AINT503: Constraint Programming (2)

Dr Torsten Anders
Prof Eduardo Miranda
Interdisciplinary Centre for Computer Music Research (ICCMR)
University of Plymouth
http://cmr.soc.plymouth.ac.uk/

2 December 2008

Goals of this Session

- Meet further application examples of constraint programming
- Look “under the hood”: how Oz constraint solver finds solutions
- First brief background info on underlying programming paradigms
  - Functional programming: higher-order programming
  - Declarative concurrent programming

Application Examples (Homework)

Discuss in groups the constraint programming application examples you found out in your homework. Afterwards you present a summary to the whole class.

At first, find volunteers for each of the following functions

- Facilitator for the discussion
- Minute taker
- Presenter (to present at the end)

Why Discussing the Search Process?

- Constraint programming greatly simplifies solving combinatorial problems
  - User only specifies all constraints the solution should fulfil
  - A constraint solver finds solution(s) for a constraint satisfaction problem (CSP) via search
- However, reasonably efficient search vital to make system useful in practice
- This talk discusses how constraint programming in Mozart works 'under the hood'

Side Note: Functions are Procedures (. . . but often more convenient)

Twice function call: returns a value

```
X = {Twice 3}
```

Equivalent Twice procedure call: no return value

```
{Twice 3 X}
```

Twice function definition: argument for return value implicit

```
fun {Twice X} 2 * X end
```

Equivalent Twice procedure definition

```
proc {Twice X Y} Y = 2 * X end
```

Side Note: Functions are Procedures (...but often more convenient)

Twice function call: returns a value

```
X = {Twice 3}
```

Equivalent Twice procedure call: no return value

```
{Twice 3 X}
```

Twice function definition: argument for return value implicit

```
fun {Twice X} 2 * X end
```

Equivalent Twice procedure definition

```
proc {Twice X Y} Y = 2 * X end
```

Problem: How to Simplify Processing of Hierarchic Data

Problem to solve

We often want to process hierarchically nested data (linked list, tree, graph)

Examples

- Filter out elements which meet a specific condition
- Sort elements

Question

How can we avoid to always program some list/tree/graph traversal from scratch whenever we want to process such data?

Problem: How to Simplify Processing of Hierarchic Data

Problem to solve

We often want to process hierarchically nested data (linked list, tree, graph)

Examples

- Filter out elements which meet a specific condition
- Sort elements

Question

How can we avoid to always program some list/tree/graph traversal from scratch whenever we want to process such data?

Higher-Order Programming Concept

Functional programming: first-class functions/procedures

Programs processing programs Procedures can process and create first-class procedures.

A procedure expecting other procedures as argument or returning procedures is called a *higher-order* procedure.

Higher-Order Programming Examples I

`Filter` returns list elements for which a given function returns true

```
{Filter [0 1 2 3 4 5 6 7] IsEven}

% returns [0 2 4 6]
```

Higher-Order Programming Examples II

Sort sorts a list according to a given comparison function (here an anonymous function)

```
{Sort [1 5 3 2 0 7]
 fun {$ X Y} X < Y end}

% returns [0 1 2 3 5 7]
```

Higher-Order Function Definition

Higher-order functions (like `Filter`) are defined like any other function

```
fun {Filter Xs F}
   if Xs == nil then nil
   else X|Xr = Xs in
      if {F X}
      then X | {Filter Xr F}
      else {Filter Xr F}
      end
   end
end
```

Systems Supporting First-Class Functions (Selection)

First-class functions are recently moving to mainstream

Functional programming languages Lisp/Scheme, Haskell, ML/OCaml/F#

Mainstream languages JavaScript, Python, Ruby, Perl, C#

Concurrent Programming Is Known to be Hard

Question

Scenario: multiple concurrent computations access a bank account. Which problems can occur?

Discuss this question in small groups, then report your results to whole class.

concurrency + state = complex programming

Combination of concurrent programming and stateful programming (imperative programming) is difficult to control

Concurrent Programming Is Known to be Hard

Question

Scenario: multiple concurrent computations access a bank account. Which problems can occur?

Discuss this question in small groups, then report your results to whole class.

concurrency + state = complex programming

Combination of concurrent programming and stateful programming (imperative programming) is difficult to control

Declarative Concurrency: Concurrency Made Easy I

- Partial values (logic variables): variable can be
  - free (nothing is known about its value)
  - partially determined (e.g. it is a list with undetermined elements)
  - fully determined
  - Constraints add information about variable values (e.g., unification, numeric constraints)
- Concurrency: computations executed in multiple *threads* (created explicitly)

Declarative Concurrency: Concurrency Made Easy II

- Synchronisation of threads on variables:
    - Thread *blocks* if logic variables used in a statement of the thread lack required information
    - Another thread might provide this information – threads communicate via (dataflow) variables
- First-class procedures: procedures are first-class values, and support lexical scope

Declarative Concurrency: Concurrency Made Easy III

Note

- Declarative concurrency is highly expressive programming model: it greatly simplifies writing concurrent programs with massive number of threads
- Reason: stateless concurrency (no conflicts of shared resources can occur)

Example: Erlang programming language

Model of programming language *Erlang* is similar to this concurrency model.

Ericsson uses Erlang successfully in several Ericsson products for telecommunication

From Declarative Concurrency to Constraint Programming

- No support for search in concurrent programming model
- Adding *computational spaces* provides support for speculative computations and search
- In spaced-based constraint model, search is encapulated
- Alternative to *backtracking*-based search (as in Prolog) – backtracking not feasible with concurrency and interoperating with external world
- We only study simplified view on the constraint model based on spaces

Propagate and Search: Constraint Propagation and Search Complement Each Other

Constraint Propagation

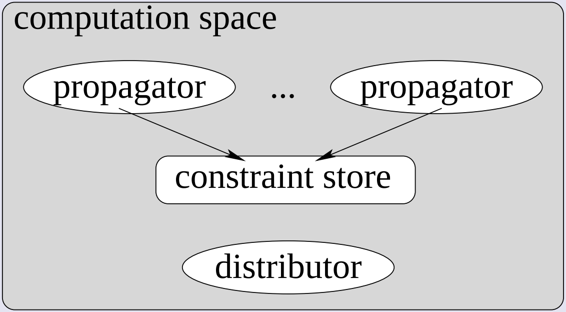
- Reducing variable domains reduces search space
- Constraint propagation performs local deductions: reduces variable domains without removing solutions

Search

- Performs decisions which potentially can fail
- Making the right decisions and making decisions in the right order important for efficiency

The Computation Space

A *computation space* encapsulates information available on a CSP at a certain stage during the search process



The Constraint Store

- Constraint store: stores information on variable values – conjunction of basic constraints
- Basic constraint: representation of information on partial value of a single variable. Example for finite domain integers (FD ints): two forms possible
  - $X \in \mathbf{D}$ means $\mathbf{D}$ (a set of natural numbers) is *domain* of X, special case $X \in \{n\}$ means $X = n$ (X determined to n)
  - $X = Y$ means X and Y are equal (unified) – both can be undetermined

Example constraint store

$$X \in \{1, \ldots, 5\} \wedge Y = 7 \wedge Z = X$$

Constraint Propagation I

Propagator

- Any more complex constraint (non-basic constraint) expressed by propagator
- A propagator is a concurrent agent
- Propagator aims to add information (i.e. narrows variable domains) which is
  - consistent with constraint store
  - follows from constraint expressed by propagator
- Implemented by algorithm usually highly optimised for its specific constraint

Constraint Propagation II

Example: propagator $X < Y$ narrows domain of X and Y

- Store before propagation: $X \in \{1, \ldots, 5\} \wedge Y \in \{1, \ldots, 5\}$
- Store after propagation: $X \in \{1, \ldots, 4\} \wedge Y \in \{2, \ldots, 5\}$

Constraint Propagation III

Note

Constraint propagation does not necessarily lead to a solution

Example: propagators $X \neq Y$, $X \neq Z$, and $Y \neq Z$ cannot reduce the domains further

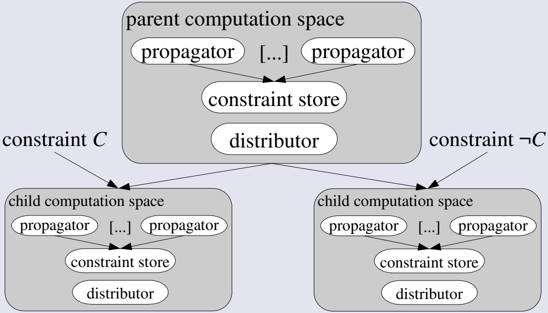
$$X \in \{1,2\} \wedge Y \in \{1,2\} \wedge Z \in \{1,2\}$$

Stable space

No further propagation is possible: hosting computation space is *stable*

Constraint Distribution: Making Decisions I

Constraint distribution creates two child spaces which are the result of two complementary decisions (expressed by the two added constraints C and $\neg C$)



Constraint Distribution: Making Decisions II

- Constraint distribution (branching): proceeds to spaces easier to solve, but with same solution set (search)
- Distributor: concurrent agent
  - Waits until space is stable
  - Then creates two child spaces (copies of parent space)
  - Add some basic constraint C to store of one child space and its complement $\neg C$ to store of other child space
  - Important: choose such C and $\neg C$ which trigger further constraint propagation

Combination of constraint propagation and distribution is a complete search method for solving CSPs

Order of Decisions

Question

You are buying clothing for an important event, say, a pair of shoes, a suit, a tie, and a shirt. What can be the criteria for your decisions? In which order do you make your decisions for your purchases. Why in this order?

Think on your own, then later report to group.

Distribution Defines Variable and Value Orderings I

Variable ordering

- Variable ordering: order in which variables are visited during the search process
- Variable ordering has great impact on efficiency (size of resulting search tree: search space)
- Suitable variable ordering is problem dependent

Static vs. dynamic variable ordering

Variable ordering is either fixed before the search starts (static), or computed during the search process (dynamic) – distribution is dynamic

Distribution Defines Variable and Value Orderings II

First-fail principle

- Common principle for designing dynamic variable orderings
- Essence: deal with hard cases first – if failure is inevitable, better fail early
- Typical approaches: first visit variable with smallest domain, or variable with most constraints applied to it

Distribution Defines Variable and Value Orderings III

Value ordering

- Order in which variable domain values are considered during the search process (speculative computation: values may fail and others may be tried later)
- Has impact on efficiency, but also on quality of first solution
- Common principle: succeed-first principle or best-first heuristic

First-Fail Distribution: a Musical Example I

Example distribution strategy: first-fail

Select variable with smallest domain, and determine it to its left-most domain value

CSP example

All-interval series: `http://strasheela.sourceforge.net/strasheela/doc/Example-AllIntervalSeries.html`

First-Fail Distribution: a Musical Example II

CSP all-distance series definition (length 4)

$Xs :=$ list of 4 FD ints, each with domain $\{0, \ldots, 3\}$

$Dxs :=$ list of 3 FD ints, each with domain $\{1, \ldots, 3\}$

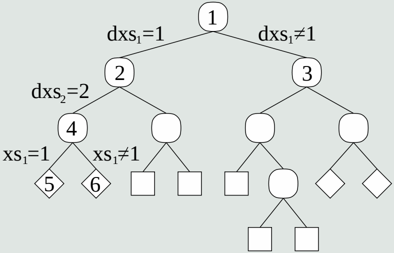
$$\bigwedge_{i=1}^{3} Dxs_i = |Xs_i - Xs_{i+1}|$$

$\wedge \; distinct(Xs)$

$\wedge \; distinct(Dxs)$

First-Fail Distribution: a Musical Example III

All-distance series: search tree for all solutions of length 4



Space	Domains of elements in *Xs*	Domains of elements in *Dxs*
1	$[\{0,\ldots,3\},\ldots,\{0,\ldots,3\}]$	$[\{1,\ldots,3\},\{1,\ldots,3\},\{1,\ldots,3\}]$
2	[{0, ..., 3}, ..., {0, ..., 3}]	[1, {2, 3}, {2, 3}]
3	[{0, ..., 3}, ..., {0, ..., 3}]	[{2, 3}, {1, 2, 3}, {1, 2, 3}]
4	[{1, 2}, {1, 2}, {0, 3}, {0, 3}]	[1, 2, 3]
5	[1, 2, 0, 3]	[1, 2, 3]
6	[2, 1, 3, 0]	[1, 2, 3]

User-Defined Variable and Value Orderings

- User can freely define distribution strategies
- Defining a distribution strategy means defining shape of search tree: i.e., a variable and value ordering
- Next distribution step is always decided only when it is required: dynamic ordering
- Distribution strategies can be changed independently of problem definition

Distribution Strategy Definition I

- Distribution strategy can be defined 'from scratch' (cf. [Schulte, 2002])
- More convenient: definition with a higher-level interface
- Simple interface example expects two first-class functions as arguments (see next slide)

Distribution Strategy Definition II

Order: which variable is distributed (variable ordering)

Boolean function expecting two variables. Returns *true* if first variable should be visited before the second

Value: how does distribution strategy effect domain of selected variable (value ordering)

Function expecting a variable, and returning a reduced domain specification for this variable (usually a single domain value)

Example: first-fail distribution strategy definition

Order `fun {$ X Y} {DomSize X} =< {DomSize Y} end`
Value `fun {$ X} {DomMin X} end`

Principles for Efficient Distribution Design

- An efficient distribution strategy results in a relatively small search tree (little amount of failure)
- Constraint propagation never causes a fail (no redundant work)
- An efficient distribution strategy keeps distribution steps at minimum, i.e. helps constraint propagation to do most of the work
- Common example: first-fail principle (see above)

Implementations of the Space-Based Constraint Model

- Mozart: implementation of the multi-paradigm programming language Oz, `http://www.mozart-oz.org/`
- Gecode: C++ library, `http://www.gecode.org/`
- Gecode bindings exist for several languages, including Java (Gecode/J, `http://www.gecode.org/gecodej/`)

Recommended Reading on Constraint Programming I

Constraint programming (CP) in general

- Roman Barták (1998). *On-Line Guide to Constraint Programming*. `http://kti.ms.mff.cuni.cz/~bartak/constraints/index.html` – gentle introduction to CP
- Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press. – general overview of the field with many CSP examples
- Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann. – explains various constraint solving algorithms

Recommended Reading on Constraint Programming II

Constraint programming model used by Mozart I

- van Roy, P. and S. Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press. – highly recommended programming textbook in general (try googling computer programming textbook), Chap. 12 explains space-based constraint model
- Schulte, C. (2002). *Programming Constraint Services*. Springer-Verlag. – most detailed explanation of the space-based constraint model, advanced text

Recommended Reading on Constraint Programming III

Constraint programming model used by Mozart II

- Tutorial of Oz. `http://www.mozart-oz.org/documentation/tutorial/index.html` – an introduction to Oz and Mozart
- Finite Domain Constraint Programming in Oz. `http://www.mozart-oz.org/documentation/fdt/index.html` – a Tutorial introduction to constraint programming in full Oz

Summary

- Application examples
- Background
  - Higher-order programming
  - Declarative concurrency
- The constraint model based on computational spaces
  - Propagate-and-search
  - User can define variable and value ordering (distribution strategy)