Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2009-002 January 26, 2009

The Art of the Propagator

Alexey Radul and Gerald Jay Sussman

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

The Art of the Propagator

Alexey Radul & Gerald Jay Sussman
{axch,gjs}@mit.edu

December 13, 2008

Abstract

We develop a programming model built on the idea that the basic computational elements are autonomous machines interconnected by shared cells through which they communicate. Each machine continuously examines the cells it is interested in, and adds information to some based on deductions it can make from information from the others. This model makes it easy to smoothly combine expression-oriented and constraint-based programming; it also easily accommodates implicit incremental distributed search in ordinary programs.

This work builds on the original research of Guy Lewis Steele Jr. [19] and was developed more recently with the help of Chris Hanson.

[0]This document, together with complete supporting code, is available as an MIT CSAIL Technical Report via http://dspace.mit.edu.

Contents

1 Introduction

Conventional programming languages allow the value stored in a place to come from only one source. We deliberately use "place" non-technically: we mean the variables of the language, the components (slots, fields, members, what have you) of compound data structures (pairs, arrays, objects, etc), and the implicit transient storage given to the intermediate results of subexpressions. The "source," then, is whatever corresponding construct (commonly: expression) produces the value that is to be put in the place — it is nearly axiomatic that for each place there is exactly one source.[1]

What happens if we relax this restriction, and allow the places to receive values from multiple sources? Then an individual source need no longer be responsible for computing the complete value that goes into a place. Indeed, some partial knowledge about a value can already be useful to some client, and can perhaps be augmented by some other source later (which perhaps used that partial knowledge to deduce the refinement!). Also, if places can accept values from multiple sources, we need not decide in advance which computation will end up producing the value that goes into a place. We can instead construct systems whose information flow depends on how they are used.

We use the idea of a *propagator network* as a computational metaphor for exploring the consequences of allowing places to accept information from multiple sources. We study the design choices available within this metaphor and their consequences for the resulting programming system.

2 Propagators

Our computational model is a network of autonomous machines, each continuously examining its inputs and producing outputs when possible. The inputs and outputs of a machine are shared with other machines so that the outputs of one machine can be used as the inputs for another. The shared communication mechanism is called a *cell* and the machines that they interconnect are called *propagators*. As a consequence of this viewpoint computational mechanisms are naturally seen as wiring diagrams.

As an elementary example of a propagator network we can implement

[1] In the presence of mutation, we mean places to be interpreted in spacetime: the value occupying a variable during any particular period when that variable is not mutated comes from just one source, and when the variable is mutated, the new value installed there also comes from just one (presumably different) source.

the improvement step in Heron's method of computing the square root of a number in cell x. If we have a guess for the square root in cell g we can put a better guess in cell h by the formula $h = (g + x/g)/2$. We can describe such a network in a conventional programming language (Scheme) by the following ugly code:

```
(define (heron-step x g h)
  (compound-propagator (list x g)       ;inputs
    (lambda ()                          ;how to build
      (let ((x/g (make-cell))
            (g+x/g (make-cell))
            (two (make-cell)))
        (divider x g x/g)
        (adder g x/g g+x/g)
        ((constant 2) two)
        (divider g+x/g two h)))))
```

This Scheme procedure takes three cells as inputs. It creates three additional cells to be internal channels. It then specifies four propagators that will carry out the computation, giving each one input cells and an output cell. (But the constant propagator has only an output cell, `two`, in which to smash the number 2.) The Scheme program is a wiring diagram that constructs the network so specified.

To use this network we need to make cells to pass in to this network constructor; we need to give some cells values, and we need to access the values in other cells:

```
(define x (make-cell))
(define guess (make-cell))
(define better-guess (make-cell))

(heron-step x guess better-guess)

(add-content x 2)
(add-content guess 1.4)
(content better-guess)
1.4142857142857141
```

So this simple propagator network gave a reasonable answer.

Things get considerably more complicated when we want to iterate the improvement to get a better answer. The propagator wiring diagram is more analogous to assembly language than the expression languages that we are used to.

What is interesting is the mechanism of iteration (or recursion) that we use. A propagator does something only if it has inputs worth working on. A compound propagator builds its body when presented with values

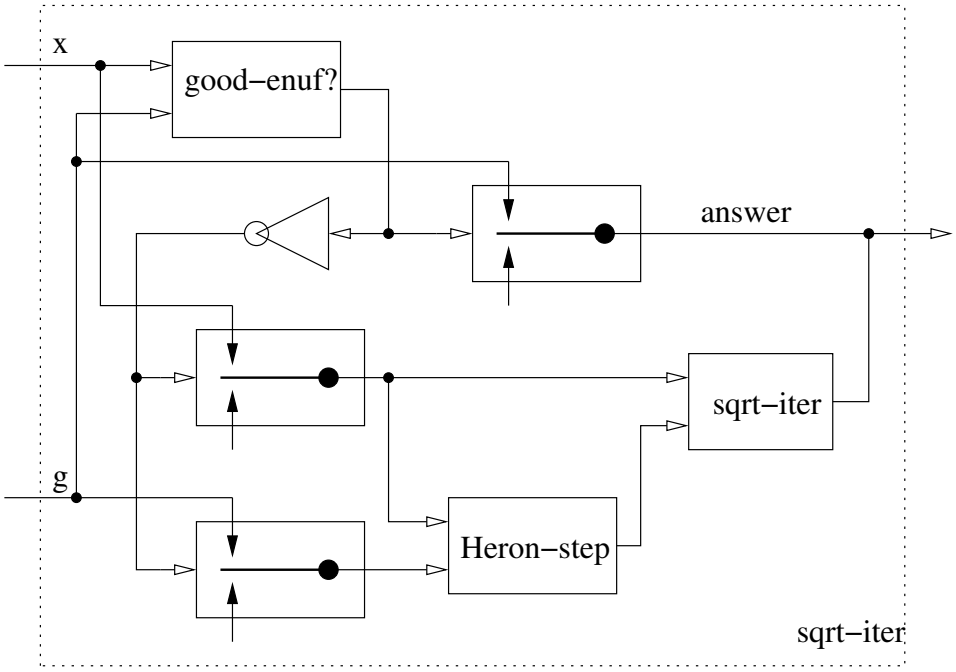

Figure 1: A wiring-diagram description of the `sqrt-iter` network.

in all of the cells declared as its inputs. The `sqrt-iter` propagator (see Figure 1) uses switches to connect inputs to the `heron-step` and the recursive call only if the `good-enuf?` test is not satisfied. This is not the only way to do recursion, but it is a good start.

```
(define (sqrt-network x answer)
  (compound-propagator x
    (lambda ()
      (let ((one (make-cell)))
        ((constant 1.) one)
        (sqrt-iter x one answer)))))
```

```
(define (sqrt-iter x g answer)
  (compound-propagator (list x g)
    (lambda ()
      (let ((done (make-cell))
            (not-done (make-cell))
            (x-if-not-done (make-cell))
            (g-if-not-done (make-cell))
            (new-g (make-cell)))
        (good-enuf? g x done)
        (switch done g answer)
        (inverter done not-done)
        (switch not-done x x-if-not-done)
        (switch not-done g g-if-not-done)
        (heron-step x-if-not-done g-if-not-done new-g)
        (sqrt-iter x-if-not-done new-g answer)))))

(define (good-enuf? g x done)
  (compound-propagator (list g x)
    (lambda ()
      (let ((g^2 (make-cell))
            (eps (make-cell))
            (x-g^2 (make-cell))
            (ax-g^2 (make-cell)))
        ((constant .00000001) eps)
        (multiplier g g g^2)
        (subtractor x g^2 x-g^2)
        (absolute-value x-g^2 ax-g^2)
        (<? ax-g^2 eps done)))))
```

With this program we get a rather nice value for the square root of 2 (even though this end test is not a good one from the numerical analyst's perspective).

```
(define x (make-cell))
(define answer (make-cell))

(sqrt-network x answer)

(add-content x 2)
(content answer)
1.4142135623746899
```

Making this work

So, what does it take to make this basic example work? We need some cells and some propagators, and the `compound-propagator` procedure used above. Let us start with the cell. In this simplest system, each cell stores a single value (or a distinguished token meaning the absence of a value) and maintains a set of propagators that need to be alerted if

the cell's content changes.

```
(define nothing (list '*the-nothing*))

(define (nothing? thing)
  (eq? thing nothing))

(define (make-cell)
  (let ((neighbors '()) (content nothing))
    (define (new-neighbor! new-neighbor)
      (if (not (memq new-neighbor neighbors))
          (begin
            (set! neighbors (cons new-neighbor neighbors))
            (alert-propagators new-neighbor))))
    (define (add-content increment)
      (cond ((nothing? increment) 'ok)
            ((nothing? content)
             (set! content increment)
             (alert-propagators neighbors))
            (else
             (if (not (default-equal? content increment))
                 (error "Ack! Inconsistency!")))))
    (define (me message)
      (cond ((eq? message 'new-neighbor!) new-neighbor!)
            ((eq? message 'add-content) add-content)
            ((eq? message 'content) content)
            (else (error "Unknown message" message))))
    me))

(define (new-neighbor! cell neighbor)
  ((cell 'new-neighbor!) neighbor))

(define (add-content cell increment)
  ((cell 'add-content) increment))

(define (content cell)
  (cell 'content))
```

Each cell stores its content and the set of propagators that have announced, by invoking the `new-neighbor!` procedure, that they are interested in that cell's content.

In these cells, the designated marker `nothing` means "I know nothing about the value that should be here," and any other value x means "I know everything about the value that should be here, and it is x." The permitted interface for changing a cell's content is to add more information to it by calling the `add-content` procedure. In this simple system, only four things can happen: adding nothing to the cell says "I don't know anything about what should be here," so it's always ok and does

not change the cell's content at all. Adding a raw value to a cell amounts to saying "I know that the content of this cell should be exactly x." This is fine if the cell knew nothing before, in which case it now knows its content is x, and alerts anyone who might be interested in that. This is also fine if the cell already knew its content was[2] x, in which case the addition taught it nothing new (and, notably, its neighbors don't need to be alerted when this happens). If, however, the cell already knew that its content was something other than x, then something is amiss. The only resolution available in this system is to signal an error.

A propagator in this system is just a nullary Scheme procedure that the scheduler runs from time to time. The only thing we need to do to bless it as a propagator is to attach it as a neighbor to the cells whose contents affect it, and schedule it for the first time:

```
(define (propagator neighbors to-do)
  (for-each (lambda (cell)
              (new-neighbor! cell to-do))
            (listify neighbors))
  (alert-propagators to-do))
```

This procedure arranges for the thunk `to-do` to be run at least once, and asks each cell in the `neighbors` argument to have `to-do` rerun if that cell's content changes.

Most of our primitive propagators just take a normal Scheme function, wait for the cells containing the arguments to acquire some interesting values, and then apply that function to those values and write the result into a cell for the output. The procedure `lift-to-cell-contents` ensures that if any cell contents are still `nothing` the result is `nothing`,

```
(define (lift-to-cell-contents f)
  (lambda args
    (if (any nothing? args)
        nothing
        (apply f args))))
```

and `function->propagator-constructor` makes a device that will construct such propagators when desired.

---

[2]Equality is a tough subject

```
(define (function->propagator-constructor f)
  (lambda cells
    (let ((output (car (last-pair cells)))
          (inputs (except-last-pair cells))
          (lifted-f (lift-to-cell-contents f)))
      (propagator inputs  ; The output isn't a neighbor![3]
        (lambda ()
          (add-content output
            (apply lifted-f (map content inputs))))))))
```

With this machinery, we can define a nice array of primitive propagators that just implement Scheme functions:

```
(define adder (function->propagator-constructor +))
(define subtractor (function->propagator-constructor -))
(define multiplier (function->propagator-constructor *))
(define divider (function->propagator-constructor /))
;;; ... for more primitives see Appendix A
```

Constants also turn out quite nicely:

```
(define (constant value)
  (function->propagator-constructor (lambda () value)))
```

To round out our collection of primitive propagators, we implement the switches we used for recursion as a special case of a two-armed conditional device,

```
(define (switch predicate if-true output)
  (conditional predicate if-true (make-cell) output))
```

which itself is a relatively straightforward wrapper around Scheme's native `if`, except that it needs to take some care in the case that the cell containing the predicate has no content yet.

```
(define (conditional p if-true if-false output)
  (propagator (list p if-true if-false)
    (lambda ()
      (let ((predicate (content p)))
        (if (nothing? predicate)
            'done
            (add-content output
                         (if predicate
                             (content if-true)
                             (content if-false))))))))
```

[3]Because the function's activities do not depend upon changes in the content of the output cell.

Finally, a compound propagator is implemented with a procedure that will construct the propagator's body on demand. We take care that it is constructed only if some neighbor actually has a value, and that it is constructed only once:

```
(define (compound-propagator neighbors to-build)
  (let ((done? #f) (neighbors (listify neighbors)))
    (define (test)
      (if done?
          'ok
          (if (every nothing? (map content neighbors))
              'ok
              (begin (set! done? #t)
                     (to-build)))))
    (propagator neighbors test)))
```

3 Partial Information

One way that the propagation model of computation differs from the expression-evaluation model is that a single cell can get information from multiple sources, whereas the return value of an expression can come from only one source—the expression. This difference can be seen in `sqrt-iter` because the `answer` cell can get a value from two sources. This particular case is not very interesting because only one of the alternatives is ever effective. Indeed in an expression language a conditional gives a way of choosing among exclusive alternatives.

More generally, however, if information can enter a cell from multiple different sources, it is natural to imagine each source producing some part of the information in question, and the cell being responsible for combining those parts. For example, we can imagine a network whose cells contain subintervals of the real numbers, each interval representing the set of possible values to which the cell's content is known to be restricted. The operations in the network perform interval arithmetic. In this case, if there is some redundancy in the network, a cell can get nontrivial constraining information from multiple sources; since each source authoritatively asserts that the cell's value must be within its limits, the net effect is that the intervals need to be intersected. [4, 5]

To make this example concrete, consider the famous problem of measuring the height of a building by means of a barometer. A great variety of solutions are known; let us start with dropping it off the roof and timing its fall. Then the height h of the building is given by $h = 1/2gt^2$, where g is the acceleration due to gravity and t is the amount of time the barometer took to hit the ground. We implement this as a propagator

network (that includes some uncertainty about the local g):

```
(define (fall-duration t h)
  (compound-propagator t
    (lambda ()
      (let ((g (make-cell))
            (one-half (make-cell))
            (t^2 (make-cell))
            (gt^2 (make-cell)))
        ((constant (make-interval 9.789 9.832)) g)
        ((constant (make-interval 1/2 1/2)) one-half)
        (squarer t t^2)
        (multiplier g t^2 gt^2)
        (multiplier one-half gt^2 h)))))
```

Trying it out, we get an estimate for the height of the building:

```
(define fall-time (make-cell))
(define building-height (make-cell))
(fall-duration fall-time building-height)

(add-content fall-time (make-interval 2.9 3.1))
(content building-height)
#(interval 41.163 47.243)
```

Of course, we can also measure the height of a building using a barometer by standing the barometer on the ground on a sunny day, measuring the height of the barometer as well as the length of its shadow, and then measuring the length of the building's shadow and using similar triangles. The formula is $h = s\frac{h_{ba}}{s_{ba}}$, where h and s are the height and shadow-length of the building, respectively, and h_{ba} and s_{ba} are the barometer's. The network for this is

```
(define (similar-triangles s-ba h-ba s h)
  (compound-propagator (list s-ba h-ba s)
    (lambda ()
      (let ((ratio (make-cell)))
        (divider h-ba s-ba ratio)
        (multiplier s ratio h)))))
```

and we can try it out:

```
(define barometer-height (make-cell))
(define barometer-shadow (make-cell))
(define building-height (make-cell))
(define building-shadow (make-cell))
(similar-triangles barometer-shadow barometer-height
                   building-shadow building-height)

(add-content building-shadow (make-interval 54.9 55.1))
(add-content barometer-height (make-interval 0.3 0.32))
(add-content barometer-shadow (make-interval 0.36 0.37))
(content building-height)
#(interval 44.514 48.978)
```

Different measurements lead to different errors, and the computation leads to a different estimate of the height of the same building. This gets interesting when we combine both means of measurement, by measuring shadows first and then climbing the building and dropping the barometer off it:

```
(define fall-time (make-cell))
(fall-duration fall-time building-height)

(add-content fall-time (make-interval 2.9 3.1))
(content building-height)
#(interval 44.514 47.243)
```

It turns out that in this case the upper bound for the building's height comes from the drop measurement, whereas the lower bound comes from the shadow measurement — we have a nontrivial combination of partial information from different sources.

Making this work

Let's see what we need to make this work. First, we need some code to actually do interval arithmetic.[4] Nothing fancy here, especially since we are assuming that all our intervals have positive bounds (and only implementing multiplication, because the examples don't add intervals).

```
(define (mul-interval x y)
  (make-interval (* (interval-low x) (interval-low y))
                 (* (interval-high x) (interval-high y))))
```

[4]For the concrete definitions of the data structures we use in this text, see Appendix B.

```
(define (div-interval x y)
  (mul-interval x
                (make-interval (/ 1.0 (interval-high y))
                               (/ 1.0 (interval-low y)))))

(define (square-interval x)
  (make-interval (square (interval-low x))
                 (square (interval-high x))))

(define (sqrt-interval x)
  (make-interval (sqrt (interval-low x))
                 (sqrt (interval-high x))))

(define (empty-interval? x)
  (> (interval-low x) (interval-high x)))

(define (intersect-intervals x y)
  (make-interval
   (max (interval-low x) (interval-low y))
   (min (interval-high x) (interval-high y))))
```

Then we need to make some propagators that expect intervals in input cells and write intervals to output cells. This is straightforward as well.

```
(define multiplier (function->propagator-constructor mul-interval))
(define divider (function->propagator-constructor div-interval))
(define squarer (function->propagator-constructor square-interval))
(define sqrter (function->propagator-constructor sqrt-interval))
```

Finally, we need to alter our cells to accept intervals, and merge them by intersection rather than checking that they are equal.

```
(define (make-cell)
  (let ((neighbors '()) (content nothing))
    (define (new-neighbor! new-neighbor)
      (if (not (memq new-neighbor neighbors))
          (begin
            (set! neighbors (cons new-neighbor neighbors))
            (alert-propagators new-neighbor))))
    (define (add-content increment)
      (cond ((nothing? increment) 'ok)
            ((nothing? content)
             (set! content increment)
             (alert-propagators neighbors))
            (else                              ; **
             (let ((new-range
                    (intersect-intervals content
                                         increment)))
               (cond ((equal? new-range content) 'ok)
                     ((empty-interval? new-range)
                      (error "Ack! Inconsistency!"))
                     (else (set! content new-range)
                           (alert-propagators neighbors)))))))
    (define (me message)
      (cond ((eq? message 'new-neighbor!) new-neighbor!)
            ((eq? message 'add-content) add-content)
            ((eq? message 'content) content)
            (else (error "Unknown message" message))))
    me))
```

The only change we need to make is the else clause of the conditional, marked **. This interesting, interval-specific merge code is what allowed our example network to accept measurements from diverse sources and combine them all to produce a final result that was more precise than the result from any one measurement alone.

4 Multidirectional Computation

So far, the only thing we seem to have gained from putting the merging of partial information into cells is aesthetic cleanliness. The building example above can be simulated perfectly well in an expression language. After all, since the information can only come from two independent sources, it is easy enough to add an explicit interval-intersection step between the outputs of each of the individual methods and the output of both together. This produces a less incremental computation, because it must wait for both means of measurement to produce answers before it commits to outputting its own, but perhaps we can live with that.

The real advantage of letting the cells merge information is that it

lets us build systems with a much broader range of possible information flows. What would happen, for instance, if we augmented our arithmetic to impose a relation, or a constraint [19, 3, 20] if you will, rather than computing a single "output" from the available "inputs"? To do that, we just stack appropriate mutual inverses on top of each other:

```
(define (product x y total)
  (multiplier x y total)
  (divider total x y)
  (divider total y x))

(define (quadratic x x^2)
  (squarer x x^2)
  (sqrter x^2 x))
```

Whichever one has enough inputs will do its computation, and the cells will take care to not get too excited about redundant discoveries.

Our building measurement methods become multidirectional just by composing multidirectional primitives.

```
(define (fall-duration t h)
  (compound-propagator (list t h)
    (lambda ()
      (let ((g (make-cell))
            (one-half (make-cell))
            (t^2 (make-cell))
            (gt^2 (make-cell)))
        ((constant (make-interval 9.789 9.832)) g)
        ((constant (make-interval 1/2 1/2)) one-half)
        (quadratic t t^2)
        (product g t^2 gt^2)
        (product one-half gt^2 h)))))

(define (similar-triangles s-ba h-ba s h)
  (compound-propagator (list s-ba h-ba s h)
    (lambda ()
      (let ((ratio (make-cell)))
        (product s-ba ratio h-ba)
        (product s ratio h)))))
```

Now the estimation of the building's height works just fine,

```
(define barometer-height (make-cell))
(define barometer-shadow (make-cell))
(define building-height (make-cell))
(define building-shadow (make-cell))
(similar-triangles barometer-shadow barometer-height
                   building-shadow building-height)

(add-content building-shadow (make-interval 54.9 55.1))
(add-content barometer-height (make-interval 0.3 0.32))
(add-content barometer-shadow (make-interval 0.36 0.37))
(content building-height)
#(interval 44.514 48.978)
```

as does the refinement of that estimate by adding another measurement.

```
(define fall-time (make-cell))
(fall-duration fall-time building-height)

(add-content fall-time (make-interval 2.9 3.1))
(content building-height)
#(interval 44.514 47.243)
```

But something else interesting happens as well. The better information available about the height of the building propagates backward, and lets us infer refinements of some of our initial measurements!

```
(content barometer-height)
#(interval .3 .31839)

(content fall-time)
#(interval 3.0091 3.1)
```

Indeed, if we offer (yet another) barometer to the building's superintendent in return for perfect information about the building's height, we can use it to refine our understanding of barometers and our experiments even further:

```
(add-content building-height (make-interval 45 45))
(content barometer-height)
#(interval .3 .30328)

(content barometer-shadow)
#(interval .366 .37)

(content building-shadow)
#(interval 54.9 55.1)

(content fall-time)
#(interval 3.0255 3.0322)
```

5 Generic Operations

The changes we needed to move from propagating numbers to propagating intervals were minor. We had to change the cell's behavior on receiving new information, and we had to redefine the "primitive" arithmetic. We can make the two kinds of networks interoperate with each other (and with future kinds we will introduce later) by transforming those procedures into published generic operations, and allowing them to be extended as needed.

First cells. The code we had to vary was what to do when new information is added to the cell. We can factor that out into a generic function, `merge`, whose methods depend upon the kind of partial information being tendered. The remaining commonality is deciding whether to alert the neighbors to the change, for which we leave the cell responsible.

```
(define (make-cell)
  (let ((neighbors '()) (content nothing))
    (define (new-neighbor! new-neighbor)
      (if (not (memq new-neighbor neighbors))
          (begin
            (set! neighbors (cons new-neighbor neighbors))
            (alert-propagators new-neighbor))))
    (define (add-content increment)  ; ***
      (let ((answer (merge content increment)))
        (cond ((eq? answer content) 'ok)
              ((contradictory? answer)
               (error "Ack! Inconsistency!"))
              (else (set! content answer)
                    (alert-propagators neighbors)))))
    (define (me message)
      (cond ((eq? message 'new-neighbor!) new-neighbor!)
            ((eq? message 'add-content) add-content)
            ((eq? message 'content) content)
            (else (error "Unknown message" message))))
    me))
```

The contract of the generic function `merge` is that it takes two arguments: the currently known information and the new information being supplied, and returns the new aggregate information. If the new information is redundant, the `merge` function should return exactly (by `eq?`) the original information, whereas if the new information contradicts the old information, `merge` should return a distinguished value indicating the contradiction. For symmetry and future use, if the new information strictly supercedes the old (i.e. if the old information would be redundant given the new, but the new is not redundant given the old) `merge` is expected to return exactly (by `eq?`) the new information.

The interface to our generic operation facility consists of the two procedures `make-generic-operator` and `assign-operation`. The procedure `make-generic-operator` creates a new generic procedure with the given arity, name, and default operation. The default operation will be called if no methods are applicable.

```
(define merge
  (make-generic-operator 2 'merge
   (lambda (content increment)
     (if (default-equal? content increment)
         content
         the-contradiction))))

(define the-contradiction (list 'contradiction))
(define (contradictory? x) (eq? x the-contradiction))
```

The procedure `assign-operation` takes the name of a generic procedure, a method, and a list of predicates, and extends the named generic procedure to invoke the given method if the supplied arguments are all accepted by the given predicates, in order. This is a predicate dispatch system [8].

```
(assign-operation 'merge
 (lambda (content increment) content)
 any? nothing?)

(assign-operation 'merge
 (lambda (content increment) increment)
 nothing? any?)
```

Together with the default operation on `merge`, these methods replicate the behavior of our first cell.

To be able to use intervals throughout a network, we need only add a method that describes how to combine them with each other,

```
(assign-operation 'merge
 (lambda (content increment)
   (let ((new-range (intersect-intervals content increment)))
     (cond ((interval-equal? new-range content) content)
           ((interval-equal? new-range increment) increment)
           ((empty-interval? new-range) the-contradiction)
           (else new-range))))
 interval? interval?)
```

but interpreting raw numbers as intervals allows us to interoperate with them as well.

```
(define (ensure-inside interval number)
  (if (<= (interval-low interval) number (interval-high interval))
      number
      the-contradiction))

(assign-operation 'merge
 (lambda (content increment)
   (ensure-inside increment content))
 number? interval?)

(assign-operation 'merge
 (lambda (content increment)
   (ensure-inside content increment))
 interval? number?)
```

Second, we can define generic variants of the standard arithmetic operations

```
(define generic-+ (make-generic-operator 2 '+ +))
(define generic-- (make-generic-operator 2 '- -))
(define generic-* (make-generic-operator 2 '* *))
(define generic-/ (make-generic-operator 2 '/ /))
(define generic-square (make-generic-operator 1 'square square))
(define generic-sqrt (make-generic-operator 1 'sqrt sqrt))
;;; ... for more generic primitives see Appendix C.1
```

for our propagators to run

```
(define adder (function->propagator-constructor generic-+))
(define subtractor (function->propagator-constructor generic--))
(define multiplier (function->propagator-constructor generic-*))
(define divider (function->propagator-constructor generic-/))
(define squarer (function->propagator-constructor generic-square))
(define sqrter (function->propagator-constructor generic-sqrt))
;;; ... the conditional propagator is subtle, see Appendix C.5
```

and attach interval arithmetic methods to them[5]

```
(assign-operation '* mul-interval interval? interval?)
(assign-operation '/ div-interval interval? interval?)
(assign-operation 'square square-interval interval?)
(assign-operation 'sqrt sqrt-interval interval?)
;;; ...
```

Now our interval arithmetic still works

[5]Besides methods implementing interval arithmetic, we also need methods that will promote numbers to intervals when needed. The details are in Appendix C.2.

```
(define fall-time (make-cell))
(define building-height (make-cell))
(fall-duration fall-time building-height)

(add-content fall-time (make-interval 2.9 3.1))
(content building-height)
#(interval 41.163 47.243)
```

and interoperates with raw numbers appearing in the network.

```
(add-content building-height 45)

(content fall-time)
#(interval 3.0255 3.0322)
```

6 Dependencies

Now that we have established a general mechanism for computing with various forms of partial information, we are ready to consider a particularly consequential form.

All humans harbor mutually inconsistent beliefs: an intelligent person may be committed to the scientific method yet have a strong attachment to some superstitious or ritual practices. A person may have a strong belief in the sanctity of all human life, yet also believe that capital punishment is sometimes justified. If we were really logicians this kind of inconsistency would be fatal, because were we to simultaneously believe both propositions P and NOT P then we would have to believe all propositions! Somehow we manage to keep inconsistent beliefs from inhibiting all useful thought. Our personal belief systems appear to be locally consistent, in that there are no contradictions apparent. If we observe inconsistencies we do not crash—we chuckle!

Dependency decorations on data that record the justifications for the data give us a powerful tool for organizing computations. Every piece of data (or procedure) came from somewhere. Either it entered the computation as a premise that can be labeled with its external provenance, or it was created by combining other data. We can add methods to our primitive operations which, when processing or combining data that is decorated with justifications, can decorate the results with appropriate justifications. The justifications can be at differing levels of detail. For example, the simplest kind of justification is just a set of those premises that contributed to the new data. A procedure such as addition can decorate a sum with a justification that is just the union of the premises of the justifications of the addends that were supplied. Multiplication can be more complicated: if a multiplicand is zero, that is sufficient to force

the product to be zero, so the justifications of the other operands are not required to be included in the justification of the product. Such simple justifications can be carried without more than a constant factor overhead in space, but they can be invaluable in debugging complex processes and in the attribution of credit or blame for outcomes of computations.

We can choose to keep more than just the set of supports of data by providing an elaboration such that with each datum we keep the way it was derived. This is good for providing explanations but it is intrinsically a bit more expensive. Although such an audit trail may be costlier, it can facilitate the debugging of complex processes.

By decorating data with dependencies a system can manage and usefully compute with multiple, possibly inconsistent world views. A world view is a subset of the data that is supported by a given set of explicit assumptions. Each computational process may restrict itself to working with some consistent world view. A *Truth Maintenance System*, [7, 15, 11], is a mechanism for implementing world views.

If a contradiction is discovered, a process can determine the particular "nogood set" of inconsistent premises. The system can then "chuckle", realizing that no computations supported by any superset of those premises can be believed. This chuckling process, *Dependency-Directed Backtracking*, [18, 14, 21], can be used to optimize a complex search process, allowing a search to make the best use of its mistakes. But enabling a process to simultaneously hold beliefs based on mutually inconsistent sets of premises, without logical disaster, is itself revolutionary.

6.1 Dependencies for Provenance

We now illustrate these ideas, and the manner in which they fit into the propagator network framework, by building a sequence of sample dependency tracking systems of increasing complexity. We start with a relatively simple system that only tracks and reports the provenance of its data.

How do we want our provenance system to work? We can make cells and define networks as usual, but if we add supported values as inputs, we get supported values as outputs:

```
(define barometer-height (make-cell))
(define barometer-shadow (make-cell))
(define building-height (make-cell))
(define building-shadow (make-cell))
(similar-triangles barometer-shadow barometer-height
                   building-shadow building-height)

(add-content building-shadow
 (supported (make-interval 54.9 55.1) '(shadows)))
(add-content barometer-height
 (supported (make-interval 0.3 0.32) '(shadows)))
(add-content barometer-shadow
 (supported (make-interval 0.36 0.37) '(shadows)))
(content building-height)
#(supported #(interval 44.514 48.978) (shadows))
```

Indeed, our estimate for the height of the building depends on our measurements of the barometer and the shadow. We can try dropping the barometer off the roof, but if we do a bad job of timing its fall, our estimate won't improve.

```
(define fall-time (make-cell))
(fall-duration fall-time building-height)

(add-content fall-time
 (supported (make-interval 2.9 3.3) '(lousy-fall-time)))
(content building-height)
#(supported #(interval 44.514 48.978) (shadows))
```

What's more, the dependency tracker tells us that it was a lousy timing job, because the resulting answer doesn't actually depend on the fall timing measurement. If we do it better, then we can get a finer estimate, which then will depend on the improved fall timing measurement.

```
(add-content fall-time
 (supported (make-interval 2.9 3.1) '(better-fall-time)))
(content building-height)
#(supported #(interval 44.514 47.243)
            (better-fall-time shadows))
```

If we then give a barometer to the superintendent, we can watch the superintendent's information supercede and obsolesce the results of our measurements

```
(add-content building-height (supported 45 '(superintendent)))
(content building-height)
#(supported 45 (superintendent))
```

and see which of the measurements themselves we can infer more about based on the now known height of the building.

```
(content barometer-height)
#(supported #(interval .3 .30328)
            (superintendent better-fall-time shadows))

(content barometer-shadow)
#(supported #(interval .366 .37)
            (better-fall-time superintendent shadows))

(content building-shadow)
#(supported #(interval 54.9 55.1) (shadows))

(content fall-time)
#(supported #(interval 3.0255 3.0322)
            (shadows superintendent))
```

Here values in cells can depend upon sets of premises. In this example each cell is allowed to hold one value, and the `merge` procedure combines the possible values for a cell into one most-informative value that can be derived from the values tendered. This value is supported by the union of the supports of those values that contributed to the most informative value. The value is accumulated by combining the current value (and justifications) with the new value being proposed, one at at time.

This particular approach to accumulating supports is imperfect, and can produce spurious dependencies. We can illustrate this with interval arithmetic by imagining three values A, B, C being proposed in order. A possible computation is shown in Figure 2. We see that premise A

```
A:     [           ]
B:           [         ]
-----------------------
A,B:         [     ]
C:         [    ]
-----------------------
A,B,C:       [  ]
```

Figure 2: The overlap anomaly

is included in the justifications for the result, even though it is actually irrelevant, because it is superseded by the value supported by C. This case actually occurs in the code example above. We will ameliorate this anomaly this later.

Making this work

What do we need to do to make this dependency tracking work? We define a data structure we name a `v&s` to store a value together with the dependecies that support it, see Appendix B.

```
(define (v&s-merge v&s1 v&s2)
  (let* ((v&s1-value (v&s-value v&s1))
         (v&s2-value (v&s-value v&s2))
         (value-merge (merge v&s1-value v&s2-value)))
    (cond ((eq? value-merge v&s1-value)
           (if (implies? v&s2-value value-merge)
               ;; Confirmation of existing information
               (if (more-informative-support? v&s2 v&s1)
                   v&s2
                   v&s1)
               ;; New information is not interesting
               v&s1))
          ((eq? value-merge v&s2-value)
           ;; New information overrides old information
           v&s2)
          (else
           ;; Interesting merge, need both provenances
           (supported value-merge
                      (merge-supports v&s1 v&s2))))))

(assign-operation 'merge v&s-merge v&s? v&s?)

(define (implies? v1 v2)
  (eq? v1 (merge v1 v2)))
```

Figure 3: Merging supported values

The important thing is to describe how to merge the information contained in two such data structures; see Figure 3. The value contained in the answer must of course be the merge of the values contained in the two inputs, but sometimes we may get away with using only some of the supporting premises. There are three cases: if neither the new nor the old values are redundant, then we need both their supports; if either is strictly redundant, we needn't include its support; and if they are equivalent, we can choose which support to use. In this case, we use the support of the value already present unless the support of the new one is strictly more informative (i.e. is a strict subset of the same premises).

If it so happens that two supported values contradict each other, we want to return an object that will be recognized as representing a contradiction, but will retain the information about which premises were involved in the contradiction. It is convenient to do that by turning the

cell's contradiction test into a generic operation; that way we can let `v&s-merge` return a supported value whose value is a contradiction, and give ourselves the ability to extend the notion of contradiction later.

```
(define contradictory?
  (make-generic-operator 1 'contradictory?
   (lambda (thing) (eq? thing the-contradiction))))

(assign-operation 'contradictory?
 (lambda (v&s) (contradictory? (v&s-value v&s)))
 v&s?)
```

Finally, we need to upgrade our arithmetic primitives to carry dependencies around, and to interoperate with data they find that lacks justifications by inserting empty dependency sets; see Appendix C.3.

6.2 Dependencies for Alternate Worldviews

The anomaly shown above was a consequence of the loss of information about the derivation of a value due to its sequential accumulation. We can begin to generalize this by allowing a cell to hold more than one value at a time, each justified by its own justification. When queried, a cell can do whatever deduction is required to give the most informative answer it can, justified by the weakest sufficient set of premises. Allowing multiple values provides further advantages. It becomes possible to efficiently support multiple alternate worldviews: a query to a cell may be restricted to return only values that are supported by a subset of the set of possible premises. Such a subset is called a *worldview*. A truth maintenance system (TMS) can be used to store multiple values with different justifications for this purpose. If we put TMSes in our cells, we can revisit the building-height problem:

```
(define barometer-height (make-cell))
(define barometer-shadow (make-cell))
(define building-height (make-cell))
(define building-shadow (make-cell))
(similar-triangles barometer-shadow barometer-height
                   building-shadow building-height)

(add-content building-shadow
 (make-tms (supported (make-interval 54.9 55.1) '(shadows))))
(add-content barometer-height
 (make-tms (supported (make-interval 0.3 0.32) '(shadows))))
(add-content barometer-shadow
 (make-tms (supported (make-interval 0.36 0.37) '(shadows))))
(content building-height)
#(tms (#(supported #(interval 44.514 48.978) (shadows))))
```

Nothing much changes while there is only one source of information,

```
(define fall-time (make-cell))
(fall-duration fall-time building-height)

(add-content fall-time
 (make-tms (supported (make-interval 2.9 3.1) '(fall-time))))
(content building-height)
#(tms (#(supported #(interval 44.514 47.243)
                   (shadows fall-time))
       #(supported #(interval 44.514 48.978)
                   (shadows))))
```

but when we add the second experiment, the TMS remembers the deductions made in the first. In this particular system, we chose to make the worldview implicit and global, and to have it include all premises by default. That works fine on a uniprocessor, but a more distributed propagator network might be better served by a more local notion of worldview, and by propagating changes thereto explicitly rather than letting them instantly affect the entire network.

With an implicit global worldview, querying a TMS requires no additional input and produces the most informative value supported by premises in the current worldview:

```
(tms-query (content building-height))
#(supported #(interval 44.514 47.243) (shadows fall-time))
```

We also provide a means to remove premises from the current worldview, which in this case causes the TMS query to fall back to the less-informative inference that can be made using only the `shadows` experiment:

```
(kick-out! 'fall-time)
(tms-query (content building-height))
#(supported #(interval 44.514 48.978) (shadows))
```

Likewise, we can ask the system for the best answer it can give if we trust the `fall-time` experiment but not the `shadows` experiment,

```
(bring-in! 'fall-time)
(kick-out! 'shadows)
(tms-query (content building-height))
#(supported #(interval 41.163 47.243) (fall-time))
```

which may involve some additional computation not needed heretofore, whose results are reflected in the full TMS we can observe at the end.

```
(content building-height)
#(tms (#(supported #(interval 41.163 47.243)
                   (fall-time))
       #(supported #(interval 44.514 47.243)
                   (shadows fall-time))
       #(supported #(interval 44.514 48.978)
                   (shadows))))
```

Now, if we give the superintendent a barometer, we can add her input to the totality of our knowledge about this building

```
(add-content building-height (supported 45 '(superintendent)))
```

and observe that it is stored faithfully along with all the rest,

```
(content building-height)
#(tms (#(supported 45 (superintendent))
       #(supported #(interval 41.163 47.243)
                   (fall-time))
       #(supported #(interval 44.514 47.243)
                   (shadows fall-time))
       #(supported #(interval 44.514 48.978)
                   (shadows))))
```

though indeed if we trust it, it provides the best estimate we have.

```
(tms-query (content building-height))
#(supported 45 (superintendent))
```

(and restoring our faith in the `shadows` experiment has no effect on the accuracy of this answer).

```
(bring-in! 'shadows)
(tms-query (content building-height))
#(supported 45 (superintendent))
```

If we now turn our attention to the height of the barometers we have been dropping and giving away, we notice that as before, in addition to the originally supplied measurements, the system has made a variety of deductions about it, based on reasoning backwards from our estimates of the height of the building and the other measurements in the shadows experiment.

```
(content barometer-height)
#(tms (#(supported #(interval .3 .30328)
                   (fall-time superintendent shadows))
       #(supported #(interval .29401 .30328)
                   (superintendent shadows))
       #(supported #(interval .3 .31839)
                   (fall-time shadows))
       #(supported #(interval .3 .32) (shadows))))
```

If we should ask for the best estimate of the height of the barometer, we observe the same problem we noticed in the previous section, namely that the system produces a spurious dependency on the `fall-time` experiment, whose findings are actually redundant for answering this question.

```
(tms-query (content barometer-height))
#(supported #(interval .3 .30328)
            (fall-time superintendent shadows))
```

We can verify the irrelevance of the `fall-time` measurements by disbelieving them and observing that the answer remains the same, but with more accurate dependencies.

```
(kick-out! 'fall-time)
(tms-query (content barometer-height))
#(supported #(interval .3 .30328) (superintendent shadows))
```

What is more, having been asked to make that deduction, the truth maintenance system remembers it, and produces the better answer thereafter, even if we subsequently restore our faith in the `fall-time` experiment,

```
(bring-in! 'fall-time)
(tms-query (content barometer-height))
#(supported #(interval .3 .30328) (superintendent shadows))
```

and takes the opportunity to dispose of prior deductions that are obsoleted by this new realization.

```
(content barometer-height)
#(tms (#(supported #(interval .3 .30328)
                   (superintendent shadows))
       #(supported #(interval .3 .31839)
                   (fall-time shadows))
       #(supported #(interval .3 .32) (shadows))))
```

We have been fortunate so far in having all our measurements agree with each other, so that all worldviews we could possibly hold were, in fact, internally consistent. But what if we had mutually contradictory data? What if, for instance, we happened to observe the barometer's readings, both while measuring its shadow on the ground and before dropping it off the roof of the building? We might then consult some pressure-altitude tables, or possibly even an appropriate formula relating altitude and pressure, and, being blessed with operating on a calm, windless day, deduce, by computations whose actual implementation as a propagator network would consume more space than it would produce enlightenment, yet another interval within which the building's height must necessarily lie. Should this new information contradict our previous store of knowledge, we would like to know; and since the system maintains dependency information, it can even tell us which premises lead to trouble.

```
(add-content building-height
  (supported (make-interval 46. 50.) '(pressure)))
(contradiction (superintendent pressure))
```

Indeed, if we ask after the height of the building under this regime of contradictory information, we will be informed of the absence of a good answer,

```
(tms-query (content building-height))
#(supported (contradiction) (superintendent pressure))
```

but it is appropriate for the system not to propagate consequences deducible in an inconsistent worldview.

```
(tms-query (content barometer-height))
#(supported #(interval .3 .30328) (superintendent shadows))
```

It is up to us as the users of the system to choose which worldview to explore. We can ascertain the consequences of disregarding the superintendent's assertions, both on our understanding of the height of the building

```
(kick-out! 'superintendent)
(tms-query (content building-height))
#(supported #(interval 46. 47.243) (fall-time pressure))
```

and on that of the barometer.

```
(tms-query (content barometer-height))
#(supported #(interval .30054 .31839)
            (pressure fall-time shadows))
```

Doing so does not cost us previously learned data, so we are free to change worldviews at will, reasoning as we like in one consistent worldview or another.

```
(bring-in! 'superintendent)
(kick-out! 'pressure)
(tms-query (content building-height))
#(supported 45 (superintendent))
(tms-query (content barometer-height))
#(supported #(interval .3 .30328) (superintendent shadows))
```

Making this work

The first component of making this work is to define a suitable TMS to put into cells. In Appendix B we define a `tms` record structure that contains a list of `v&s` records.

A TMS is a set of `v&s`s. These `v&s`s represent the direct deductions the surrounding system has added to the TMS, and any consequences thereof the TMS has deduced on its own. Asking the TMS to deduce all the consequences of all its facts all the time is perhaps a bad idea, so when we merge TMSes we assimilate the facts from the incoming one into the current one, and then only deduce those consequences that are relevant to the current worldview. If this deduction reveals a contradiction, we need to signal that to prevent the system from pursuing further computations in a worldview that is known to be inconsistent.

```
(define (tms-merge tms1 tms2)
  (let ((candidate (tms-assimilate tms1 tms2)))
    (let ((consequence (strongest-consequence candidate)))
      (check-consistent! consequence)
      (tms-assimilate candidate consequence))))

(assign-operation 'merge tms-merge tms? tms?)
```

The procedure `tms-assimilate` incorporates all the given items, one by one, into the given TMS with no deduction of consequences.

```
(define (tms-assimilate tms stuff)
  (cond ((nothing? stuff) tms)
        ((v&s? stuff) (tms-assimilate-one tms stuff))
        ((tms? stuff)
         (fold-left tms-assimilate-one
                    tms
                    (tms-values stuff)))
        (else (error "This should never happen"))))
```

When we add a new `v&s` to an existing TMS we check whether the information contained in the new `v&s` is deducible from that in one already in the TMS. If so, we can just throw the new one away. Conversely, if the information in any existing `v&s` is deducible from the information in the new one, we can throw those existing ones away. The predicate `subsumes?` returns true only if the information contained in the second argument is deducible from that contained in the first.

```
(define (subsumes? v&s1 v&s2)
  (and (implies? (v&s-value v&s1) (v&s-value v&s2))
       (lset<= eq? (v&s-support v&s1) (v&s-support v&s2))))

(define (tms-assimilate-one tms v&s)
  (if (any (lambda (old-v&s) (subsumes? old-v&s v&s))
           (tms-values tms))
      tms
      (let ((subsumed
             (filter (lambda (old-v&s) (subsumes? v&s old-v&s))
                     (tms-values tms))))
        (make-tms
         (lset-adjoin eq?
           (lset-difference eq? (tms-values tms) subsumed)
           v&s)))))
```

The procedure `strongest-consequence` finds the most informative consequence of the current worldview. It does this by using `merge` to combine all of the currently believed facts in the tms.

```
(define (strongest-consequence tms)
  (let ((relevant-v&ss
         (filter all-premises-in? (tms-values tms))))
    (fold-left merge nothing relevant-v&ss)))

(define (all-premises-in? thing)
  (if (v&s? thing)
      (all-premises-in? (v&s-support thing))
      (every premise-in? thing)))
```

Finally, `check-consistent!` looks for a contradictory conclusion and

extracts the *nogood set* of premises that support it for later processing.

```
(define (check-consistent! v&s)
  (if (contradictory? v&s)
      (process-nogood! (v&s-support v&s))))
```

To interpret a given TMS in the current worldview is not quite as simple as just calling `strongest-consequence`, because if the consequence has not been deduced previously, which can happen if the worldview changed after the last time `tms-merge` was called, the consequence should be fed back into the TMS and checked for consistency.

```
(define (tms-query tms)
  (let ((answer (strongest-consequence tms)))
    (let ((better-tms (tms-assimilate tms answer)))
      (if (not (eq? tms better-tms))
          (set-tms-values! tms (tms-values better-tms)))
      (check-consistent! answer)
      answer)))
```

To support the implicit global worldview, we need a mechanism to distinguish premises that are believed in the current worldview from premises that are not. Premises may be marked with "sticky notes"; Appendix B shows how this is arranged.

Manually changing these sticky notes violates the network's monotonicity assumptions, so all propagators whose inputs might change under them need to be alerted when this happens. Altering all the propagators indiscriminately is a conservative approximation that works reasonably for a single process simulation.

```
(define (kick-out! premise)
  (if (premise-in? premise) (alert-all-propagators!))
  (mark-premise-out! premise))
(define (bring-in! premise)
  (if (not (premise-in? premise)) (alert-all-propagators!))
  (mark-premise-in! premise))
```

In order to let cells containing TMSes interoperate with cells containing other kinds of partial information that can be viewed as TMSes, we sprinkle on some boring, repetitive coercions and we augment our generic arithmetic operations to unpack TMSes and coerce things to ensure the same level of unpacking. The interested reader can find both in Appendix C.4.

The `process-nogood!` procedure just aborts the process, giving the user a chance to adjust the worldview to avoid the contradiction.

```
(define (process-nogood! nogood)
  (abort-process `(contradiction ,nogood)))
```

This will be expanded next.

6.3 Dependencies for Implicit Search

Implicit generate and test can be viewed as a way of making systems that are modular and independently evolvable. Consider a very simple example: suppose we have to solve a quadratic equation. There are two roots to a quadratic. We could return both, and assume that the user of the solution knows how to deal with that, or we could return one and hope for the best. (The canonical `sqrt` routine returns the positive square root, even though there are two square roots!) The disadvantage of returning both solutions is that the receiver of that result must know to try his computation with both and either reject one, for good reason, or return both results of his computation, which may itself have made some choices. The disadvantage of returning only one solution is that it may not be the right one for the receiver's purpose.

A better way to handle this is to build a backtracking mechanism into the infrastructure.[10, 13, 16, 2, 17] The square-root procedure should return one of the roots, with the option to change its mind and return the other one if the first choice is determined to be inappropriate by the receiver. It is, and should be, the receiver's responsibility to determine if the ingredients to its computation are appropriate and acceptable. This may itself require a complex computation, involving choices whose consequences may not be apparent without further computation, so the process is recursive. Of course, this gets us into potentially deadly exponential searches through all possible assignments to all the choices that have been made in the program. As usual, modular flexibility can be dangerous.

We can reap the benefit of this modular flexibility by burying search into the already-implicit control flow of the propagator network. Tracing dependencies helps even more because it enables a smarter search. When the network is exploring a search space and reaches a dead end, it can determine which of the choices it made actually contributed to the dead end, so that it can reverse one of those decisions, instead of an irrelevant one, and so that it can learn from the mistake and avoid repeating it in the future.

We illustrate this idea with some information about the building's occupants we have gleaned, with due apologies to Dinesman [6], from the chatty superintendent while we were discussing barometric transactions:

> Baker, Cooper, Fletcher, Miller, and Smith live on the first five floors of this apartment house. Baker does not live on the fifth floor. Cooper does not live on the first floor. Fletcher does not live on either the fifth or the first floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's.

Should we wish to determine from this where everyone lives, we are faced with a search problem. This search problem has the interesting character that the constraints on the search space are fairly local; the failure of any particular configuration is attributable to the violation of a unary or binary constraint. This means that even though the entire space has size $5^5 = 3125$, studying the cause of any one failure can let one eliminate 5^4- or 5^3-sized chunks of the space at each dead end. Using a system that tracks these dependencies automatically can relieve us of having to embed that knowledge into the explicit structure of the search program we write.

The extension we need make to the system we already have is to add a propagator that makes guesses and manufactures new premises to support them, and modify the contradiction detection machinery to inform the guessers of their mistakes and give them the opportunity to change their minds.

To make the example concrete, Figure 4 shows a direct encoding of the problem statement above as a propagator network.[6]

Observe the generators `one-of`, which guess floors where people live but reserve the right to change their minds later, and testers `require` and `abhor` that point out situations that necessitate changing of minds.

We can run this in our augmented system to find the right answer,

---

[6]This ugly program is written in the moral equivalent of assembly language. Even a relatively straightforward expression-oriented frontend could let us write something far more pleasant:

```
(define (multiple-dwelling)
  (let ((baker (one-of 1 2 3 4 5))     (cooper (one-of 1 2 3 4 5))
        (fletcher (one-of 1 2 3 4 5))  (miller (one-of 1 2 3 4 5))
        (smith (one-of 1 2 3 4 5)))
    (require-distinct
     (list baker cooper fletcher miller smith))
    (abhor (= baker 5))     (abhor (= cooper 1))
    (abhor (= fletcher 5))  (abhor (= fletcher 1))
    (require (> miller cooper))
    (abhor (= 1 (abs (- smith fletcher))))
    (abhor (= 1 (abs (- fletcher cooper))))
    (list baker cooper fletcher miller smith)))
```

```
(define (multiple-dwelling)
  (let ((baker    (make-cell))  (cooper (make-cell))
        (fletcher (make-cell))  (miller (make-cell))
        (smith    (make-cell))  (floors '(1 2 3 4 5)))
    (one-of floors baker)       (one-of floors cooper)
    (one-of floors fletcher)    (one-of floors miller)
    (one-of floors smith)
    (require-distinct
     (list baker cooper fletcher miller smith))
    (let ((b=5  (make-cell))    (c=1 (make-cell))
          (f=5  (make-cell))    (f=1 (make-cell))
          (m>c  (make-cell))    (sf  (make-cell))
          (fc   (make-cell))    (one (make-cell))
          (five (make-cell))    (s-f (make-cell))
          (as-f (make-cell))    (f-c (make-cell))
          (af-c (make-cell)))
      ((constant 1) one)        ((constant 5) five)
      (=? five baker b=5)       (abhor b=5)
      (=? one cooper c=1)       (abhor c=1)
      (=? five fletcher f=5)    (abhor f=5)
      (=? one fletcher f=1)     (abhor f=1)
      (>? miller cooper m>c)    (require m>c)
      (subtractor smith fletcher s-f)
      (absolute-value s-f as-f)
      (=? one as-f sf)          (abhor sf)
      (subtractor fletcher cooper f-c)
      (absolute-value f-c af-c)
      (=? one af-c fc)          (abhor fc)
      (list baker cooper fletcher miller smith))))
```

Figure 4: The superintendent's puzzle

```
(define answers (multiple-dwelling))
(map v&s-value (map tms-query (map content answers)))
(3 2 4 5 1)
```

and observe how few dead ends the network needed to consider before finding it.

```
*number-of-calls-to-fail*
63
```

In contrast, a naive depth-first search would examine 582 configurations before finding the answer. Although clever reformulations of the program that defines the problem can reduce the search substantially, they defeat much of the purpose of making the search implicit.

Making this work

We start by adding a mechanism for making guesses in Figure 5. This

```
(define (binary-amb cell)
  (let ((true-premise (make-hypothetical))
        (false-premise (make-hypothetical)))
    (define (amb-choose)
      (let ((reasons-against-true
             (filter all-premises-in?
               (premise-nogoods true-premise)))
            (reasons-against-false
             (filter all-premises-in?
               (premise-nogoods false-premise))))
        (cond ((null? reasons-against-true)
               (kick-out! false-premise)
               (bring-in! true-premise))
              ((null? reasons-against-false)
               (kick-out! true-premise)
               (bring-in! false-premise))
              (else                  ; this amb must fail.
               (kick-out! true-premise)
               (kick-out! false-premise)
               (process-contradictions
                (pairwise-union reasons-against-true
                                reasons-against-false))))))
    ((constant (make-tms
                (list (supported #t (list true-premise))
                      (supported #f (list false-premise)))))
     cell)
    ;; The cell is a spiritual neighbor...
    (propagator cell amb-choose)))
```

Figure 5: A guessing machine

works by manufacturing two new premises, and adding to its cell the guess `#t` supported by one premise and the guess `#f` supported by the other. It also creates a propagator that will ensure that any stable worldview always believes exactly one of these premises. This propagator is awakened every time the worldview changes.

Premises accumulate reasons why they should not be believed (data structure details in Appendix B). Such a reason is a set of premises which forms a nogood if this premise is added to it. If all the premises in any such set are currently believed, that consitutes a valid reason not to believe this premise. If neither of a guesser's premises can be believed, the guesser can perform a resolution step to deduce a nogood that does not involve either of its premises.

The `pairwise-union` procedure is a utility that takes two lists of sets (of premises) and produces a list of all the unions (eliminating `eq?`-duplicates) of all pairs of sets from the two input lists. This constitutes a resolution of the nogoods represented by the `reasons-against-true` with those represented by the `reasons-against-false`.

When multiple contradictions are discovered at once, we choose to act upon just one of them, on the logic that the others will be rediscovered if they are significant. We choose one with the fewest hypothetical premises because it produces the greatest constraint on the search space.

```
(define (process-contradictions nogoods)
  (process-one-contradiction
   (car (sort-by nogoods
          (lambda (nogood)
            (length (filter hypothetical? nogood)))))))
```

If a contradiction contains no hypotheticals, there is nothing more to be done automatically; we abort the process, giving the user a chance to adjust the worldview manually. If there are hypotheticals, however, we avoid the contradiction by arbitrarily disbelieving the first hypothetical that participates in it. We also tell all the participating premises about the new nogood so that it can be avoided in the future.

```
(define (process-one-contradiction nogood)
  (let ((hyps (filter hypothetical? nogood)))
    (if (null? hyps)
        (abort-process `(contradiction ,nogood))
        (begin
          (kick-out! (car hyps))
          (for-each (lambda (premise)
                      (assimilate-nogood! premise nogood))
                    nogood)))))
```

Teaching a premise about a nogood has two bits. The first is to remove the premise from the nogood to create the premise-nogood we need to store. The second is to add it to the list of premise-nogoods already associated with this premise, taking care to eliminate any subsumed premise-nogoods (supersets of other premise-nogoods).

```
(define (assimilate-nogood! premise new-nogood)
  (let ((item (delq premise new-nogood))
        (set (premise-nogoods premise)))
    (if (any (lambda (old) (lset<= eq? old item)) set)
        #f
        (let ((subsumed
               (filter (lambda (old) (lset<= eq? item old))
                       set)))
          (set-premise-nogoods! premise
            (lset-adjoin eq?
              (lset-difference eq? set subsumed) item))))))
```

Finally, we have the machinery to let a contradiction discovered in the network (by `check-consistent!`, see page 32) trigger an automatic change in the worldview.

```
(define (process-nogood! nogood)
  (set! *number-of-calls-to-fail*
        (+ *number-of-calls-to-fail* 1))
  (process-one-contradiction nogood))
```

Just for fun, we count the number of times the network hits a contradiction.

The emergent behavior of a bunch of `binary-amb` propagators embedded in a network is that of a distributed incremental implicit-SAT solver based on propositional resolution. This particular SAT solver is deliberately as simple as we could make it; a "production" system could incorporate modern SAT-solving techniques (e.g. [1]) from the abundant literature. The network naturally integrates SAT-solving with discovering additional clauses by arbitrary other computation. From the point of view of the SAT solver, the SAT problem is implicit (in the computation done by the network). From the point of view of the computation, the search done by the SAT solver is implicit.

Ambiguity Utilities

A few "assembly macros" are still needed to make our example program even remotely readable. Since the cells already detect contradictions, `require` and `abhor` turn out to be very elegant:

```
(define (require cell)
  ((constant #t) cell))

(define (abhor cell)
  ((constant #f) cell))
```

The `require-distinct` procedure just `abhor`s the equality of any two of the supplied cells.

```
(define (require-distinct cells)
  (for-each-distinct-pair
   (lambda (c1 c2)
     (let ((p (make-cell))) (=? c1 c2 p) (abhor p)))
   cells))
```

Upgrading a binary choice to an n-ary choice can be a simple matter of constructing a linear chain of binary choices controlling conditionals.

```
(define (one-of values output-cell)
  (let ((cells
         (map (lambda (value)
                (let ((cell (make-cell)))
                  ((constant value) cell)
                  cell))
              values)))
    (one-of-the-cells cells output-cell)))

(define (one-of-the-cells input-cells output-cell)
  (cond ((= (length input-cells) 2)
         (let ((p (make-cell)))
           (conditional p
             (car input-cells) (cadr input-cells)
             output-cell)
           (binary-amb p)))
        ((> (length input-cells) 2)
         (let ((link (make-cell)) (p (make-cell)))
           (one-of-the-cells (cdr input-cells) link)
           (conditional
            p (car input-cells) link output-cell)
           (binary-amb p)))
        (else
         (error "Inadequate choices for one-of-the-cells"
                input-cells output-cell))))
```

7 There is More to Do

The networks described so far resemble applicative order lambda calculus, in that all the propagators compute all the deductions they can as soon as possible (except for limitations due to the worldview introduced in Section 6.2): the propagators "push" data through the network. It is natural to ask about the meaning of normal order in this context: can propagators "pull" only the data they need? A novel problem arises: in contrast with an expression-oriented language, information can enter a

cell from multiple sources, and there is no clear way to know in advance which source(s) are right to pull from. Even worse, since sources are free to produce partial information and refine it later, there is no clear way to know when to stop pulling (the only non-refinable information is a contradiction, which hopefully one does not see often). On the bright side, a novel opportunity arises as well: since the partialness of information is acknowledged by the architecture, one can choose to pull on only some portion of the information space. For example, one might be interested only in answers that don't require supposing some particular premise; in this case, computations of values whose supports do contain that premise can be avoided. Perhaps one can represent "pulls" as explicit requests, indicating interest in information, and perhaps restrictions on the information of interest, which can also be propagated as their own kind of information.

The networks in our examples propagate partial information about only "atomic" values — numbers and booleans. The dependency tracking will work as written[7] over compound objects like pairs and vectors, but it can be far better to maintain separate justifications for the first and last component of a given pair (and for believing that it *is* a pair), because those individual justifications may be much smaller than the complete justification of the whole. In fact, maintaining separate justifications for the upper and lower bounds of an interval would have eliminated the anomaly of Figure 2.

We come, at the end, to the eternal problem of time. The structure of a propagator network is space. To the extent that there is no "spooky" action at a distance, there is no reason to require time to pass uniformly and synchronously in all regions of the network. A concurrent implementation of a propagator network could make this observation a reality; the only things that would need to be synchronized are the individual cells, and the neighbor sets of individual propagators. This observation also lends insight into why the implicit global worldview of Sections 6.2 and 6.3 entailed such kludgery: changes of the worldview "travel" instantly throughout the entire network without any respect for the network's structure.

[7] Well, with some more methods on some generic procedures

A Primitives for Section 2

```
(define adder (function->propagator-constructor +))
(define subtractor (function->propagator-constructor -))
(define multiplier (function->propagator-constructor *))
(define divider (function->propagator-constructor /))
(define absolute-value (function->propagator-constructor abs))
(define squarer (function->propagator-constructor square))
(define sqrter (function->propagator-constructor sqrt))
(define =? (function->propagator-constructor =))
(define <? (function->propagator-constructor <))
(define >? (function->propagator-constructor >))
(define <=? (function->propagator-constructor <=))
(define >=? (function->propagator-constructor >=))
(define inverter (function->propagator-constructor not))
(define conjoiner (function->propagator-constructor boolean/and))
(define disjoiner (function->propagator-constructor boolean/or))
```

B Data Structure Definitions

Our data structures are simple tagged vectors. Mercifully, we do not use vectors elsewhere in the system, so no confusion is possible. Also mercifully, the `define-structure` macro from MIT Scheme [9] automates the construction and abstraction of such data structures.

Intervals:

```
(define-structure
 (interval
  (type vector) (named 'interval) (print-procedure #f))
 low high)

(define interval-equal? equal?)
```

Supported values:

```
(define-structure
 (v&s (named 'supported) (type vector)
      (constructor supported) (print-procedure #f))
 value support)

(define (more-informative-support? v&s1 v&s2)
  (and (not (lset= eq? (v&s-support v&s1) (v&s-support v&s2)))
       (lset<= eq? (v&s-support v&s1) (v&s-support v&s2))))

(define (merge-supports . v&ss)
  (apply lset-union eq? (map v&s-support v&ss)))
```

Lists of supported values for truth maintenance:

```
(define-structure
  (tms (type vector) (named 'tms)
       (constructor %make-tms) (print-procedure #f))
  values)

(define (make-tms arg)
  (%make-tms (listify arg)))
```

Synthetic premises for hypothetical reasoning

```
(define-structure
  (hypothetical (type vector) (named 'hypothetical) (print-procedure #f)))
```

turn out to need only their identity (and the `hypothetical?` type testing predicate).

To support the implicit global worldview, we put sticky notes on our premises indicating whether or not we believe them. A "sticky note" is the presence or absence of the premise in question in a global `eq`-hash table. This mechanism is nice because we can use arbitrary objects as premises, without having to make any arrangements for this in advance. Since the table holds its keys weakly, this does not interfere with garbage collection.

```
(define *premise-outness* (make-eq-hash-table))

(define (premise-in? premise)
  (not (hash-table/get *premise-outness* premise #f)))

(define (mark-premise-in! premise)
  (hash-table/remove! *premise-outness* premise))

(define (mark-premise-out! premise)
  (hash-table/put! *premise-outness* premise #t))
```

We attach premise-nogoods to premises the same way as their membership in the global worldview.

```
(define *premise-nogoods* (make-eq-hash-table))

(define (premise-nogoods premise)
  (hash-table/get *premise-nogoods* premise '()))

(define (set-premise-nogoods! premise nogoods)
  (hash-table/put! *premise-nogoods* premise nogoods))

(define *number-of-calls-to-fail* 0)
```

C Generic Primitives

C.1 Definitions

```
(define generic-+   (make-generic-operator 2 '+   +))
(define generic--   (make-generic-operator 2 '-   -))
(define generic-*   (make-generic-operator 2 '*   *))
(define generic-/   (make-generic-operator 2 '/   /))
(define generic-abs (make-generic-operator 1 'abs abs))
(define generic-square (make-generic-operator 1 'square square))
(define generic-sqrt (make-generic-operator 1 'sqrt sqrt))
(define generic-=   (make-generic-operator 2 '=   =))
(define generic-<   (make-generic-operator 2 '<   <))
(define generic->   (make-generic-operator 2 '>   >))
(define generic-<=  (make-generic-operator 2 '<=  <=))
(define generic->=  (make-generic-operator 2 '>=  >=))
(define generic-not (make-generic-operator 1 'not not))
(define generic-and (make-generic-operator 2 'and boolean/and))
(define generic-or  (make-generic-operator 2 'or  boolean/or))

(define adder (function->propagator-constructor generic-+))
(define subtractor (function->propagator-constructor generic--))
(define multiplier (function->propagator-constructor generic-*))
(define divider (function->propagator-constructor generic-/))
(define absolute-value (function->propagator-constructor generic-abs))
(define squarer (function->propagator-constructor generic-square))
(define sqrter (function->propagator-constructor generic-sqrt))
(define =? (function->propagator-constructor generic-=))
(define <? (function->propagator-constructor generic-<))
(define >? (function->propagator-constructor generic->))
(define <=? (function->propagator-constructor generic-<=))
(define >=? (function->propagator-constructor generic->=))
(define inverter (function->propagator-constructor generic-not))
(define conjoiner (function->propagator-constructor generic-and))
(define disjoiner (function->propagator-constructor generic-or))
```

C.2 Intervals

```
(define (->interval x)
  (if (interval? x)
      x
      (make-interval x x)))

(define (coercing coercer f)
  (lambda args
    (apply f (map coercer args))))

(assign-operation '* mul-interval interval? interval?)
(assign-operation '* (coercing ->interval mul-interval) number? interval?)
(assign-operation '* (coercing ->interval mul-interval) interval? number?)
(assign-operation '/ div-interval interval? interval?)
(assign-operation '/ (coercing ->interval div-interval) number? interval?)
(assign-operation '/ (coercing ->interval div-interval) interval? number?)
(assign-operation 'square square-interval interval?)
(assign-operation 'sqrt sqrt-interval interval?)
```

C.3 Supported Values

```
(define (flat? thing)
  (or (interval? thing)
      (number? thing)
      (boolean? thing)))

(define (v&s-unpacking f)
  (lambda args
    (supported
     (apply f (map v&s-value args))
     (apply merge-supports args))))

(define (->v&s thing)
  (if (v&s? thing)
      thing
      (supported thing '())))

(for-each
 (lambda (name underlying-operation)
   (assign-operation
    name (v&s-unpacking underlying-operation) v&s? v&s?)
   (assign-operation
    name (coercing ->v&s underlying-operation) v&s? flat?)
   (assign-operation
    name (coercing ->v&s underlying-operation) flat? v&s?))
 '(+ - * / = < > <= >= and or)
 (list generic-+ generic-- generic-* generic-/
       generic-= generic-< generic-> generic-<= generic->=
       generic-and generic-or))

(for-each
 (lambda (name underlying-operation)
   (assign-operation
    name (v&s-unpacking underlying-operation) v&s?))
 '(abs square sqrt not)
 (list generic-abs generic-square generic-sqrt generic-not))

(assign-operation 'merge (coercing ->v&s v&s-merge) v&s? flat?)
(assign-operation 'merge (coercing ->v&s v&s-merge) flat? v&s?)
```

Observe how uniform this is if we are willing to forgo the refinement that `(* 0 x)` is `0` irrespective of the dependencies of `x`.

C.4 Truth Maintenance Systems

```
(define (tms-unpacking f)
  (lambda args
    (let ((relevant-information (map tms-query args)))
      (if (any nothing? relevant-information)
          nothing
          (make-tms (list (apply f relevant-information)))))))

(define (full-tms-unpacking f)
  (tms-unpacking (v&s-unpacking f)))

(define (->tms thing)
  (if (tms? thing)
      thing
      (make-tms (list (->v&s thing)))))

(for-each
 (lambda (name underlying-operation)
   (assign-operation
    name (full-tms-unpacking underlying-operation) tms? tms?)
   (assign-operation
    name (coercing ->tms underlying-operation) tms? v&s?)
   (assign-operation
    name (coercing ->tms underlying-operation) v&s? tms?)
   (assign-operation
    name (coercing ->tms underlying-operation) tms? flat?)
   (assign-operation
    name (coercing ->tms underlying-operation) flat? tms?))
 '(+ - * / = < > <= >= and or)
 (list generic-+ generic-- generic-* generic-/
       generic-= generic-< generic-> generic-<= generic->=
       generic-and generic-or))

(for-each
 (lambda (name underlying-operation)
   (assign-operation
    name (full-tms-unpacking underlying-operation) tms?))
 '(abs square sqrt not)
 (list generic-abs generic-square generic-sqrt generic-not))

(assign-operation 'merge (coercing ->tms tms-merge) tms? v&s?)
(assign-operation 'merge (coercing ->tms tms-merge) v&s? tms?)
(assign-operation 'merge (coercing ->tms tms-merge) tms? flat?)
(assign-operation 'merge (coercing ->tms tms-merge) flat? tms?)
```

C.5 Conditionals

The conditional propagator is nontrivial because, when given a supported value say, it both must branch correctly even if given a supported `#f` (which would read as a true value if naively passed to Scheme's native `if`), and must attach the support of the predicate to the support of the result produced by that branch.

We accomplish this by introducing two additional generic operations and adding appropriate methods to them.

```
(define (conditional p if-true if-false output)
  (propagator (list p if-true if-false)
    (lambda ()
      (let ((predicate (content p)))
        (if (nothing? predicate)
            'done
            (add-content
             output
             (if (generic-true? predicate)
                 (generic-ignore-first predicate (content if-true))
                 (generic-ignore-first predicate (content if-false)))))))))

(define true? (lambda (x) (not (not x))))

(define generic-true? (make-generic-operator 1 'true? true?))
(assign-operation
 'true? (lambda (v&s) (generic-true? (v&s-value v&s))) v&s?)
(assign-operation
 'true? (lambda (tms) (generic-true? (tms-query tms))) tms?)

(define generic-ignore-first
  (make-generic-operator 2 'ignore-first ignore-first))
```

```
((lambda (name underlying-operation)
   (assign-operation
    name (v&s-unpacking underlying-operation) v&s? v&s?)
   (assign-operation
    name (coercing ->v&s underlying-operation) v&s? flat?)
   (assign-operation
    name (coercing ->v&s underlying-operation) flat? v&s?)

   (assign-operation
    name (full-tms-unpacking underlying-operation) tms? tms?)
   (assign-operation
    name (coercing ->tms underlying-operation) tms? v&s?)
   (assign-operation
    name (coercing ->tms underlying-operation) v&s? tms?)
   (assign-operation
    name (coercing ->tms underlying-operation) tms? flat?)
   (assign-operation
    name (coercing ->tms underlying-operation) flat? tms?))
 'ignore-first generic-ignore-first)
```

The same trick also applies to `apply`; that is, if one is `apply`ing a supported procedure, the value returned by that procedure must depend upon the justification of the procedure. The similarity of `if` with `apply` is not surprising given the Church encoding of booleans as procedures that ignore one or the other of their arguments.

One problem that can arise with this strategy, however, is that the resulting code is not tail recursive because `generic-ignore-first` must wait for the result in order to attach the dependencies from the procedure to it. We suspect that this problem may be solvable by attaching the justification to the continuation that will accept the result. [12]

References

[1] A. Abdelmeged, C. Hang, D. Rinehart, and K. Lieberherr. Superresolution and P-Optimality in Boolean MAX-CSP Solvers. *Transition*, 2007.

[2] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1984, 1996.

[3] A.H. Borning. *Thinglab–a constraint-oriented simulation laboratory.* PhD thesis, Stanford University, Stanford, CA, USA, 1979.

[4] Johan de Kleer. Local Methods for Localizing Faults in Electronic Circuits. AI Memo 394, MIT Artificial Intelligence Laboratory, Cambridge, MA, USA, 1976.

[5] Johan de Kleer and John Seely Brown. Model-based diagnosis in SOPHIE III. *Readings in Model-Based Diagnosis*, 1992.

[6] H.P. Dinesman. *Superior Mathematical Puzzles, with Detailed Solutions.* Simon and Schuster, New York, 1968.

[7] Jon Doyle. Truth Maintenance Systems for Problem Solving. AI Memo 419, MIT Artificial Intelligence Laboratory, Cambridge, MA, USA, 1978.

[8] M. Ernst, C. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. *Lecture Notes in Computer Science*, pages 186–211, 1998.

[9] Chris Hanson et al. MIT/GNU Scheme Reference Manual, December 2005. http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/index.html.

[10] R.W. Floyd. Nondeterministic Algorithms. *Journal of the ACM (JACM)*, 14(4):636–644, 1967.

[11] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers.* M.I.T. Press, 1993.

[12] Chris Hanson. Personal communication.

[13] Carl E. Hewitt. Planner: A language for proving theorems in robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 295–301, 1969.

[14] Karl Lieberherr. *Information Condensation of Models in the Propositional Calculus and the P=NP Problem.* PhD thesis, ETH Zurich, 1977. 145 pages, in German.

[15] David Allen McAllester. A Three Valued Truth Maintenance System. AI Memo 473, MIT Artificial Intelligence Laboratory, Cambridge, MA, USA, 1978.

[16] John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[17] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic Common Lisp. *University of Pennsylvania Institute for Research in Cognitive Science, tech. report IRCS-93-03*, 1993.

[18] Richard Stallman and Gerald Jay Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[19] Guy L. Steele Jr. The definition and implementation of a computer programming language based on constraints. AI Memo 595, MIT Artificial Intelligence Laboratory, Cambridge, MA, USA, 1980.

[20] Guy L. Steele Jr. and Gerald Jay Sussman. Constraints-A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, 14(1):1–39, 1980.

[21] Ramin Zabih, David Allen McAllester, and David Chapman. Nondeterministic Lisp with dependency-directed backtracking. In *Proceedings of AAAI 87, pages 59-64, July 1987*, 1987.