

First Class Constraint Programming

Tom Schrijvers, Peter Stuckey, Phil Wadler

K.U.Leuven, University of Melbourne, University of Edinburgh

November 28, SingHaskell, Singapore

Overview

2/38



- 1 CP Background
- 2 Goals
- 3 The Solver Interface
- 4 The Search Tree
- 5 Basic Search Strategies
- 6 Search Strategy Transformers
- 7 Advanced Features
- 8 Conclusion

Overview

3/38



- 1 CP Background
- 2 Goals
- 3 The Solver Interface
- 4 The Search Tree
- 5 Basic Search Strategies
- 6 Search Strategy Transformers
- 7 Advanced Features
- 8 Conclusion

Constraint Programming

- ▶ programmer: declarative (logic) specification of problem
- ▶ constraint solver: solution
 - ▶ black box
 - ▶ “general purpose”
 - ▶ clever algorithms

CP Background

5/38

Constraint domain: finite domain (integers), Herbrand terms, reals, booleans, ...

- ▶ terms: variables and e.g. integers
- ▶ primitive constraints: e.g. $x @:: (1,10)$
- ▶ compound constraints: e.g. $x @:: (1,10) /\wedge x @< y$
- ▶ global constraints: e.g. alldifferent xs

CP Background

6/38

```

sudoku n puzzle =
  mexist (n * n) $ \matrix ->
    mconj (choices matrix puzzle) /\
    malldifferent (rows matrix) /\
    malldifferent (cols matrix) /\
    malldifferent (boxes matrix)
  where choices = (zipWith . zipWith) choice
                 choice v w
                 | w == 0      = v @:: (1,n*n)
                 | otherwise   = v @:: (w,w)

```

CP Applications

7/38

Beyond sudoku puzzles:

- ▶ Operations Research: planning, allocation, ...
 - ▶ crew assignment: BA, SAS, Swissair, TGV
 - ▶ dock allocation: Hong Kong harbour
- ▶ Other
 - ▶ type checking
 - ▶ layout GUIs, graphs
 - ▶ ...

How does CP work?

8/38

Constraint Solver

- ▶ rewrites constraints

- 1 propagation

- 2 labelling

- ▶ to *inconsistency* or *solved form*

e.g. $x @:: (2,1)$ or $x @:: (2,2)$

How does CP work?

9/38

Propagation

- ▶ (hard-wired) reasoning
- ▶ to refine the unknown

$$x @:: (1, 10) /\ \ y @:: (1, 5) /\ \ x @ < y$$
$$\implies$$
$$x @:: (1, \mathbf{4}) /\ \ y @:: (\mathbf{2}, 5)$$

How does CP work?

10/38

Labelling

- ▶ dumb & general counterpart of smart & incomplete propagation
- ▶ applied when propagation is stuck to get it going again
- ▶ labelling = search: enumerate alternatives and see what works

```
x @:: (1,10) /\ labelling x
```

```
==>
```

```
x @:: (1,1) \/ ... \/ x @:: (10,10)
```

Overview

11/38



- 1 CP Background
- 2 Goals**
- 3 The Solver Interface
- 4 The Search Tree
- 5 Basic Search Strategies
- 6 Search Strategy Transformers
- 7 Advanced Features
- 8 Conclusion

A Haskell framework for CP:

- ▶ model CP in Haskell
- ▶ exploit abstraction mechanisms
 - ▶ polymorphism
 - ▶ type classes
- ▶ exploit HOFs
 - ▶ building blocks
 - ▶ for complex functionality

Overview

13/38



- 1 CP Background
- 2 Goals
- 3 The Solver Interface**
- 4 The Search Tree
- 5 Basic Search Strategies
- 6 Search Strategy Transformers
- 7 Advanced Features
- 8 Conclusion

The Solver Interface

14/38

```
class Monad solver =>  
  Solver  
  solver      -- e.g. FDSolver  
  constraint  -- e.g. FDConstraint  
  term        -- e.g. FDTerm  
  label       -- state label  
  | solver -> constraint term label  
  ...
```

Solver Methods

15/38

Sequential processing:

where

```
newvarS :: solver term
  -- create fresh variable
addS    :: constraint -> solver Bool
  -- add constraint
answersS :: sm [constraint]
  -- get solved form
runS    :: sm a -> a
  -- run the solver
```

Solver Methods

16/38

Revisiting states for non-linear processing (branches):

```
markS :: solver label
      -- create label for current state

gotoS :: label -> solver ()
      -- go to state
```

Indepent of copying or trailing!

Overview

17/38



- 1 CP Background
- 2 Goals
- 3 The Solver Interface
- 4 The Search Tree**
- 5 Basic Search Strategies
- 6 Search Strategy Transformers
- 7 Advanced Features
- 8 Conclusion

The Search Tree

18/38

```
data Tree (s :: * -> *) c t
  = Fail
  | Return
  | Add c (Tree s c t)
  | NewVar (t -> Tree s c t)
  | Try (Tree s c t) (Tree s c t)
  | Label (s (Tree s c t))
```

Composing Trees

19/38

Replacing all Return nodes with another tree:

```
extendTree
  :: Monad s => Tree s c t -> Tree s c t -> Tree s c t
Fail      `extendTree` k = Fail
Return   `extendTree` k = k
Try m n  `extendTree` k =
    Try (m `extendTree` k) (n `extendTree` k)
Add c m  `extendTree` k =
    Add c (m `extendTree` k)
NewVar f `extendTree` k = ...
Label m `extendTree` k = ...
```

Declarative Sugar

20/38

```
true      = Return
```

```
false     = Fail
```

```
x \ / y   = Try x y
```

```
x /\ y    = x 'extendTree' y
```

```
disj      = foldr1 (\/)  -- t1 \ / ... \ / tn
```

```
conj      = foldr1 (/ \) -- t1 /\ ... /\ tn
```

```
exists    = NewVar
```

2-Queens

21/38

Put 2 queens on a 2x2 chess board:

```

queens = exists $ \q1 ->
        exists $ \q2 ->
            q1 @:: (1,2)      /\
            q2 @:: (1,2)      /\
            q1 @\= q2         /\
            q1 @\= (q2 @+ 1) /\
            q2 @\= (q1 @+ 1) /\
            label [q1,q2]

```

n-Queens

22/38

Put n queens on an $n \times n$ chess board:

```
queens n = exist n $ \queens ->
    queens 'allin' (1,n) /\
    alldifferent queens /\
    diagonals queens /\
    label queens
```

```
allin xs r      =
  conj [x @:: r | x <- xs]
alldifferent xs =
  conj [x @\= y | x:ys <- tails xs
        , y <- ys]
...

```

Overview

23/38



- 1 CP Background
- 2 Goals
- 3 The Solver Interface
- 4 The Search Tree
- 5 Basic Search Strategies**
- 6 Search Strategy Transformers
- 7 Advanced Features
- 8 Conclusion

Search Strategies

24/38

Search Strategy:

- ▶ explore the search tree
- ▶ solutions: path from root to Return

```
primSearch
  :: (Solver s c t l,
      SearchQueue sq (l,Tree s c t))
=> sq
-> Tree s c t
-> s [[c]]
```

Search Strategies

25/38

```
primSearch sq Fail      = continue sq
primSearch sq Return    =
  do sol  <- answerS
     sols <- continue sq
     return $ sol : sols
primSearch sq (Try left right) =
  do l <- markS
     continue (pushQ sq [(l,left),(l,right)])
primSearch sq (Add c t) =
  do b <- addS c
     if b then primSearch sq t
        else continue sq
...
```

Search Strategies

26/38

```
continue sq
  | isEmptyQ sq = return []
  | otherwise   = let ((l,tree),sq') = popQ sq
                   in do gotoS l
                       primSearch sq' tree
```

- ▶ search strategy = order of nodes visited
- ▶ determined by search queue

Search Queues

27/38

Generic interface for Search Queues:

```
class SearchQueue sq e | sq -> e where  
    emptyQ    :: sq  
    isEmptyQ  :: sq -> Bool  
    popQ      :: sq -> (e, sq)  
    pushQ     :: sq -> [e] -> sq
```

Search Queues

28/38

Search Queue instances determine search strategies:

```
-- LIFO
instance SearchQueue [e] e where ...

dfSearch = primSearch []

-- FIFO
instance SearchQueue (Seq e) e where ...

bfSearch = primSearch Data.Sequence.empty
```

Overview

29/38



- 1 CP Background
- 2 Goals
- 3 The Solver Interface
- 4 The Search Tree
- 5 Basic Search Strategies
- 6 Search Strategy Transformers**
- 7 Advanced Features
- 8 Conclusion

Strategy Transformers

30/38

Transformations of basic search strategies:

- ▶ depth-limited search
- ▶ node-limited search
- ▶ limited discrepancy search (limited right branches)
- ▶ ...

Strategy Transformers

31/38

```
class SearchTransformer st stg stn
  | st -> stg stn
where
  initSearch :: st -> (stg,stn)
  visitLeft, visitRight :: st -> stn -> stn
  visitNext :: st -> stg -> stn
             -> sq -> Tree s c t -> s [[c]]
```

+ slightly adjusted primSearch

Strategy Transformers

32/38

```
-- depth-bounded transformer  
data DBST = DBST Int  
  
instance SearchTransformer DBST () Int where  
  initSearch (DBST n) = ((),n)  
  visitLeft _ n = n - 1  
  visitRight = visitLeft  
  visitNext st stg stn sq tree  
    | n == 0      = continue st stg sq  
    | otherwise  = primSearch st stg stn sq tree
```

Overview

33/38



- 1 CP Background
- 2 Goals
- 3 The Solver Interface
- 4 The Search Tree
- 5 Basic Search Strategies
- 6 Search Strategy Transformers
- 7 Advanced Features**
- 8 Conclusion

Advanced Features

34/38

Our framework also contains:

- ▶ composable search transformers
- ▶ iterative *deepening*
- ▶ optimization
 - ▶ restart
 - ▶ threaded

Overview

35/38



- 1 CP Background
- 2 Goals
- 3 The Solver Interface
- 4 The Search Tree
- 5 Basic Search Strategies
- 6 Search Strategy Transformers
- 7 Advanced Features
- 8 Conclusion**

Conclusions

36/38

We have a CP framework that:

- ▶ ...is parametric in the underlying constraint solver
- ▶ ...does not depend on either copying or trailing
- ▶ ...allows for arbitrary queueing strategies
- ▶ ...offers search transformers
- ▶ ...which can be stacked for complex heuristics
- ▶ and optionally optimizes an objective function

Extensions and Generalizations

- ▶ interface (FFI) to state-of-the-art solvers, like Gecode (C++)
- ▶ explore performance characteristics
- ▶ interface to CHR (custom solver framework)
- ▶ generalize to (non-CP) search framework

Thank you!

38/38

Questions?