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 ⇒ algebraic data types
  - complex procedures ⇒ 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
- 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 \(\leadsto\) design patterns
  - declarative I/O
  - concurrent constraints
**Imperative vs. Declarative Programming**

**Readability, safety:**

```plaintext
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:

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 := 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
```

Declarative version:

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

```plaintext
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)

```
data Bool    = True | False
data Nat     = Z   | S Nat
data List a  = []  | a : List a       -- [a]
data Tree a   = Leaf a | Node [Tree a]
data Int     = 0  | 1  | -1 | 2  | -2 | ...
```

Value  \(\approx\) data term, constructor term:
well-formed expression containing variables and data type constructors

\((S \ Z) \ 1:(2:[]) \ [1,2] \ Node [Leaf 3, Node [Leaf 4, Leaf 5]]\)
**FUNCTIONAL PROGRAMS**

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

\[ f \; t_1 \; \ldots \; t_n \; | \; c = r \]

- defined operation
- data terms
- condition (optional)
- expression

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

\[
\begin{align*}
[] & \; \text{++} \; ys = ys \\
(x:xs) & \; \text{++} \; ys = x : (xs \; \text{++} \; ys)
\end{align*}
\]

\[
\begin{align*}
\text{depth} \; (\text{Leaf } \_ ) &= 1 \\
\text{depth} \; (\text{Node } []) &= 1 \\
\text{depth} \; (\text{Node } (t:ts)) &= \max (1+\text{depth} \; t) \; (\text{depth} \; (\text{Node} \; ts))
\end{align*}
\]
Evaluate: Computing Values

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

\[ \text{match lhs against subterm (instantiate these variables)} \]

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

\[(S \times)+(S \times) \rightarrow S(Z+(S \times)) \rightarrow S(S \times)\]
Expressions with several redexes: which evaluate first?

**Strict evaluation**: select an innermost redex (≈ call-by-value)

**Lazy evaluation**: select an outermost redex

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

**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...

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

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

\[ x_1 \leq x_2 \]
\[ Z \leq x_2 \]
\[ (S \ x_3) \leq x_2 \]

True
\[ (S \ x_3) \leq Z \]
\[ (S \ x_3) \leq (S \ x_4) \]
False
\[ x_3 \leq x_4 \]

- \[ Z \leq y = \text{True} \]
- \[ (S \ x) \leq Z = \text{False} \]
- \[ (S \ x) \leq (S \ y) = x \leq y \]
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

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x) : map f xs

map (1 +) [2,3,4] ~> [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:

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

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

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

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

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

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

Now the code for quicksort becomes straightforward:

```haskell
qsort [] = []
qsort (x:l) = qsort (filter (<x) l)
    ++ [x] ++ qsort (filter (>=x) l)
```
Data type for representing HTML expressions:

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

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

Get all hypertext links in an HTML document:

```haskell
hrefs [] = []
hrefs (HText _ : hs) = hrefs hs
hrefs (HStruct tag attrs shs : hs) =
  (if tag="A" then map snd (filter (\(t,_)->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 \( \text{True} \lor \text{True} \)

\( \leadsto \) 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**

\[
\begin{align*}
\text{true} \lor x &= \text{true} \\
    x \lor \text{true} &= \text{true} \\
\text{false} \lor \text{false} &= \text{false}
\end{align*}
\]

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: \( e \rightarrow e_1 \mid \cdots \mid e_n \)

\( \leadsto \) non-deterministic functions
Non-deterministic Functions

Functions can have more than one result value:

\[
\begin{align*}
\text{choose } x \ y &= x \\
\text{choose } x \ y &= y \\
\text{choose } 1 \ 2 &\rightarrow 1 \ | \ 2
\end{align*}
\]

Non-deterministic list insertion and permutations:

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

\[
\text{permute } [1,2,3] \rightarrow \\
[1,2,3] \ | \ [2,1,3] \ | \ [2,3,1] \ | \ [1,3,2] \ | \ [3,1,2] \ | \ [3,2,1]
\]
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: \(\text{bind}\) and \(\text{reduce}\) : \(e \leadsto \{\sigma_1\} \ e_1 | \cdots | \{\sigma_n\} \ e_n\)

Reduce: \((f \ 0) \leadsto 2\)

Bind and reduce: \((f \ x) \leadsto \{x=0\} \ 2 \ | \ \{x=1\} \ 3\)

Compute necessary bindings with needed strategy
\(\leadsto \text{needed narrowing} \) [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 (≈ 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:**

   \[
   f = 0:f \quad \text{and} \quad g = 0:g
   \]

   Is \( f = g \) valid?

2. **Semantics of non-terminating functions:**

   \[
   f \ x = f \ (x+1) \quad \text{and} \quad g \ x = g \ (x+1)
   \]

   Is \( f \ 0 = g \ 0 \) 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:

\[
\text{append} :: [a] \rightarrow [a] \rightarrow [a]
\]
\[
\text{append} \ [\] \ y = y
\]
\[
\text{append} \ (x:xs) \ y = x : \text{append} \ xs \ y
\]

Functional programming:

\[
\text{append} \ [1,2] \ [3,4] \rightarrow [1,2,3,4]
\]

Logic programming:

\[
\text{append} \ x \ y =: [1,2] \rightarrow
\{x=[] , y=[1,2]\} \mid \{x=[1], y=[2]\} \mid \{x=[1,2], y=[]\}
\]

Last list element:

\[
\text{last} \ xs \mid \text{append} \ ys \ [x] =: xs = x
\]
FUNCTIONAL LOGIC PROGRAMMING: EXAMPLES

Infinite list of natural numbers:

\[
\begin{align*}
\text{from } x &= x : \text{from } (S \ x) \\
\text{first } Z \quad \text{ys} &= [] \\
\text{first } (S \ x) \ (y:ys) &= y : \text{first } x \ ys
\end{align*}
\]

Lazy functional programming:

\[
\text{first } (S \ (S \ Z)) \ (\text{from } Z) \ \leadsto \ [Z,(S \ Z)]
\]

Lazy functional logic programming:

\[
\text{first } x \ (\text{from } y) \ := : [Z] \ \leadsto \ \{x=(S \ Z),y=Z\}
\]
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:

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

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

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

\[ \Rightarrow^* \text{sorted (5:4:permute } [3,2,1]) | \ldots \]

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 \(\leadsto\) parallel architectures
- explore alternatives by backtracking \(\leadsto\) Prolog
- support flexible search strategies \(\leadsto\) 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])
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:

```
try :: Constraint -> [Constraint]
```
**SEARCH OPERATOR: FIRST APPROACH**

**try :: Constraint \rightarrow [Constraint]**

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

try (1::=2) \leadsto [] \quad \text{failure} \\
try ([x]::=[0]) \leadsto \{x::=0\} \quad \text{success} \\
try (f \ x ::= 3) \leadsto \{x::=0 \land f \ 0 ::= 3, \\
&& x::=1 \land f \ 1 ::= 3\} \quad \text{disjunction}

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

**Solution:** abstract search variable in constraints: \(\forall x \rightarrow c\)
**Search Operator: Final Approach**

**Search goal:** constraint with abstracted search variable

**Search operator** \texttt{try}: maps search goal into list of search goals

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

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

\[
\begin{array}{lll}
\text{try } \lambda x -> 1 =: 2 & \sim & [] & \text{failure} \\
\text{try } \lambda x -> [x] =: [0] & \sim & [\lambda x -> x =: 0] & \text{success} \\
\text{try } \lambda x -> f x =: 3 & \sim & [\lambda x -> x =: 0 \land f 0 =: 3, \\
& & \lambda x -> x =: 1 \land f 1 =: 3] & \text{disjunction}
\end{array}
\]
**ENCAPSULATED SEARCH: SEARCH STRATEGIES**

try \(x \rightarrow c\): evaluate \(c\), stop after non-deterministic step

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

\[
\text{all} :: (a \rightarrow \text{Constraint}) \rightarrow [a \rightarrow \text{Constraint}]
\]

\[
\text{all} \ g = \text{collect} \ (\text{try} \ g)
\]

where
\[
\text{collect} \ [] = []
\]
\[
\text{collect} \ [g] = [g]
\]
\[
\text{collect} \ (g1:g2:gs) = \text{concat} \ (\text{map} \ \text{all} \ (g1:g2:gs))
\]

\[
\text{all} \ (\backslash xs \rightarrow \text{append} \ xs \ [1] =:= [0,1]) \leadsto [\backslash xs \rightarrow xs =:= [0]]
\]
• compute only the first solution:

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

Note: lazy evaluation is important here!
(strict languages, like Oz, must define new search operator)

\[\rightsquigarrow \text{ lazy evaluation supports better reuse}\]

• \textit{findall}, best solution search, parallel search, ...

• negation as failure:

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

\[\rightsquigarrow \text{ control failures}\]
Extract value of the search variable by application of search goal:

\( (\text{x} \to \text{x}=:1) \) freevar  \( \leadsto \) freevar=:1
\( \leadsto \{\text{freevar}=1\} \) success

**Prolog’s findall:**

\[
\begin{align*}
\text{unpack} &:: (\text{a} \to \text{Constraint}) \to \text{a} \\
\text{unpack g} & | \text{g x = x} \quad \text{where x free} \\
\text{findall g} &= \text{map unpack (all g)}
\end{align*}
\]

**Compute all splittings of a list:**

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

```
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) -> append x y =:= [1,2]
\* ([], [1,2]) \? ;
([1], [2]) \? <-
  yes
prolog \(x -> 1 =:= 2 \* no
```
Laziness easily supports demand-driven encapsulated search

⇒ Separation of Logic and Control

⇒ Modularity:

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

- Prolog’s top-level with depth-bounded search:
  \[
  \text{prolog\_bound } g \text{ bd } = \text{printloop}\left(\text{bound } g \text{ bd}\right)
  \]
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: $\text{IO } a \simeq \text{World } \rightarrow \langle a, \text{World} \rangle$

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

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

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

**Compose actions:** \((\ggg) : \text{IO a} \rightarrow (\text{a} \rightarrow \text{IO b}) \rightarrow \text{IO b}

\text{getChar} \ \ggg \ \text{putChar}: \text{copy character from input to output}

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

\begin{align*}
(\ggg) & : \text{IO a} \rightarrow \text{IO b} \rightarrow \text{IO b} \\
x \ggg y & = x \ggg \_ \rightarrow y
\end{align*}
Example: output action for strings \((\text{String} \approx [\text{Char}])\)

\[
\begin{align*}
\text{putStr} & : \text{String} \rightarrow \text{IO} () \\
\text{putStr} \, [] & = \text{return} () \\
\text{putStr} \, (c:cs) & = \text{putChar} \, c \gg \text{putStr} \, cs
\end{align*}
\]

Example: read a line

\[
\begin{align*}
\text{getLine} & : \text{IO} \ \text{String} \\
\text{getLine} & = \text{getChar} \gg= \backslash c \rightarrow \\
& \quad \text{if} \ c == \backslash n \ \text{then} \ \text{return} \ [] \\
& \quad \text{else} \ \text{getLine} \gg= \backslash cs \rightarrow \text{return} \ (c:cs)
\end{align*}
\]
Monadic composition not well readable

\[ \rightsquigarrow \text{syntactic sugar: Haskell's } \textbf{do} \text{ notation} \]

\[ \text{do } p \leftarrow a_1 \quad \approx \quad a_1 \gg\gg \lambda p \rightarrow a_2 \]

\[ a_2 \]

Example: read a line (with \textbf{do} notation)

\begin{verbatim}
   getline = do c <- getChar
               if c==\'\n\' then return []
               else do cs <- getline
                       return (c:cs)
\end{verbatim}

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

\[ \rightsquigarrow \text{encapsulate search between I/O actions} \]
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 domain: real numbers

Basic constraints: equations / inequations over real arithmetic expressions

Constraint solvers: Gaussian elimination, simplex method

Examples:

\[ 5.1 =: x + 3.5 \quad \leadsto \quad \{x=1.6\} \]
\[ x \leq 1.5 \; \& \; x+1.3 \geq 2.8 \quad \leadsto \quad \{x=1.5\} \]
Define relation $c_{vi}$ 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 $c_{vi}$:

```haskell
cvi (Resistor r) v i = v :== i * r  -- Ohm's law

cvi (Series c1 c2) v i =
    v :== v1 + v2 & cvi c1 v1 i & cvi c2 v2 i  -- Kirchhoff's law

cvi (Parallel c1 c2) v i =
    i :== i1 + i2 & cvi c1 v i1 & cvi c2 v i2  -- Kirchhoff's law
```
Querying the circuit specification:

Current in a sequence of resistors:
\[ \text{cvi (Series (Resistor 180.0) (Resistor 470.0)) 5.0 i} \]
\[ \leadsto \{i = 0.007692307692307693\} \]

Relation between resistance and voltage in a circuit:
\[ \text{cvi (Series (Series (Resistor r) (Resistor r)) (Resistor r)) v 5.0} \]
\[ \leadsto \{v=15.0*r\} \]

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:

\[
\begin{align*}
\text{puzzle } & \quad \text{send} + \text{more} = \text{money} \\
\text{domain } & \quad [s,e,n,d,m,o,r,y] 0 \, 9 \\
\text{s > 0 } & \quad \text{m > 0} \\
\text{all different } & \quad [s,e,n,d,m,o,r,y] \\
\text{1000 } & \quad \text{100} \\
\text{+ } & \quad \text{10} \\
\text{= } & \quad \text{10000} \\
\text{labeling } & \quad [s,e,n,d,m,o,r,y] \\
\text{puzzle } & \quad \text{send} \sim \{s=9,e=5,n=6,d=7,m=1,o=0,r=8,y=2\}
\end{align*}
\]
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

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

Distinguish: **rigid** (consumer) and **flexible** (generator) functions

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

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

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

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

Default in Curry: constraints are flexible, all others are rigid
Parallel Functional Programming

Parallel evaluation of arguments:

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

with concurrent conjunction of equations:

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

where \( x, y \) free

Skeleton-based parallel programming:

\( f\text{arm} \): parallel version of map

\[
f\text{arm} f \ [] = []
\]

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

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

\[
\begin{align*}
0+0 &= 0 \\
1+0 &= 1 \\
2+0 &= 2 \\
0+1 &= 1 \\
1+1 &= 2 \\
0+2 &= 2 \\
\ldots
\end{align*}
\]

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

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

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

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

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

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

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

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

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

The constraint \texttt{digit} acts as a generator:

\[
x + x = := y \ & \ x \times x = := y \ & \ \texttt{digit x}
\]

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

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

map f [] = []

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

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

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

\( \rightarrow \) consider application function \( f \$ x = (f \ x) \) as external

\( \rightarrow \) consider partial applications as data terms

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

\[ (+) \$ x = (+ x) \quad -- \text{right-hand side is data term} \]

\[ (+ x) \$ y = x+y \quad -- \text{evaluate right-hand side} \]
Reasonable: application function (\$) is rigid

\[\Rightarrow\] delay applications of unknown functions

\[\Rightarrow\] map f \([2,3,4]\) suspends

Other solutions possible but more expensive:

\[\Rightarrow\] (\$) is flexible \(\Rightarrow\) guess unknown functions

\[\Rightarrow\] solver for higher-order equations
  (higher-order unification, higher-order needed narrowing)
<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 (~Babel)</td>
<td>only flexible functions</td>
</tr>
<tr>
<td>Resolution (~Prolog)</td>
<td>only (flexible) predicates (~ constraints)</td>
</tr>
<tr>
<td>Lazy functional languages (~Haskell)</td>
<td>no free variables in expressions</td>
</tr>
<tr>
<td>Parallel functional langs. (~Goffin, Eden)</td>
<td>only rigid functions, concurrent conjunction</td>
</tr>
<tr>
<td>Residuation (~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**

```
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=\ldots\}

Also: incremental instantiation of s (message sending)

Several sending processes ⇒ merge message streams
Distributed systems: $n \rightarrow 1$-communication with dynamic connections

**Port** [Janson et al. 93, AKL]: constraint between multiset $p$ and stream $s$ satisfied if elements in $p$ and $s$ are identical

Two constraints on ports:
- **openPort** $p$ $s$ open port $p$ with stream $s$
- **send** $m$ $p$ constrain $p$ to hold message $m$

Previous counter with two clients:
- **openPort** $p$ $s$ &\> **counter** 0 $s$ & **client1** $p$ & **client2** $p$
• communication based on logic (constraint solving)

• simple extension of base semantics

• \texttt{send} instantiates end of stream \( s \) (in constant time)
  \[
  s_{\text{tail}} := (m: \text{new}\ s_{\text{tail}})
  \]
  \( \sim \) strict communication

• provides efficient implementation
  (senders have no access to old messages)

• free variables in messages \( \approx \) reply channels

• dynamic extension of senders (pass port variable)
I/O actions for external communication
(between different programs running on different machines):

openNamedPort :: String -> IO [a]
connectPort :: String -> IO (Port a)

openNamedPort pn: open new external port with global name pn and return stream of incoming messages
connectPort pn: return port with global name pn

(similar concepts: external objects in Oz, registered processes in Erlang)
A simple example: a global counter server

The server side: \texttt{(started on medoc.cs.uni-kiel.de)}

\begin{verbatim}
main = openNamedPort "counter" >>= c_server

     c_server s | counter 0 s = done
\end{verbatim}

The client side:

\begin{verbatim}
client pn m = connectPort pn >>= sendPort m

     sendPort msg p | send msg p = done
\end{verbatim}

Increment the global counter:

\begin{verbatim}
client "counter@medoc.cs.uni-kiel.de" Inc
\end{verbatim}

Ask the counters current value:

\begin{verbatim}
client "counter@medoc.cs.uni-kiel.de" (Get v) \leadsto \{v=...\}
\end{verbatim}
Messages: “\texttt{PutName }n\ i” (assign \texttt{i} to name \texttt{n}) \quad “\texttt{GetName }n\ i”

\begin{verbatim}
nameserver = openNamedPort "nameserver" >>= serverloop \_->0
serverloop n2i (GetName n i : ms) \mid i:=:(n2i n)
  = serverloop n2i ms
serverloop n2i (PutName n i : ms) = serverloop new_n2i ms
where new_n2i m = if m==n then i else n2i m
\end{verbatim}

The client side:

\begin{verbatim}
client "nameserver@..." (PutName "talk" 42)
client "nameserver@..." (GetName "talk" x) \leadsto \{ x=42 \}
\end{verbatim}
Internet domain name server: ask master server if name locally unknown

Implementation by slight modification of previous name server:

```plaintext
serverloop n2i (GetName n i : ms)
    | if (n2i n)==0 then send (GetName n i) master
    | else i:=:(n2i n)
    | = serverloop n2i ms

serverloop n2i (PutName n i : ms) = serverloop new_n2i ms
where new_n2i m = if m==n then i else n2i m
```
Strict communication, no RPCs $\leadsto$ no direct way to distribute work

**Computation server:** accepts messages $(f,x,y)$

```haskell
start_cserver = openNamedPort "compserver" >>= compservlet
compservlet ((f,x,y) : ms) | y:=:=f x = compservlet ms
```

**Client side:** client "compserver@cs" (prime,1000,p) $\leadsto$ {p=7919}

$\rightarrow$ consider partially applied function calls as data terms

$\rightarrow$ asynchronous RPCs
  (free result variable $\approx$ “promise” [Liskov/Shrira 88])

$\rightarrow$ **concurrent server:**

```haskell
compservlet eval rigid
compservlet ((f,x,y) : ms) = y:=:=f x & compservlet ms
```
Integration of different programming paradigms is possible

Functional programming is a good starting point:

- lazy evaluation $\leadsto$ modularity, optimal evaluation
- higher-order functions $\leadsto$ code reuse, design patterns
- polymorphism $\leadsto$ type safety, static checking

Stepwise extensible in a conservative manner to cover

- logic programming: non-determinism, free variables
- constraint programming: specific constraint structures
- concurrent programming: suspending function calls, synchronization on logical variables
- object-oriented programming: constraint functions, ports
- imperative programming: monadic I/O, sequential composition
- distributed programming: external ports
Why Integration of Declarative Paradigms?

- more expressive than pure functional languages (compute with partial information/constraints)
- 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 (arithmetic, I/O)
- combine research efforts in FP and LP
- do not teach two paradigms, but one: declarative programming
  [Hanus PLILP’97]
- choose the most appropriate features for application programming
So far: high-level approach to

- search problems
- constraint solving
- distributed systems

In the following: appropriate to develop domain-specific languages for

- graphical user interfaces
- parsing
- HTML/CGI programming
Graphical User Interfaces (GUIs) have a

- layout structure \(\sim\) hierarchical structure, algebraic data type
- logical structure \(\sim\) dependencies in the layout structure

Tcl/Tk: assign strings to layout elements \(\sim\) run-time errors

Here: use logical variables as references \(\sim\) compiler errors

A simple “Hello world” GUI:

```latex
runWidget "Hello"
(TkCol [TkLabel [TkText "Hello world!"],
  TkButton tkExit [TkText "Stop"]])
```
Specify hierarchical GUI layout as a “TkWidget” term:

data TkWidget a =
  TkButton (GUIRef -> a) [TkConfItem a]
  TkCheckButton [TkConfItem a]
  TkEntry [TkConfItem a]
  TkLabel [TkConfItem a]
  TkScale Int Int [TkConfItem a]
  TkTextEdit [TkConfItem a]
  ...
  TkRow [TkWidget a]
  TkCol [TkWidget a]
A specification of a counter GUI:

TkCol

[TkEntry [TkRef val, TkText "0"],
   TkRow [TkButton (tkUpdate incr val) [TkText "Increment"],
         TkButton (tkSetValue val "0") [TkText "Reset"],
         TkButton tkExit [TkText "Stop"]]]

where val free

- the free variable val is a reference to the entry widget
- val is used in the event handlers of other widgets
- val is part of the logical structure of the GUI
Logical Structure of GUIs

Configuration options for GUIs:

data TkConfItem a =
   TkText String -- initial text
| TkBackground String -- background color
| TkRef TkRefType -- widget reference
| TkCmd (GUIRef -> a) -- event handler
:

TkRef: reference to a widget, used in event handlers
(TkRefType is abstract ~ argument is a logical variable)

rkExit :: GUIRef -> IO ()
tkGetValue :: TkRefType -> GUIRef -> IO String
tkSetValue :: TkRefType -> String -> GUIRef -> IO ()
tkUpdate :: (String->String) -> TkRefType -> GUIRef -> IO ()

Remark: event handlers also available as constraints
Convert a temperature from Celsius into Fahrenheit:

![Temperature Conversion](image)

TkCol [TkLabel [TkText "Temperature in Celsius:"],
    TkScale 0 100 [TkRef cels, TkCmd convert],
    TkRow [TkLabel [TkText "Temperature in Fahrenheit: "]],
    TkMessage [TkRef fahr, TkBackground "white"]]

where cels,fahr free

convert gr =
    tkGetValue cels gr >>= \cs ->
    tkSetValue fahr (show ((parseInt cs) * 9 'div' 5 + 32)) gr
Implementation consists of two parts:

1. **Object for storing the state**
   - state: (operand, accumulator function)
   - messages: `Display s`, `Button b`

2. **GUI for showing the state**
Object for storing the state:

Message  Display s: instantiate s with current display

calcMgr (d,f) (Display s : ms) = s =:= (show d) &>
    calcMgr (d,f) ms

Message  Button b: the user has pressed button b

calcMgr (d,f) (Button b : ms)
    | isDigit b = calcMgr (10*d + ord b - ord ’0’, f) ms
    | b==’+’    = calcMgr (0, ((f d) +)) ms
    | b==’-’    = calcMgr (0, ((f d) -)) ms
    | b==’*’    = calcMgr (0, ((f d) *)) ms
    | b==’/’    = calcMgr (0, ((f d) ‘div‘)) ms
    | b==’=’    = calcMgr (f d, id) ms
    | b==’C’    = calcMgr (0, id) ms
GUI for showing the state with a reference \texttt{cm} to calculator object:

```haskell
calc_GUI cm = TkCol [TkEntry [TkRef display, TkText "0"],
    TkRow (map cbutton ['1','2','3','+']),
    TkRow (map cbutton ['4','5','6','-']),
    TkRow (map cbutton ['7','8','9','*']),
    TkRow (map cbutton ['C','0','=','/'])]
```

where \texttt{display} free

```haskell
cbutton b = TkButton (click b) [TkText [b]]
```

```haskell
click b gr = let d free in send (Button b) cm &>
    send (Display d) cm &>
    tkCSSetValue display d gr
```

$\rightarrow$ model-view-controller paradigm \textit{à la} Smalltalk-80

$\rightarrow$ different (distributed) views on one application
Functional features useful for

- layout specification
- event handlers (data structures with functional components)
- application-oriented extensions

Logic programming features useful for

- dealing with dependencies inside a structure (free variables)
- handling state (concurrent objects)

Distributed features

GUIs for distributed applications

Specification (rather than imperative programming) of GUIs

Domain-specific language for GUIs, but:

no extension to base language necessary
Logic programming of parsers:

- nonterminals consume corresponding tokens (difference lists)
- definite clause grammars for nice notation
- non-deterministic grammars/parsing
- resulting representations as arguments

Functional programming of parsers:

- parsers consume corresponding tokens
- powerful parser combinators
- more complex handling of alternatives and representations
Functional logic programming of parsers:
simpler handling of representations and alternatives due to

- non-deterministic functions
- free variables as arguments

Parser \( \approx \) function of type \([\text{token}] -> [\text{token}]\)
Argument: list of tokens to be parsed
Result: list of remaining unparsed tokens

A parser recognizing token ’a’:
\[
\text{parse}_a (’a’:\text{ts}) = \text{ts}
\]

A parser recognizing a given token:
\[
\text{terminal sym (t:ts) | sym=:=t} = \text{ts}
\]

Parser recognizing the empty word:
\[
\text{empty sentence} = \text{sentence}
\]
Parser combinators: higher-order functions to combine parsers

Alternative of two parsers \( p \) and \( q \): combinator \( p \ <| > \ q \)

\[
(p < | > q) \text{ sentence } = p \text{ sentence } \\
(p < | > q) \text{ sentence } = q \text{ sentence }
\]

Sequence of two parsers \( p \) and \( q \): combinator \( p \ <\* > q \)

\[
(p1 <\* > p2) s0 \mid p1 s0 =:= s1 = p2 s1 \quad \text{where } s1 \text{ free }
\]

Repetition of a parser: (zero or more times)

\[ \text{star } p = (p <\* > \text{star } p) < | > \text{empty} \]

Parser for \( a(a|b)^* \):

\[
\text{terminal } 'a' <\* > \text{star } (\text{terminal } 'a' < | > \text{terminal } 'b')
\]
A parser for palindromes over the alphabet \{a, b\}

\[
pali = \text{empty} \ (\rightarrow) \ a \ (\rightarrow) \ b \ (\rightarrow) \ a \ (\star) \ pali \ (\star) \ a \ (\rightarrow) \ b \ (\star) \ pali \ (\star) \ b
\]

\[
a = \text{terminal} \ 'a'
\]

\[
b = \text{terminal} \ 'b'
\]

Checking a sentence for a palindrome:

\[
pali \ "abaaba" \ =: \ []
\]

Using logic programming features, we can also generate palindromes:

\[
pali \ [x,y,z] \ =: \ []
\]

\[
\leadsto \quad \{x='a',y='a',z='a'\} \ | \ \{x='a',y='b',z='a'\} \\
\quad | \ \{x='b',y='a',z='b'\} \ | \ \{x='b',y='b',z='b'\}
\]
Parsers should not only check a list of tokens but also return a representation (e.g., abstract syntax tree)

- **Functional programming**: parsers have result \((rep, tokens)\)
- **Logic programming**: parsers have \(rep\) argument \(\leadsto\) simpler definitions

Parser with representation \(\approx rep \to [token] \to [token]\)

Representation argument:

- usually free variable
- will be instantiated during parsing
Alternative of two parsers \( p \) and \( q \): combinator \( p \ <||> \ q \)

\((p \ <||> \ q) \ rep \ = \ p \ rep \ <|> \ q \ rep\)

(reuse combinator for parsers without representation)

Attach representation \( \text{exp} \) to a parser \( p \): combinator \( p \ >>> \ \text{exp} \)

\((p \ >>> \ \text{exp}) \ rep \ s\_in \mid p \ s\_in =:= s\_out \ & \ \text{exp} =:= \ rep \ = s\_out\)

where \( s\_out \) free

Repetition of a parser with representation: (representation is list)

\(\text{star} \ p = p \ r \ <*> \ (\text{star} \ p) \ \text{rs} \ >>> \ (r:\text{rs})\)

\(<||> \ \text{empty} \ >>> \ []\)

where \( r, \text{rs} \) free

At least one repetition of a parser:

\(\text{some} \ p = p \ r \ <*> \ \text{star} \ p \ \text{rs} \ >>> \ (r:\text{rs})\)

where \( r, \text{rs} \) free
Example: Parser for Arithmetic Expressions

\[
\begin{align*}
\text{expr} & \quad = \quad \text{term} \; t \; \text{!!} \; \text{plus\_minus} \; \text{op} \; \text{!!} \; \text{expr} \; e \quad \text{!!} \; \text{!!} \; (\text{op} \; t \; e) \\
\quad & \quad \| \| \quad \text{term} \\
\text{term} & \quad = \quad \text{factor} \; f \; \text{!!} \; \text{prod\_div} \; \text{op} \; \text{!!} \; \text{term} \; t \quad \text{!!} \; \text{!!} \; (\text{op} \; f \; t) \\
\quad & \quad \| \| \quad \text{factor} \\
\text{factor} & \quad = \quad \text{terminal} \; \text{'}(\text{!!} \; \text{expr} \; e \; \text{!!} \; \text{terminal} \; 	ext{'})' \quad \text{!!} \; e \\
\quad & \quad \| \| \quad \text{num} \\
\text{plus\_minus} & \quad = \quad \text{terminal} \; \text{'}+\text{'} \quad \text{!!} \; (+) \\
\quad & \quad \| \| \quad \text{terminal} \; \text{'}-\text{'} \quad \text{!!} \; (-) \\
\text{prod\_div} & \quad = \quad \text{terminal} \; \text{'}*\text{'} \quad \text{!!} \; (*) \\
\quad & \quad \| \| \quad \text{terminal} \; \text{'}/\text{'} \quad \text{!!} \; \text{div} \\
\text{num} & \quad = \quad \text{some} \; \text{digit} \; l \quad \text{!!} \; \text{numeric\_value} \; l \\
\text{Example:} \quad \text{expr} \; \text{val} \; "(10+5*2)/4" \; =:= \; [] \; \leadsto \; \{ \text{val}=5 \}
\end{align*}
\]
Higher-order features useful for

- combining parsers (parsers are functions)
- computing representations

Logic programming features useful for

- dealing with alternatives (non-deterministic functions)
- managing representations (free variables in arguments)
- parsing with constraints

Domain-specific language for parsing, but:

no extension to base language necessary
Early days of the World Wide Web: web pages with static contents

Common Gateway Interface (CGI): web pages with dynamic contents

Retrieval of a dynamic page:

- server executes a program
- program computes an HTML string, writes it to stdout
- server sends result back to client

HTML with input elements (forms):

- client fills out input elements
- input values are sent to server
- server program decodes input values for computing its answer
TRADITIONAL CGI PROGRAMMING

CGI programs on the server can be written in any programming language

- access to environment variables (for input values)
- writes a string to stdout

Scripting languages: (Perl, Tk, . . . )

- simple programming of single pages
- error-prone: correctness of HTML result not ensured
- difficult programming of interaction sequences

Specialized languages: (MAWL, DynDoc, . . . )

- HTML support (structure checking)
- interaction support (partially)
- restricted or connection to existing languages
Library in multi-paradigm language

Exploit functional and logic features for

- HTML support (data type for HTML structures)
- simple access to input values (free variables and environments)
- simple programming of interactions (event handlers)
- wrapper for hiding details

Exploit imperative features for

- environment access (files, databases, ...)

Domain-specific language for HTML/CGI programming
Data type for representing HTML expressions:

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

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!*
Advantages:

- static checking of HTML structure (well-balanced parentheses)
- flexible dynamic documents
- functions for computing HTML documents

Converting tree structure (leaves contain strings) into nested HTML lists:

```haskell
data Tree a = Leaf a | Node [Tree a]

htmlTree :: Tree String -> [HtmlExp]
htmlTree (Leaf s) = [htxt s]
htmlTree (Node trees) = [ulist (map htmlTree trees)]

ulist :: [[HtmlExp]] -> HtmlExp
ulist items = HStruct "UL" [] (map litem items)
litem hexps = HStruct "LI" [] hexps
```
Specific HTML elements for dealing with user input

```html
<INPUT TYPE="TEXT" NAME="INPTEXT" VALUE="fill out!"> 
```

Form is submitted ~

clients sends the current value of this field (identified by "INPTEXT")

Expressible as HTML term:

```html
HStruct "INPUT" [("TYPE","TEXT"),("NAME","INPTEXT"),
                    ("VALUE","fill out!")] []
```

Problems:

- server program must decode input values
- server program must know right names of field identifiers ("INPTEXT")
- error-prone
Solution:

- use free variables as references to input fields (*CGI references*)
- collect input values in *CGI environments*: mapping from CGI references to strings
- associate *event handlers* to submit buttons
- event handlers take a CGI environment and produces an HTML form

Implementation:

*straightforward in a functional logic language!*
CGI references:

data CgiRef = CgiRef String  -- data constructor not exported

→ no construction of wrong references
→ only free variables of type CgiRef
→ global wrapper function instantiates with the right strings

HTML elements with CGI references:

data HtmlExp = ... | HtmlCRef HtmlExp CgiRef

Example: Text fields with a CGI reference and initial contents

textfield :: CgiRef -> String -> HtmlExp

textfield (CgiRef ref) contents =

  HtmlCRef (HStruct "INPUT" [(["TYPE","TEXT"),
                          (["NAME",ref],["VALUE",contents])])

  (CgiRef ref)
HTML form: title + list of HTML expressions

data HtmlForm = Form String [HtmlExp]

Example: simple form with a single input element (a text field)
Form "Form" [h1 [htxt "A Simple Form"],
               htxt "Enter a string:", textfield sref ""]

CGI environments: map CGI references to strings

type CgiEnv = CgiRef -> String

Event handlers have type CgiEnv -> IO Form

Event handlers are associated to submit buttons:
user presses a submit button
~ execute associated event handler with current environment
**Example: Form to Reverse/Duplicate a String**

Form "Question" [htxt "Enter a string: ", textfield tref ",", hr, button "Reverse string" revhandler, button "Duplicate string" duphandler]

where tref free

revhandler env = return $ Form "Answer"

[hl [htxt ("Reversed input: " ++ rev (env tref))]]

duphandler env = return $ Form "Answer"

[hl [htxt ("Duplicated input: " ++ env tref ++ env tref)]]
Form to show the contents of an arbitrary file stored at the server:

Form "Get File" [htxt "Enter local file name:",
    textfield fileref "",
    button "Get file!" handler]

where fileref free

handler env =
    do contents <- readFile (env fileref)
    return $ Form "Answer"
        [h1 [htxt ("Contents of file " ++ env fileref)],
         verbatim contents]
The main form is executed by a wrapper function

\[
\text{run\text{c}gi :: String} \rightarrow \text{IO HtmlForm} \rightarrow \text{IO ()}
\]

- takes a title string and a form and transforms it into HTML text
- replaces all CGI references by unique strings
- decodes input values and invokes associated event handler

Event handlers return forms rather than HTML expressions

- sequences of interactions
- use control abstractions (branching, recursion) of underlying language
- state between interactions handled by CGI environments

Note: no language extension necessary (CGI library)

multi-paradigm languages as scripting languages
**A Few Further Multi-Paradigm Languages**

**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: resudiation
- 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]
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