Controlling Search in Declarative Languages

Frank Steiner

RWTH Aachen

Joint work with

Michael Hanus

RWTH Aachen

Global Search

Logic and functional logic languages:
free variables in goals ⇒ search for solutions

Problems with global search/backtracking:

- trashing (failure of global predicates)
- non-backtrackable/monadic I/O
- built-in search strategy: unflexible
- no control over failures

Idea of encapsulated search:
Control non-determinism via search operator

Evaluation in Curry

Curry: functional logic language with
residuation and narrowing [POPL'97]

Needed Narrowing: [Antoy/Echahed/Hanus POPL'94]

- lazily reduce expressions to their values
- evaluate subterms as demanded by lhs of rules
- bind free variables in goals if neccessary

$\Rightarrow$ compute pairs $\{\sigma\}$ e

Evaluation in Curry

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

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

f(X) $\stackrel{*}{\Rightarrow}$ {X=a} c | {X=b} d

{append(Y,X)=[1,2]}   % constraint
$\Rightarrow$ {Y=[]}   {append([],X)=[1,2]}
  | {Y=[Z|Zs]} {append([Z|Zs],X)=[1,2]}

$\Rightarrow$ non-determinism causes global splitting

Encapsulated Search

Provide search operator to control non-determinism:

- avoid global splitting/global backtracking
- enable easy programming of user-defined search strategies
- enable comparison of solutions (best solution search)
- control failures

Base language must provide a one step relation $\Rightarrow$:

$$e \;\Rightarrow\; \underbrace{\{\sigma_1\}\ e_1 \mid \cdots \mid \{\sigma_n\}\ e_n}_{\text{disjunctive expression}}$$

Search operator: Call "$\Rightarrow$" and process results

Search Operator

Generalization of Oz's operator [Schulte, Smolka '94]

Idea:

- Search in a constraint $\{c_1, \ldots, c_n\}$
- Evaluate constraint until failure, success or non-determinism
- Return results in a list

Search Operator:

```
try :: constraint -> [constraint]
```

```
try({1=2})      ⇒ []                      failure
try({X=f(a)})   ⇒ [{X=c}]                 success
try({f(X)=c})   ⇒ [{X=a,f(a)=c},
                   {X=b,f(b)=c}]
                                          disjunction
```

$\Rightarrow$ splitting avoided, failure controlled

- abstract search variable to enable different bindings:
  search goal \x->$\{c\}$

(a->   )   a->

\X->

\X->   \X->

\X->   \X->

              \X->

Local Variables

Free variables in search goals:

```
try(\X->{append(Y,X)=[1,2]})
```

Binding `Y` causes problems:

- global binding ⇒ no encapsulation
- different bindings of `Y` cause failure

⇒ suspend if free variables must be bound

Other possibility: local variables

```
(*) try(\X->{let Y free in append(Y,X)=[1,2]})
```

⇒ `Y` existentially quantified, (different) bindings allowed

```
(*) ⇒
[\X->{append([],X)=[1,2]},
 \X->{let Z,Zs free in append([Z|Zs],X)=[1,2]}]
```

Search Strategies

try returns alternatives after non-determ. $\Rightarrow$ selection

Depth-first search: collect all solutions

```
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(\X->{let Y free in append(Y,X)=[1,2]})
```

$\overset{*}{\Rightarrow}$ `[\X->{X=[1,2]}, \X->{X=[2]}, \X->{X=[]}]`

Search Strategies

Further strategies:

- first solution: `once(G) = head(all(G))`
- best solution search (`best`)
- breadth-first search (`bfs`)
- bounded depth-first search (`bound`)
- iterative deepening (`id`)
- ...

Handling solutions

Extract value of the search variable:

`(\X->{X=1})@L` $\overset{*}{\Rightarrow}$ `{L=1} {}`

Prolog's findall:

```
unpack([]) = []
unpack([G|Gs]) if {G@L} = [L|unpack(Gs)]

findall(G) = unpack(all(G))
```

`findall(\X->{let Y free in append(Y,X)=[1,2]})`
$\overset{*}{\Rightarrow}$ `[[1,2],[2],[]]`

Exploiting laziness

Demand-driven search easy to implement with laziness:

```
prolog(G) = printloop(all(G))

printloop([])     = putStr("no") >> nl
printloop([A|As]) = browse(A) >> putStr("? ") >>
                    getChar >>= evalAnswer(As)

evalAnswer(As,';')  = nl >> printloop(As)
evalAnswer(As,'\n') = nl >> putStr("yes") >> nl
```

```
prolog(\X->{let Y free in append(Y,X)=[1,2]})
```
$\stackrel{*}{\Rightarrow}$ `[1,2] ? ;`
`[2] ? <-`
`yes`

`prolog(\X->{1=2})` $\stackrel{*}{\Rightarrow}$ `no`

Flexibility:

- `prolog_bfs(G) = printloop(bfs(G))`
- `prolog_bound(G,B) = printloop(bound(G,B))`
- ...

Soundness and Completeness

Theorem (**Soundness of all**)
If `all(\X->c)` $\overset{*}{\Rightarrow}$ `[`$L_1, L_2, \ldots$`]` *then each* L_i *is a* $\Rightarrow$*-computed answer for* `c`*.*

Theorem (**Completeness of all**)
Similarly, for finite search trees.

Theorem (**Completeness of bfs**)
Similarly, for deterministically terminating search trees.

Related Work

Prolog:

- fixed search strategy: depth-first search
- search predicates: `findall/setof/bagof`
  - fixed to dfs, no best solution search etc.
  - no laziness
- `try` covers Prolog's shell and search predicates

Haskell: List Comprehension

- fixed search strategy, no best solution search etc.
- no partial data structures allowed
- list comprehension cannot directly be expressed with `try`, but in most concrete examples

Oz: solve operator

- bound to certain construct of the language (`or`)
- no laziness ⇒ separate algorithms for demand-driven search

Conclusions

Primitive for encapsulating search: `try operator`

- encapsulation of non-determinism (I/O) and failures
- variety of (user-defined) search strategies
- based on the meaning of computation steps, not bound to a language construct
- soundness and completeness results from the base language

Applicable to (functional) logic languages providing

- equational constraints
- existential quantification
- lambda abstractions (optional)
- laziness (optional) $\Rightarrow$ easy demand-driven search