending processes ≈ merge message streams
Data type for representing HTML expressions:

```haskell
data HtmxExp = HText String
            | HStruct String [(String, String)] [HtmxExp]
```

Some useful abbreviations:

```haskell
htxt s = HText (htmlQuote s)  -- plain string
bold hexps = HStruct "B" [] hexps  -- bold font
italic hexps = HStruct "I" [] hexps  -- italic font
h1 hexps = HStruct "H1" [] hexps  -- main header
```

Example:
```
[h1 [htxt "1. Hello World"],
   italic [htxt "Hello"],
   bold [htxt "world!"]]
```

1. Hello World

Hello world!
Erlang (Ericsson)

- developed by Ericsson for telecommunication applications
- concurrent functional language with features to support the development of robust distributed systems
- reduced development time and maintainance

Escher (University of Bristol)

- extension of Haskell by features for logic programming
- functions are evaluated by residuation
- explicit disjunctions for logic programming
- simplification rules for logic formulas
**Mercury** (University of Melbourne)

- logic functional language with highly optimized execution algorithm
- origin: logic programming (syntax) with type/mode/determinism annotations
- adapted concepts from functional programming, strict semantics

**Oz** (DFKI Saarbrücken)

- concurrent constraint language with features for higher-order functional, object-oriented, and distributed programming
- operational behavior: residuation
- search via explicit disjunctions and search operators
**Toy** (Univ. Complutense de Madrid)

- prototype for a functional logic language
- based on lazy narrowing, supports non-deterministic functions
- contraints, in particular, disequality constraints

... and, of course, there are many, many more...
Several implementations available:

- Interpreter in Prolog: TasteCurry-System
- Compiler Curry → Java [Hanus/Sadre ILPS’97/JFLP’99]
  (Java threads for concurrency and non-determinism)
  - portable
  - simplified implementation (garbage collection, threads)
  - slow but (hopefully!) better Java implementations in the future
- [Antoy/Hanus FroCoS’00]: Efficient implementation by transformation into Sicstus-Prolog (reuse of various constraint solvers)
  (also Sloth-System [Mariño/Rey WFLP’98])

⇒ PACS (Portland Aachen Curry System)
  http://www-i2.informatik.rwth-aachen.de/~hanus/pacs
- abstract Curry machine [Lux FLOPS’99]
CONCLUSIONS

Appropriate abstractions are important for software development and maintenance.

Multi-paradigm languages have the potential to express these abstractions.

High-level languages support domain-specific languages.

Multi-paradigm programming

- possible and advantageous
- constraint functional logic programming: many improvements in recent years
- imperative/concurrent/distributed + declarative programming: possible but many different approaches

More infos on Curry:
http://www-i2.informatik.rwth-aachen.de/~hanus/curry