MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.891 Spring 2006
Problem Set 7

Issued: Fri. 14 April 2006 Due: Wed. 3 May 2006

Reading: Logic Puzzles in SICP pp.419--420

Code: The contents of the code subdirectory on the ps7 website.

Constraint-Propagation Systems

Over the last few weeks we have been discussing constraint-propagation systems, for several reasons. They are composite systems, with subsystems for truth-maintenance, for arithmetic and algebra on values, and for propagation. Interesting systems are made by combining such subsystems to make a coherent whole. Constraint-propagation systems are also a paradigm of the idea that effective problem-solving strategies are finitary: indeed, people often draw diagrams to express finitary representations of problems that can be attacked by manipulating and annotating the diagrams. Finally, I (GJS) am particularly fond of constraint-propagation systems. In this problem set we invite you to build a simple algebra of finite sets that can be used with the constraint propagator (and tms, etc.) that we provide, and apply the resulting system to some simple problems, such as the Sudoku puzzles.

This is a real engineering-design problem set. We are not asking discrete questions for you to answer. You will have to read lots of code and design and implement a subsystem to be added to this code. This may be harder than we anticipate, so be sure to ask for help if you get into trouble.

The system we want you to work with is quite complicated, so we can give only a few highlights in this introduction. To understand the details you will have to read the code. However, a nice program should be as much fun to read as it is to extend.

Using the Constraint Propagator

To use a constraint propagator we must make a network, add connectors and constraints to the network, and propagate assumptions about the values assigned to the connectors around the network.

Connectors are created with the procedure cp:make-connector. For example:

```
(cp:make-connector 'esh ckt2)
```

Here we make a connector that is part of the network ckt2. It has a name, esh, for it to report when asked. However, to be able to reference this connector we also define a Scheme variable to have the connector as its value:

```
(define esh (cp:make-connector 'esh ckt2))
```

That the Scheme variable is the same name as the name passed to the connector-constructor is inessential -- this is just for our convenience. The procedure cp:make-connector takes an optional third argument -- the value model for the connector -- to be supplied if the connector should not have the default value model for the network. We will discuss this later.

Constraints are created among connectors using the procedure cp:make-constraint:

```
(cp:make-constraint adder 'KVL-E ckt2 esh vs esl)
```

This makes an adder constraint, named KVL-E in the network ckt2. It enforces the condition that the value of the connector referred to by the Scheme variable esh is the sum of the values of the connectors referred to by vs and esl.

The value of the cp:make-constraint expression is the constraint object. If we want to refer to that constraint we would grab that value and name it.

Given a connector

```
(define zero-i (cp:make-connector 'zero-i ckt2))
```

we can assign it a value (here zero) by assumption:

```
(cp:assume-value zero-i 0)
```

The assignment is justified in the TMS as a logical premise, which can be retracted by the command

```
(cp:retract-value zero-i)
```

A new assignment may stimulate a constraint that the connector participates in to make further deductions.

If a connector already has a supported assignment when a new value for the connector is assumed or deduced there are several possibilities. If the new deduction is redundant -- just a rededuction of an existing assignment by the same constraint from the same antecedents -- then no action is taken.

If the new deduction is not redundant then the value of the supported assignments must be reconciled with the new value. The results of the reconciliation are determined by components of the value model of the connector.

1. If the new value is equivalent to the value of another supported assignment then the supported assignment is given an additional justification.

2. If the new value is implied by or implies the value of another supported assignment, a new assignment is added for the new value and appropriate justifications are constructed.

3. If the new value is clearly incompatible with a supported assignment the TMS contradiction node is supported and a contradiction is signaled.

4. If none of the above hold the value model may post an equation to be processed later by the post-propagation-hook.

At any time a connector may be queried to determine if it has a supported assignment. For example, if the connector referred to by esh has a value the following will be true:

```
(cp:has-value? esh)
```

If it has one, the value may be found by:

```
(cp:value-of esh)
```

The chain of deductions that led to the value can be seen by

```
(cp:why? esh)
```

This procedure produces a standard four-column Suppes-style proof.

Value Models

In this problem set your main job will be to construct a value model for connectors that hold finite sets as values, and to construct constraints that use these values to solve a non-trivial problem. The value model of a connector provides components for manipulating the values that appear as assignments to that connector.

A value model must provide the following components:

A membership predicate.
Must be true of any value assigned to the connector.
This is a gatekeeper, intended to help catch bugs.

An equality test.
Determines if two values are equivalent, for reconciliation.
(See 1 above.)

A best-value predicate.
Gives the cp:value-of procedure a way of choosing among the supported assignments to the connector. For example, in numerical constraints a number is a better value than an expression whose value is that number. Also, if numerical ranges are the possible values, and if there is an assignment to a range that is entirely contained in the range of another assignment, the assignment of the smaller range is the better one. This test is also used to determine if one supported assignment implies another. (See 2 above.)

A coincidence handler.
This procedure is called when a connector that already has a supported assignment is given a non-redundant new value. It takes the connector, the new value, and the justification for that new value. The coincidence handler must determine if there is a subsumption to be justified, a contradiction to be signaled, or an equation to be posted.

A post-propagation processor.
If the coincidence handler posts equations (or inequalities, or other "facts" that are not represented as parts of the constraint network) such that the aggregate of those equations may be used to make further deductions and assignments or contradictions, the value model must provide a post-propagation processor to do that work.

A rather elaborate value model can be found in the file "handler.scm." This value model is used for numbers and algebraic expressions that represent them. It makes equations, when necessary, and solves them. The solver is in a separate file "solve.scm," but that is not important for your problem. The value model needed by the constraint propagator is returned by the message processor at the end of the file as follows:

```
(cp:make-value-model numerical-quantity?
                     numerical-quantities-equal?
                     choose-numerical-quantity
                     coincidence-handler
                     post-propagation-processor)
```

You can see how the algebraic subsystem is interfaced to the constraint propagator in the numerical-examples.scm file and in the electric.scm file.

YOUR JOB

You have been "hired" to build a value model to be used with the propagator we provided. The particular value model must give you a way to deal with values that are finite sets. You may use any representation of finite sets you like. (For small sets, an unordered list of distinct elements works fine). You need to make the best-value predicate so that a smaller set is better than a bigger one. You also need to make a coincidence handler, so that subsets and supersets of a supported value are compatable and make appropriate justifications, while signaling contradictions when appropriate.

You must demonstrate your constraint-propagator code with a non-trivial application of your choice. We suggest that you do something simpler, such as the multiple-dwelling Logic puzzle in SICP. Note: The guts of the constraint-propagator solution will probably not look much like the AMB interpreter solution to the problem without alot of work.

Unless you have lots of time you should not try to do Sudoku, because the constraints are not obvious.

Good skill!