Revised Report on the Propagator Model

Alexey Radul and Gerald Jay Sussman

Abstract

In the past year we have made serious progress on elaborating the propagator programming model [2, 3]. Things have gotten serious enough to build a system that can be used for real experiments.

The most important problem facing a programmer is the revision of an existing program to extend it for some new situation. Unfortunately, the traditional models of programming provide little support for this activity. The programmer often finds that commitments made in the existing code impede the extension, but the costs of reversing those commitments are excessive.

Such commitments tend to take the form of choices of strategy. In the design of any significant system there are many implementation plans proposed for every component at every level of detail. However, in the system that is finally delivered this diversity of plans is lost and usually only one unified plan is adopted and implemented. As in an ecological system, the loss of diversity in the traditional engineering process has serious consequences.

The Propagator Programming Model is an attempt to mitigate this problem. It is a model that supports the expression and integration of multiple viewpoints on a design. It incorporates explicit structure to support the integration of redundant pieces and subsystems that solve problems in several different ways. It will help us integrate the diversity that was inherent in the design process into the delivered operational product.

The Propagator Programming Model is 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 computations it can make from information from the others. Cells accumulate information from the propagators that produce that information. The key idea here is additivity. New ways to make contributions can be added just by adding new propagators; if an approach to a problem doesn't turn out to work well, it can be identified by its premises and ignored, dynamically and without disruption.

This work was supported in part by the MIT Mind Machine Project.

Contents

1 Propagator System

Although most of this document introduces you to the Scheme-Propagator system that we have developed in MIT Scheme, the Propagator Model is really independent of the language. You should be able to write propagators in any language you choose, and others should be able to write subsystems in their favorite language that cooperate with your subsystems. What is necessary is that all users agree on the protocol by which propagators communicate with the cells that are shared among subsystems. These rules are very simple and we can enumerate them right here:

Cells must support three operations:

- add some content
- collect the content currently accumulated
- register a propagator to be notified when the accumulated content changes

When new content is added to a cell, the cell must merge the addition with the content already present. When a propagator asks for the content of a cell, the cell must deliver a complete summary of the information that has been added to it.

The merging of content must be commutative, associative, and idempotent. The behavior of propagators must be monotonic with respect to the lattice induced by the merge operation.

2 Getting Started

Scheme-Propagators is implemented in MIT/GNU Scheme, which you will need in order to use it. You will also need Scheme-Propagators itself, which you can download from Once you have it, go to the `propagator/` directory, start up your Scheme and load the main entry file with `(load "load")`. This gives you a read-eval-print loop (traditionally called a REPL for short) for both the Scheme-Propagators system and the underlying Scheme implementation. Check out the README for more on this.

Once you've got your REPL, you can start typing away at it to create propagator networks, give them inputs, ask them to do computations, and look at the results.

Here's a little propagator example that adds two and three to get five:

```
(define-cell a)
(define-cell b)
(add-content a 3)
(add-content b 2)
(define-cell answer (e:+ a b))
(run)
(content answer) ==> 5
```

Each of the parenthesized phrases above are things to type into the REPL, and the `==> 5` at the end is the result that Scheme will print. The results of all the other expressions are not interesting.

Let's have a closer look at what's going on in this example, to serve as a guide for more in-depth discussion later. `define-cell` is a Scheme macro for making and naming propagator cells:

```
(define-cell a)
```

creates a new cell and binds it to the Scheme variable `a`.

```
(define-cell b)
```

makes another one. Then `add-content` is the Scheme procedure that directly zaps some information into a propagator cell (all the propagators use it to talk to the cells, and you can too). So:

```
(add-content a 3)
```

puts a `3` into the cell named `a`, and:

```
(add-content b 2)
```

puts a 2 into the cell named b. Now e:+ (the naming convention will be explained later) is a Scheme procedure that creates a propagator that adds, attaches it to the given cells as inputs, and makes a cell to hold the adder's output and returns it. So:

```
(define-cell answer (e:+ a b))
```

creates an adding propagator, and also creates a cell, now called answer, to hold the result of the addition. Be careful! No computation has happened yet. You've just made up a network, but it hasn't done its work yet. That's what the Scheme procedure run is for:

```
(run)
```

actually executes the network, and only when the network is done computing does it give you back the REPL to interact with. Finally content is a Scheme procedure that gets the content of cells:

```
(content answer)
```

looks at what the cell named answer has now, which is 5 because the addition propagator created by e:+ has had a chance to do its job. If you had forgotten to type (run) before typing (content answer), it would have printed out #(*the-nothing*), which means that cell has no information about the value it is meant to have.

2.1 Examples

There are some basic examples of Scheme-Propagators programs in core/example-networks.scm; more elaborate examples are available in examples/.

3 The Details

Now that you know how to play around with our propagators we have to tell you what we actually provide. In every coherent system for building stuff there are primitive parts, the means by which they can be combined, and means by which combinations can be abstracted so that they can be named and treated as if they are primitive.

4 Making Propagator Networks

The ingredients of a propagator network are cells and propagators. The cells' job is to remember things; the propagators' job is to compute. The analogy is that propagators are like the procedures of a traditional programming language, and cells are like the memory

locations; the big difference is that cells accumulate partial information (which may involve arbitrary internal computations), and can therefore have many propagators reading information from them and writing information to them.

The two basic operations when making a propagator network are making cells and attaching propagators to cells. You already met one way to make cells in the form of `define-cell`; we will talk about more later, but let's talk about propagators first.

4.1 Attaching Basic Propagators: d@

The Scheme procedure `d@` attaches propagators to cells. The name `d@` is mnemonic for "diagram apply". For example, `p:+` makes adder propagators:

```
(d@ p:+ foo bar baz)
```

means attach a propagator that will add the contents of the cells named `foo` and `bar` and write the sum into the cell named `baz`. Once attached, whenever either the `foo` cell or the `bar` cell gets any new interesting information, the adding propagator will eventually compute the appropriate sum and give it to `baz` as an update.

`(d@ propagator boundary-cell ...)` Attaches a propagator to the given boundary cells. By convention, cells used as outputs go last. As a Scheme procedure, `d@` does not return a useful value.

As in Scheme, `p:+` is actually the name of a cell that contains a propagator constructor for attaching propagators that do addition. The first argument to `d@` can be any cell that contains any desired partial information (see Section 6) about a propagator constructor. Actual attachment of propagators will occur as the propagator constructor becomes sufficiently well constrained.

4.2 Propagator Expressions: e@

The `d@` style is the right underlying way to think about the construction of propagator networks. However, it has the unfortunate feature that it requires the naming of cells for holding all intermediate values in a computation, and in that sense programming with `d@` feels a lot like writing assembly language.

It is pretty common to have expressions: one's propagator networks will have some intermediate values that are produced by only one propagator, and consumed by only one propagator. In this case it is a drag to have to define and name a cell for that value, if one would just name it "the output of foo". Scheme-Propagators provides a syntactic sugar for writing cases like this in an expression style, like a traditional programming language.

The Scheme procedure `e@` attaches propagators in expression style. The name `e@` is mnemonic for "expression apply". The `e@` procedure is just like `d@`, except it synthesizes an extra cell to serve as the last argument to `d@`, and returns it from the `e@` expression (whereas the return value of `d@` is unspecified).

`(e@ propagator boundary-cell ...)` Attaches the given propagator to a boundary consisting of the given boundary cells augmented with an additional, synthesized cell. The synthesized cell goes last, because that is the conventional position for an output cell. Returns the synthesized cell as the Scheme return value of `e@`.

For example, here are two ways to do the same thing:

```
(define-cell x)
(define-cell y)
(define-cell z)
(d@ p:* x y z)
```

and:

```
(define-cell x)
(define-cell y)
(define-cell z (e@ p:* x y))
```

Generally the `e@` style is convenient because it chains in the familiar way

```
(e@ p:- w (e@ p:* (e@ p:+ x y) z))
```

Because of the convention that output cells are listed last, expressions in `e@` style build propagator networks that compute corresponding Lisp expressions.

On the other hand, the `d@` style is necessary when a propagator needs to be attached to a full set of cells that are already there. For example, if one wanted to be able to go back from `z` and one of `x` or `y` to the other, rather than just from `x` and `y` to `z`, one could write:

```
(define-cell x)
(define-cell y)
(define-cell z (e@ p:* x y))
(d@ p:/ z x y)
(d@ p:/ z y x)
```

and get a multidirectional constraint:

```
(add-content z 6)
(add-content x 3)
(run)
(content y) ==> 2
```

To save typing when the propagator being attached is known at network construction time, the `p:foo` objects are also themselves applicable in Scheme, defaulting to applying themselves in the `d@` style. Each also has an `e:foo` variant that defaults to the `e@` style. So the following also works:

```
(define-cell x)
(define-cell y)
(define-cell z (e:* x y))
(p:/ z x y)
(p:/ z y x)
```

4.3 Late Binding of Application

The preceding discusses attaching propagators to cells when the propagators being attached are known at network construction time. That will not always be the case. For example:

```
(define-cell operation)
(define-cell answer)
(d@ operation 3 4 answer)
(run)
(content answer)  ==>  nothing
```

We didn't say what operation to perform. This is not an error, but since nothing is known about what to do with the 3 and the 4, nothing is known about the answer. Now if we supply an operation, the computation will proceed:

```
(p:id p:* operation)
(run)
(content answer)  ==>  12
```

In fact, in this case, `d@` (or `e@`) will build an *apply propagator* that will wait until an operation appears in the `operation` cell, and then apply it.

What would have happened if we had left off the `d@` and just written `(operation 3 4 answer)`? If you put into operator position a cell that does not have a fully-known propagator at network construction time, it will be applied in diagram style by default. If you put into operator position a cell that contains a fully-known propagator at network construction time, it will be applied either in diagram style or expression style, as dependent on that propagator's default preference. `d@` and `e@` override these defaults.

4.4 Provided Primitives: p:foo and e:foo

Many propagator primitives directly expose procedures from the underlying Scheme, with the naming conventions that `p:foo`, and `e:foo` does the job `foo` to the contents of an appropriate pile of input cells and gives the result to an output cell (which is passed in as the last argument to `p:foo` and synthesized and returned by `e:foo`). `p:` is mnemonic for "propagator" and `e:` is mnemonic for "expression".

`(p:foo input ... output)` Attaches a propagator that does the `foo` job to the given input and output cells. `p:abs`, `p:square`, `p:sqrt`, `p:not`, `p:pair?`, and `p:null?` accept one input cell and one output cell. `p:+`, `p:-`, `p:*`, `p:/`, `p:=`, `p:<`, `p:>`, `p:<=`, `p:>=`, `p:and`, `p:or`, `p:eq?`, `p:eqv?`, `p:atan2`, and `p:expt`, accept two input cells and one output cell.

`(e:foo input ...)` The `e:foo` equivalents of all the `p:foo` propagator constructors are all available and accept the same number of input cells (and make their own output cell).

`(p:id input output)`, `(e:id input)` Attaches an identity propagator to the given cells. The identity propagator will continuously copy the contents of the `input` cell to the `output` cell.

`(p:== input ... output)`, `(e:== input ...)` These are variadic versions of `p:id`. The result is a star topology, with every input feeding into the one output.

`(p:switch control input output)`, `(e:switch control input)` Conditional propagation. The propagator made by `switch` copies its `input` to its `output` if and only if its `control` is "true". The presence of partial information (see Section 6) makes this interesting. For example, a `#t` contingent on some premise will cause `switch` to propagate, but the result written to the `output` will be contingent on that premise (in addition to any other premises the `input` may already be contingent on).

`(p:conditional control consequent alternate output)`,

`(e:conditional control consequent alternate)` Two-armed conditional propagation. May be defined by use of two `switch` propagators and a `not` propagator.

`(p:conditional-router control input consequent alternate)`,

`(e:conditional-router control input consequent)` Two-output-armed conditional propagation. This is symmetric with `conditional`; the `consequent` and `alternate` are possible output destinations.

4.5 Cells are Data Too

Cells, and structures thereof, are perfectly good partial information (see Section 6) and are therefore perfectly legitimate contents of other cells. The event that two different cells A and B find themselves held in the same third cell C means that A and B are now known to contain information about the same thing. The two cells are therefore merged by attaching `c:id` propagators to them so as to keep their contents in sync in the future.

`(p:deposit cell place-cell)`, `(e:deposit cell)` Grabs the given `cell` and deposits it into `place-cell`. The rule for merging cells has the effect that the given `cell` will be identified with any other cells that `place-cell` may come to hold.

`(p:examine place-cell cell)`, `(e:examine place-cell)` Grabs the given `cell` and deposits it into `place-cell`. The rule for merging cells has the effect that the given `cell` will be identified with any other cells that `place-cell` may come to hold.

In fact, `p:deposit` and `p:examine` are the same operation, except with the arguments reversed.

The `e:examine` variant includes an optimization: if the `place-cell` already contains a cell, `e:examine` will just Scheme-return that cell instead of synthesizing a new one and identifying it with the cell present.

4.6 Compound Data

Propagator compound data structures are made out of Scheme compound data structures that carry around cells collected as with `deposit`. The corresponding accessors take those cells out as with `examine`.

`(p:cons car-cell cdr-cell output)`, `(e:cons car-cell cdr-cell)` Constructs a propagator that collects the `car-cell` and the `cdr-cell`, makes a pair of them, and writes that pair into the `output` cell. This is like a binary `p:deposit`.

`(p:pair? input output)`, `(e:pair? input)` Attaches a propaagtor that tests whether input cell contains a pair.

`(p:null? input output)`, `(e:null? input)` Attaches a propaagtor that tests whether input cell contains the empty list.

`(p:car input output)`, `(e:car input)` Makes a propagator that identifies the given `output` with the cell in the `car` of the pair in the given `input`. This is like a `p:examine` of that field. Note that using `p:car` on an `input` implies the expectation that said `input` contains a pair. That wish is treated as a command, and a pair appears. If fact, `(p:car input output)` is equivalent to `(p:cons output nothing input)`.

The `e:` variant includes the same optimization that `e:examine` does: if the `input` already contains a pair with a cell in the `car`, `e:car` will just Scheme-return that cell instead of synthesizing a new one and identifying it with the cell present.

`(p:cdr input output)`, `(e:cdr input)` Same as `p:car` and `e:car`, except the other field of the pair.

Note that the identification of cells that merge is bidirectional, so information written into the `output` of a `p:car` will flow into the cell in the `car` of the pair in the `input` (and therefore into any other cells identified with it by other uses of `p:car` on the same pair). For example, in a program like:

```
(let-cell frob
  (let-cell (quux (e:car frob))
    ... quux ...)
  (let-cell (quux2 (e:car frob))
    ... quux2 ...))
```

the two cells named `quux` and `quux2` will end up identified, and the cell named `frob` will end up containing a pair whose car field will contain one of them.

Scheme pairs created by `p:cons` and company are partial information structures, and they merge by recursively merging their corresponding fields. Together with the rule for merging cells, the emergent behavior is unification (with a merge delay instead of the occurs check).

4.7 Propagator Constraints: c:foo and ce:foo

Although the primitive propagators are like functions in that they compute only from inputs to outputs, we can also define constraints, which may also derive information about the arguments of a function from information about the value. Constraints are so useful that many are predefined, and they have their own naming convention. `c:` stands for "constraining". A thing named `c:foo` is the constraining analogue of `p:foo`, in that in addition to attaching a propagator that does `foo` to its cells, it also attaches `foo-inverse` propagators that deduce "inputs" from "outputs". For example, the product constraint that we built in a previous section is available as `c:*`:

```
(define-cell x)
(define-cell y)
(define-cell z)
(d@ c:* x y z)

(add-content z 12)
(add-content y 4)
(run)
(content x) ==> 3
```

The `c:foo` objects, like the `p:foo` objects, are also self-applicable, and also default to applying themselves in diagram style:

```
(c:* x y z)  ==  (d@ c:* x y z)
```

The `c:foo` objects also have `ce:foo` analogues, that apply themselves in expression style:

```
(ce:* x y)  ==  (e@ c:* x y)
```

Of course, not every operation has a useful inverse, so there are fewer `c:` procedures defined than `p:`:

`(c:foo constrainee ...)` Attaches propagators to the given boundary cells that collectively constrain them to be in the `foo` relationship with each other. `c:+` and `c:*` accept three cells to constrain. `c:square`, `c:not`, and `c:id` accept two cells to constrain. `c:==` accepts any number of cells.

`(ce:foo constrainee ...)` Synthesizes one additional constrainee cell and attaches propagators that constrain the given cells to be in the `foo` relationship with the new one. Since the position of the synthesized cell in the argument list is fixed, some diagram style constraints have multiple expression style variants:

```
c:+       ce:+       ce:-
c:*       ce:*       ce:/
c:square  ce:square  ce:sqrt
c:not     ce:not
c:and     ce:and
c:or      ce:or
c:id      ce:id
c:==      ce:==
c:negate  ce:negate
c:invert  ce:invert
c:sin     ce:sin
c:cos     ce:cos
c:tan     ce:tan
c:exp     ce:exp
c:eq?     ce:eq
c:eqv?    ce:eqv
```

4.8 Constants and Literal Values

Programs have embedded constants all the time, and propagator programs are no different (except that constant values, like all other values, can be partial). We've already seen one way to put a Scheme value into a propagator program: the `add-content` procedure zaps a value straight into a cell. This is generally encouraged at the REPL, but frowned upon in actual programs. It is much nicer to use `constant` or `p:constant` (they're the same) to make a propagator that will zap your value into your cell for you:

```
(define-cell thing)
((constant 5) thing)
(content thing) ==> #(*the-nothing*)
(run)
(content thing) ==> 5
```

There is also an expression-oriented version, called, naturally, `e:constant`:

```
(define-cell thing (e:constant 5))
(run)
(content thing) ==> 5
```

4.9 Constant Conversion

In fact, inserting constants is so important, that there is one more nicification of this: whenever possible, the system will convert a raw constant (i.e. a non-cell Scheme object) into a cell, using `e:constant`.

Some examples:

```
(e:+ x 2)          ==   (e:+ x (e:constant 2))
(define-cell x 4)  ==   (define-cell x (e:constant 4))
(c:+ x y 0)        ==   (c:+ x y (e:constant 0))
```

4.10 Making Cells

Cells are the memory locations of the Scheme-Propagators language: Scheme variables whose bindings are cells correspond to Scheme-Propagators variables (Scheme variables whose bindings are other things look like syntax to Scheme-Propagators). We've already met one way to make cells:

```
(define-cell x)
```

creates a Scheme variable named `x` and binds a cell to it. The underlying mechanism underneath this is the procedure `make-cell`, which creates a cell and lets you do whatever you want with it. So you could write:

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

which would also make a Scheme variable named `x` and bind a cell to it. In fact, that is almost exactly what `define-cell` does, except that `define-cell` attaches some metadata to the cell it creates to make it easier to debug the network (see Section 10) and also does constant conversion (so `(define-cell x 5)` makes `x` a cell that will get a `5` put into it, whereas `(define x 5)` would just bind `x` to `5`).

Just as Scheme has several mechanisms of making variables, so Scheme-Propagators has corresponding ones. Corresponding to Scheme's `let`, Scheme-Propagators has `let-cells`:

```
(let-cells ((foo (e:+ x y))
            (bar (e:* x y)))
  ...)
```

will create the Scheme bindings `foo` and `bar`, and bind them to the cells made by `(e:+ x y)` and `(e:* x y)`, respectively (this code is only sensible if `x` and `y` are already bound to cells (or subject to constant conversion)). The new bindings will only be visible inside the scope of the `let-cells`, just like in Scheme; but if you attach propagators to them, the cells themselves will continue to exist and function as part of your propagator network.

One notable difference from Scheme: a cell in a propagator network, unlike a variable in Scheme, has a perfectly good "initial state". Every cell starts life knowing `nothing` about its intended contents; where Scheme variables have to start life in a weird "unassigned" state, `nothing` is a perfectly good partial information structure. This means that it's perfectly reasonable for `let-cells` to make cells with no initialization forms:

```
(let-cells (x y (foo (some thing))) ...)
```

creates cells named `x` and `y`, which are empty and have no propagators attached to them initially, and also a cell named `foo` like above. `let-cells` also recognizes the usage:

```
(let-cells ((x) (y) (foo (some thing))) ...)
```

by analogy with Scheme `let`.

Corresponding to Scheme's `let*`, Scheme-Propagators has `let-cells*`. `let-cells*` is to `let-cells` what `let*` is to `let`:

```
(let-cells* ((x)
             (y (e:+ x x)))
  ...)
```

will make a cell named `x` and a cell named `y` with an adder both of whose inputs are `x` and whose output is `y`.

Corresponding to Scheme's `letrec`, Scheme-Propagators has `let-cells-rec`. `let-cells-rec` has the same scoping rules as Scheme's `letrec`, namely that all the names it defines are available to all the defining forms. Moreover, since an "uninitialized" propagator cell can still start life in a perfectly sensible state, namely the state of containing `nothing`, `let-cells-rec` removes a restriction that Scheme's `letrec` enforced; namely, you may use the names defined by a given `let-cells-rec` directly in the defining forms, without any explicit intermediate delay in evaluation. For example:

```
(let-cells-rec ((z (e:+ x y))
                (x (e:- z y))
                (y (e:- z x)))
  ...)
```

is a perfectly reasonable bit of Scheme-Propagators code, and binds the names `x`, `y` and `z` to cells that are interconnected with the three propagators indicated.

Now, `let-cells`, `let-cells*`, and `let-cells-rec` are, like `define-cell`, basically a convenience over doing the same thing in Scheme with `let`, `let*`, `letrec` and `make-cell`. Also like `define-cell`, `let-cells`, `let-cells*`, and `let-cells-rec` do constant conversion (so in `(let-cells ((x 3)) ...)`, `x` becomes a cell, not a Scheme object), and attach debugging information to the cells they bind.

Since `let-cells` is plural (where `let` was number-neutral), Scheme-Propagators also define `let-cell` and `let-cell-rec` (`let-cell*` being useless) for the case when you just want to make one cell:

```
(let-cell x ...)              ==  (let-cells (x) ...)
(let-cell (x (e:+ y z)) ...)  ==  (let-cells ((x (e:+ y z))) ...)
(let-cell-rec (ones (e:cons 1 ones)) ...)  ==
(let-cells-rec ((ones (e:cons 1 ones))) ...)
```

Scheme-Propagators currently has no analogue of Scheme's named `let` syntax.

Finally, there is one more, somewhat sneaky way to make cells. The `e@` procedure makes and returns a cell to hold the "output" of the propagator being applied. These implicit cells are just like the implicit memory locations that Scheme creates under the hood for holding the return values of expressions before they get used by the next expression or assigned to variables.

4.11 Conditional Network Construction

The `switch` propagator does conditional propagation --- it only forwards its input to its output if its control is "true". As such, it serves the purpose of controlling the flow of data through an existing propagator network. However, it is also appropriate to control the construction of more network, for example to design recursive networks that expand themselves no further than needed. The basic idea here is to delay the construction of some chunk of network until some information appears on its boundary, and control whether said information appears by judicious use of `switch` propagators. The low-level tools for accomplishing this effect are `delayed-propagator-constructor` and `switch`. The supported user interface is:

`(p:when internal-cells condition-cell body ...)` Delays the construction of the body until sufficiently "true" (in the sense of `switch`) partial information appears in the `condition-cell`. The `condition-cell` argument is an expression to evaluate to produce the cell controlling whether construction of the `body` takes place. The `body` is an arbitrary collection of code, defining some amount of propagator network that will not be built until the controlling cell indicates that it should. The `internal-cells` argument is a list of the free variables in `body`. This is the same kind of kludge as the `import` clause in `define-propagator` (see Section 5.1).

`(e:when internal-cells condition-cell body ...)` Expression-style variant of `p:when`. Augments its boundary with a fresh cell, which is then synchronized with the cell returned from the last expression in `body` when `body` is constructed.

`(p:unless internal-cells condition-cell body ...)`

`(e:unless internal-cells condition-cell body ...)` Same as `p:when` and `e:when`, but reversing the sense of the control cell.

`(p:if internal-cells condition-cell consequent alternate)` Two-armed conditional construction. Just like a `p:when` and a `p:unless`: constructs the network indicated by the `consequent` form when the `condition-cell` becomes sufficiently "true", and constructs the network indicated by the `alternate` form when the `condition-cell` becomes sufficiently "false". Note that both can occur for the same `p:if` over the life of a single computation, for example if the `condition-cell` comes to have a TMS that includes a `#t` contingent on some premises and later a `#f` contingent on others.

`(e:if internal-cells condition-cell consequent alternate)` Expression-style variant of `p:if`.

5 Making New Compound Propagators

So, you know the primitives (the supplied propagators) and the means of combination (how to make cells and wire bunches of propagators up into networks). Now for the means of abstraction. A propagator constructor such as `p:+` is like a wiring diagram with a few holes where it can be attached to other structures. Supply `p:+` with cells, and it makes an actual propagator for addition whose inputs and outputs are those cells. How do you make compound propagator constructors?

The main way to abstract propagator construction is with the `define-d:propagator` and `define-e:propagator` Scheme macros. `define-d:propagator` defines a compound propagator in diagram style, that is, with explicit named parameters for the entire boundary of the compound:

```
(define-d:propagator (my-sum-constraint x y z)
  (p:+ x y z)
  (p:- z y x)
  (p:- z x y))
```

`define-e:propagator` defines a compound propagator in expression style, that is, expecting the body of the propagator to return one additional cell to add to the boundary at the end:

```
(define-e:propagator (double x)
  (e:+ x x))
```

Both defining forms will make variants with names beginning in `p:` and `e:`, that default to being applied in diagram and expression style, respectively. Note that this definition does not bind the Scheme variable `double`.

With these definitions we can use those pieces to build more complex structures:

```
(p:my-sum-constraint x (e:double x) z)
```

which can themselves be abstracted so that they can be used as if they were primitive:

```
(define-d:propagator (foo x z)
  (p:my-sum-constraint x (e:double x) z))
```

`define-propagator` is an alias for `define-d:propagator` because that's the most common use case.

Just as in Scheme, the definition syntaxes have a corresponding syntax for anonymous compound propagators, `lambda-d:propagator` and `lambda-e:propagator`.

Compound propagator constructors perform constant conversion:

```
(p:my-sum-constraint x 3 z)  ==  (p:my-sum-constraint x (e:constant 3) z)
```

`define-propagator` and `define-e:propagator` respect the `c:` and `ce:` naming convention, in that if the name supplied for definition begins with `c:` or `ce:`, that pair of prefixes will be used in the names actually defined instead of `p:` and `e:`. So:

```
(define-propagator (foo ...) ...)     defines  p:foo and e:foo
(define-propagator (p:foo ...) ...)   defines  p:foo and e:foo
(define-propagator (e:foo ...) ...)   defines  p:foo and e:foo
(define-propagator (c:foo ...) ...)   defines  c:foo and ce:foo
(define-propagator (ce:foo ...) ...)  defines  c:foo and ce:foo
```

5.1 Lexical Scope

Compound propagator definitions can be closed over cells available in their lexical environment:

```
(define-e:propagator (addn n)
  (define-e:propagator (the-adder x)
    (import n)
    (e:+ n x))
  e:the-adder)
```

`import` is a kludge, which is a consequence of the embedding of Scheme-Propagators into Scheme. Without enough access to the Scheme interpreter, or enough macrological wizardry, we cannot detect the free variables in an expression, so they must be listed explicitly by the user. Globally bound objects like `e:+` (and `p:addn` and `e:addn` if the above were evaluated at the top level) need not be mentioned.

5.2 Recursion

Propagator abstractions defined by `define-propagator` are expanded immediately when applied to cells. Therefore, magic is needed to build recursive networks, because otherwise the structure would be expanded infinitely far. As in Scheme, this magic is in `if`. The Scheme-Propagators construct `p:if` (which is implemented as a Scheme macro) delays the construction of the diagrams in its branches until sufficient information is available about the predicate. Specifically, the consequent is constructed only when the predicate is sufficiently "true", and the alternate is constructed only when the predicate is sufficiently "false". Note that, unlike in Scheme, these can both occur to the same `p:if`.

In Scheme-Propagators, the one-armed conditional construction construct `p:when` is more fundamental than the two-armed construct `p:if`. This is because, where Scheme's `if` is about selecting values, and so has to have two options to select from, `p:when` and `p:if` are about building machinery, and there is no particular reason why choosing among two pieces of machinery to construct is any more basic than choosing whether or not to construct one particular piece.

For example, here is the familiar recursive `factorial`, rendered in propagators with `p:if`:

```
(define-propagator (p:factorial n n!)
  (p:if (n n!) (e:= 0 n)
    (p:== 1 n!)
    (p:== (e:* n (e:factorial (e:- n 1))) n!)))
```

The only syntactic difference between this and what one would write in Scheme for this same job is that this is written in diagram style, with an explicit name for the cell that holds the answer, and that `p:if` needs to be told the names of the non-global variables that are free in its branches, just like the `import` clause of a propagator definition (and for the same kludgerous reason). `p:when` is the one-armed version. `p:unless` is also provided; it reverses the sense of the predicate.

Like everything else whose name begins with `p:`, `p:if` and co have expression-style variants. The difference is that the tail positions of the branches are expected to return cells, which are wired together and returned to the caller of the `e:if`. Here is `factorial` again, in expression style:

```
(define-e:propagator (e:factorial n)
  (e:if (n) (e:= 0 n)
    1
    (e:* n (e:factorial (e:- n 1)))))
```

Looks familiar, doesn't it?

6 Using Partial Information

Partial, cumulative information is essential to multidirectional, non-sequential programming. Each "memory location" of Scheme-Propagators, that is each cell, maintains not "a value", but "all the information it has about a value". Such information may be as little as "I know absolutely nothing about my value", as much as "I know everything there is to know about my value, and it is `42`", and many possible variations in between; and also one not-in-between variation, which is "Stop the presses! I know there is a contradiction!"

All these various possible states of information are represented (perforce) as Scheme objects. The Scheme object `nothing` represents the information "I don't know anything". This requires only a single Scheme object, because not knowing anything is a single state of knowledge. Most Scheme objects represent "perfect, consistent" information: the Scheme object `5` represents the information "I know everything there is to know, and the answer is `5`." There are also several Scheme types provided with the system that denote specific other states of knowledge, and you can make your own. For example, objects of type `interval?` contain an upper bound and a lower bound, and represent information of the form "I know my value is between this real number and that one."

The way to get partial knowledge into the network is to put it into cells with `add-content` or constant propagators. For example:

```
(define-cell x (make-interval 3 5))
```

produces a cell named `x` that now holds the partial information `(make-interval 3 5)`, which means that its value is between `3` and `5`.

Partial information structures are generally built to be contagious, so that once you've inserted a structure of a certain type into the network, the normal propagators will generally produce answers in kind, and, if needed, coerce their inputs into the right form to cooperate. For example, if `x` has an interval like above,

```
(define-cell y (e:+ x 2))
```

will make an adder that will eventually need to add `2` to the interval between `3` and `5`. This is a perfectly reasonable thing to ask, because both `2` and `(make-interval 3 5)` are states of knowledge about the inputs to that adder, so it ought to produce the best possible representation of the knowledge it can deduce about the result of the addition. In this case, that would be the interval between `5` and `7`:

```
(run)
(content y)  ==>  #(interval 5 7)
```

The key thing about partial information is that it's cumulative. So if you also added some other knowledge to the `y` cell, it would need to merge with the interval that's there to represent the complete knowledge available as a result:

```
(add-content y (make-interval 4 6))
(content y)  ==>  #(interval 5 6)
```

If incoming knowledge hopelessly contradicts the knowledge a cell already has, it will complain:

```
(add-content y 15)  ==>  An error
```

stop the network mid-stride, and give you a chance to examine the situation so you can debug the program that led to it, using the standard MIT Scheme debugging facilities.

The partial information types are defined by a suite of Scheme procedures. The ones defining the actual partial information types are `equivalent?`, `merge`, and `contradictory?`, which test whether two information structures represent the same information, merge given information structures, and test whether a given information structure represents an impossible state, respectively. Each partial information structure also defines the way various propagators treat it. The behavior in the control position of a `switch` propagator and in the operator position of an apply propagator are particularly important.

7 Built-in Partial Information Structures

The following partial information structures are provided with Scheme-Propagators:

- nothing
- just a value
- intervals
- propagator cells
- compound data
- closures
- supported values
- truth maintenance systems
- contradiction

7.1 Nothing

`nothing` A single Scheme object that represents the complete absence of information.

`(nothing? thing)` A predicate that tests whether a given Scheme object is the `nothing` object.

`nothing` is `equivalent?` only to itself.

`nothing` never contributes anything to a merge --- the merge of anything with `nothing` is the anything.

`nothing` is not `contradictory?`.

Strict propagators, such as ones made by `p:+`, output `nothing` if any of their inputs are `nothing`.

A `switch` whose control cell contains `nothing` will emit `nothing`.

An apply propagator whose operator cell contains `nothing` will not do anything.

7.2 Just a Value

A Scheme object that is not otherwise defined as a partial information structure indicates that the content of the cell is completely known, and is exactly (by `eqv?`) that object. Note: floating point numbers are compared by approximate numerical equality; this is guaranteed to screw you eventually, but we don't know how to do better.

Raw Scheme objects are `equivalent?` if they are `eqv?` (or are approximately equal floating point numbers).

Non-`equivalent?` raw Scheme objects merge into the contradiction object.

A raw Scheme object is never `contradictory?`.

A `switch` interprets any non-`#f` raw Scheme object in its control cell as true and forwards its input cell to its output cell unmodified. A `switch` whose control cell is `#f` emits `nothing` to its output cell.

An apply propagator whose operator cell contains a raw Scheme procedure will apply it to the boundary cells. It is an error for a raw Scheme object which is not a Scheme procedure to flow into the operator cell of an apply propagator.

7.3 Numerical Intervals

An object of type `interval?` has fields for a lower bound and an upper bound. Such an object represents the information "This value is between these bounds."

`(make-interval low high)` Creates an interval with the given lower and upper bounds

`(interval-low interval)` Extracts the lower bound of an interval

`(interval-high interval)` Extracts the upper bound of an interval

`(interval? thing)` Tests whether the given object is an interval

Two interval objects are `equivalent?` if they are the same interval. An interval is `equivalent?` to a number if both the upper and lower bounds are that number.

Arithmetic can be performed on intervals. They can be compared, and the comparison predicates will have a truth value only when no future shrinkage of the intervals can change that value. For example, `(e:< int1 int2)` will be true only if `(e:< (interval-high int1) (interval-low int2))`; it will be false only if `(e:>= (interval-low int1) (interval-high int2))`; otherwise the result of the comparison is `nothing`.

Interval objects merge with each other by intersection. Interval object merge with numbers by treating the number as a degenerate interval and performing intersection (whose result will either be that number or an empty interval). Interval objects merge with other raw Scheme objects into the contradiction object.

An interval object is `contradictory?` if and only if it represents a strictly empty interval (that is, if the upper bound is strictly less than the lower bound).

The arithmetic propagators react to interval objects by performing interval arithmetic.

A `switch` propagator treats any interval object in its control as a non-`#f` object and forwards its input to its output.

It is an error for an interval object to appear in the operator position of an apply propagator.

As an interval arithmetic facility, this one is very primitive. It cannot extract new information from division by an interval that contains zero, because that would require intervals around the point at infinity. The main purpose of including intervals is to have a partial information structure with an intuitive meaning, and that requires nontrivial operations on the information it is over.

7.4 Propagator Cells as Partial Information

A propagator cell interpreted as partial information is an indirection: it means "I contain the structure that describes this value". Cells can appear as the contents of cells or other structures via the `deposit` and `examine` propagators (see Section 4.5).

Propagator cells are `equivalent?` if they are known to contain information about the same subject. This occurs only if they are identically the same cell, or if they have previously been unconditionally identified (by merging).

Propagator cells merge with each other by attaching bidirectional identity propagators that keep the contents of the cells in sync. These identity propagators will cause the contents of the cells to merge, both now and in the future.

A propagator cell is never `contradictory?`.

7.5 Compound Data

A Scheme pair is partial information that means "This object is a pair. My car and cdr contain cells that describe the car and cdr of this object." A Scheme empty list means

"This object is the empty list".

The propagators `p:cons`, `e:cons`, `p:car`, `e:cdr`, `p:pair?`, `e:pair?`, `p:null?`, and `e:null?` (see Section 4.6) introduce and examine pairs and empty lists.

Two pairs are `equivalent?` if their cars and cdrs are both `equivalent?`. A pair is not `equivalent?` to any non-pair. The empty list is only `'equivalent?` to itself.

Pairs merge by recursively merging the `car` and `cdr` fields. Given the behavior of propagator cells as mergeable data, the effect will be unification (with a delay instead of the occurs check). A pair merged with a Scheme object of a different type will produce a contradiction. An empty list merged with anything that is not the empty list will produce a contradiction.

Neither a pair nor the empty list is ever `contradictory?`.

A `switch` propagator treats any pair or empty list in its control as a non-`#f` object and forwards its input to its output.

It is an error for a pair or the empty list to appear in the operator position of an apply propagator.

Other compound data structures can be made partial information that behaves like pairs using `define-propagator-structure`.

`(define-propagator-structure type constructor accessor ...)` Declares that additional Scheme data structures are partial information like pairs, and defines appropriate propagators that handle them. For example:

```
(define-propagator-structure pair? cons car cdr)
```

is the declaration that causes Scheme pairs to `merge`, be `equivalent?`, and be `contradictory?` the way they are, and defines the propagators `p:pair?`, `e:pair?`, `p:cons`, `e:cons`, `p:car`, and `e:cdr`.

7.6 Closures

Propagator closures as mergeable data behave like a compound data structure. A closure is a code pointer together with an environment. The code pointer is a Scheme procedure; the environment is a map from names to cells, and as such is a compound structure containing cells. Code pointers merge by testing that they point to the same code (merging closures with different code produces a contradiction), and environments merge by merging all the cells they contain in corresponding places.

`lambda-d:propagator`, `lambda-e:propagator` Scheme-Propagators syntax for anonymous compound propagator constructors (which are implemented as closures).

`define-propagator` Internally produces lambda-d:propagator or lambda-e:propagator and puts the results into appropriately named cells.

7.7 Truth Maintenance Systems

A Truth Maintenance System (TMS) is a set of contingent values. A contingent value is any partial information object that describes the "value" in the cell, together with a set of premises. The premises are Scheme objects that have no interesting properties except identity (by `eq?`). A worldview defines which premises are believed.

The meaning of a TMS as information is the logical `and` of the meanings of all of its contingent values. The meaning of each contingent value is an implication: The conjunction of the premises implies the contingent information. Therefore, given a worldview, some of the contingent information is believed and some is not. If the TMS is queried, it produces the best summary it can of the believed information, together with the full set of premises that information is contingent upon.

In this system, there is a single current global worldview, which starts out believing all premises. The worldview may be changed to exclude (or re-include) individual premises, allowing the user to examine the consequences of different consistent subsets of premises.

`(kick-out! premise)` Remove the given premise from the current worldview.

`(bring-in! premise)` Return the given premise to the current worldview.

`(premise-in? premise)` Is the given premise believed in the current worldview?

`(contingent info premises)` Constructs a contingency object representing the information that the given info is contingent on the given list of premises.

`(contingent-info contingency-object)` The information that is contingent.

`(contingent-premises contingency-object)` The list of premises on which that information is contingent.

`(contingency-object-believed? contingency-object)` Whether the given contingency object is believed.

`(make-tms contingency-object-list)` Constructs a TMS with the given contingency objects as its initial set.

`(tms-query tms)` Returns a contingency object representing the strongest deduction the given TMS can make in the current worldview. tms-query gives the contingency with the strongest contingent information that is believed in the current worldview. Given that desideratum, tms-query tries to minimize the premises that information is contingent upon.

Calling `initialize-scheduler` resets the worldview to believing all premises.

TMSes are `equivalent?` if they contain equivalent contingent objects. Contingent objects are equivalent if they have equivalent info and the same set of premises.

TMSes merge by appending their lists of known contingencies (and sweeping out redundant ones).

Strict propagators react to TMSes by querying them to obtain ingredients for computation. The result of a computation is contingent on the premises of the ingredients that contribute to that result.

If a TMS appears in the control of a `switch`, the `switch` will first query the TMS to extract a contingent object. The `switch` will choose whether to forward its input or not based on the info that is contingent, but if it does forward, it will additionally make the result contingent upon the premises on which that info was contingent (as well as any premises on which the input may have been contingent). If the input itself is a TMS, `switch` queries it and (possibly) forwards the result of the query, rather than forwarding the entire TMS. For example:

```
(define-cell frob (make-tms (contingent 4 '(bill))))
(define-cell maybe-frob (e:switch (make-
tms (contingent #t '(fred))) frob))
(run)
(tms-query (content maybe-frob))  ==>  #(contingent 4 (bill fred))
```

If a TMS appears in the operator cell of an apply propagator, the apply propagator will query the TMS. If the result of the query is a contingent propagator constructor, the apply propagator will execute that constructor in a sandbox that ensures that the premises on which the constructor was contingent are both forwarded to the constructed propagator's inputs and attached to the constructed propagator's outputs. For example, suppose Bill wanted us to add 3 to 4:

```
(define-cell operation)
(define-cell answer)
(p:switch (make-tms (contingent #t '(bill))) p:+ operation)
(d@ operation 3 4 answer)
(run)
(tms-query (content answer))  ==>  #(contingent 7 (bill))
```

The `answer` cell contains a 7 contingent on the Bill premise. This is the right thing, because that answer depends not only on the inputs to the operation being performed, but also on the identity of the operation itself.

7.8 Contradiction

The Scheme object `the-contradiction` represents a completely contradictory state of information. If a cell ever finds itself in such a contradictory state, it will signal an error. The explicit `the-contradiction` object is useful, however, for representing contradictory information in recursive contexts. For example, a truth maintenance system may discover

that some collection of premises leads to a contradiction --- this is represented by a `the-contradiction` object contingent on those premises.

`the-contradiction` A Scheme object representing a contradictory state of information with no further structure.

`the-contradiction` is `equivalent?` only to itself.

Any information state merges with `the-contradiction` to produce `the-contradiction`.

`the-contradiction` is `contradictory?`.

Propagators cannot operate on `the-contradiction` because any cell containing it will signal an error before any such propagator might run.

7.9 Implicit Dependency-Directed Search

If a cell discovers that it contains a TMS that harbors a contingent contradiction, the cell will signal that the premises of that contradiction form a nogood set, and that nogood set will be recorded. For the worldview to be consistent, at least one of those premises must be removed. The system maintains the invariant that the current worldview never has a subset which is a known nogood.

If a nogood set consists entirely of user-introduced premises, the computation will be suspended, a description of the nogood set will be printed, and the user will have the opportunity to remove an offending premise (with `kick-out!`) and, if desired, resume the computation (with `run`).

There is also a facility for introducing hypothetical premises that the system is free to manipulate automatically. If a nogood set contains at least one hypothetical, some hypothetical from that nogood set will be retracted, and the computation will proceed.

`(p:amb cell)`, `(e:amb)` A propagator that emits a TMS consisting of a pair of contingencies. One contains the information `#t` contingent on one fresh hypothetical premise, and the other contains the information `#f` contingent on anther. `amb` also tries to maintain the invariant that exactly one of those premises is believed. If doing so does not cause the current worldview to believe a known nogood set, `amb` can just `bring-in!` one premise or the other. If the current worldview is such that bringing either premise in will cause a known nogood set to be believed, then, by performing a cut, the `amb` discovers and signals a new nogood set that does not include either of them. Together with the reaction of the system to nogood sets, this induces an emergent satisfiability solver by the resolution principle.

`(p:require cell)`, `(e:require)` A propagator that requires its given cell to be true (to wit, signals contradictions if it is not).

`(p:forbid cell)`, `(e:forbid)` A propagator that forbids its given cell from being true (to wit, signals contradictions if it is).

`(p:one-of input ... output)`, `(e:one-of input ...)` An n-ary version of `amb`. Picks one of the objects in the given input cells using an appropriate collection of `amb` and `switch` propagators and puts it into its output cell.

`(require-distinct cells)` Requires all of the objects in its list of input cells to be distinct (in the sense of `eqv?`)

8 Making New Kinds of Partial Information

The procedures defining the behavior of partial information are generic, and therefore extensible. The ones that define the actual partial information types are `equivalent?`, `merge`, and `contradictory?`, which test whether two information structures represent the same information, merge given information structures, and test whether a given information structure represents an impossible state, respectively. In addition, the primitive propagators are equipped with generic operations for giving them custom behaviors on the various information structures. The generic operation `binary-map` is very useful for the circumstance when all the strict propagators should handle a particular information type uniformly.

To create your own partial information structure, you should create an appropriate Scheme data structure to represent it, and then add handlers to the operations `equivalent?`, `merge`, and `contradictory?` to define that data structure's interpretation as information. In order to do anything useful with your new information structure, you will also need to make sure that the propagators you intend to use with it can deal with it appropriately. You can of course create custom propagators that handle your partial information structure. Standard generic operations are also provided for extending the built-in primitive propagators to handle new partial information types. Compound propagators are a non-issue because they will just pass the relevant structures around to the appropriate primitives.

It is also important to make sure that your new partial information structure intermixes and interoperates properly with the existing ones (see Section 7).

Method addition in the generic operation system used in Scheme-Propagators is done with the `defhandler` procedure:

```
(defhandler operation handler arg-predicate ...)
```

The generic operations system is a predicate dispatch system. Every handler is keyed by a bunch of predicates that must accept the arguments to the generic procedure in turn; if they do, that handler is invoked. For example, merging two intervals with each other can be defined as:

```
(defhandler merge intersect-intervals interval? interval?)
```

You can also define your own generic operations, but that is not relevant here.

8.1 An Example: Adding Interval Arithmetic

The first step is to define a data structure to represent an interval. Intervals have upper and lower bounds, so a Scheme record structure with constructor `make-interval`, accessors `interval-low` and `interval-high`, and predicate `interval?` will do.

The second step is to define handlers for the generic operations that every partial information structure must implement. Assuming appropriate procedures for intersecting intervals and for testing them for equality and emptiness, those handlers would be:

```
(defhandler equivalent? interval-equal? interval? interval?)
(defhandler merge intersect-intervals interval? interval?)
(defhandler contradictory? empty-interval? interval?)
```

To make intervals interoperate with numbers in the same network, we can add a few more handlers:

```
(define (number=interval? number interval)
  (= number (interval-low interval) (interval-high interval)))
(defhandler equivalent? number=interval? number? interval?)
(defhandler equivalent? (binary-
flip number=interval?) interval? number?)

(define (number-in-interval number interval)
  (if (<= (interval-low interval) number (interval-
high interval))
      number
      the-contradiction))
(defhandler merge number-in-interval number? interval?)
(defhandler merge (binary-flip number-in-interval) interval? number?)
```

The third step is to teach the arithmetic propagators to handle intervals. Interval arithmetic does not fit into the `binary-map` worldview (see Section 8.5) so the only way to do intervals is to individually add the appropriate handlers to the generic procedures underlying the primitive propagators:

```
(defhandler generic-+ add-interval interval? interval?)
(defhandler generic-- sub-interval interval? interval?)
(defhandler generic-* mul-interval interval? interval?)
(defhandler generic-/ div-interval interval? interval?)
(defhandler generic-sqrt sqrt-interval interval?)
;; ...
```

In order for the binary propagators to handle the situation where that propagator has an interval on one input and a number on the other, further handlers need to be added that tell it what to do in those circumstances. The generic procedure system has been extended with support for automatic coercions for this purpose.

8.2 Generic Coercions

Every number can be seen as an interval (whose lower and upper bounds are equal). The definition of arithmetic on mixed intervals and numbers can be deduced from the definitions of arithmetic on just intervals, arithmetic on just numbers, and this procedure for viewing numbers as intervals. The generic operations system provided with Scheme-Propagators has explicit support for this idea.

`(declare-coercion-target type [ default-coercion ])` This is a Scheme macro that expands into the definitions needed to declare `type` as something that other objects may be coerced into. If supplied, it also registers a default coercion from anything declared coercible to `type`.

`declare-coercion-target` defines the procedure `type-able?`, which tests whether a given object has been declared to be coercible to `type`, and the procedure `->type`, which does that coercion. These rely on the type-tester for `type` already being defined and named `type?`. For example:

```
(declare-coercion-target interval)
```

relies on the procedure `interval?` and defines the procedures `->interval` and `interval-able?`. This call does not declare a default means of coercing arbitrary objects into intervals.

`(declare-coercion from-type to-coercer [ mechanism ])` Declares that the given `from-type` is coercible by the given coercer operation, either by the given `mechanism` if supplied or by the default mechanism declared in the definition of the given coercer. For example:

```
(declare-coercion number? ->interval (lambda (x) (make-interval x x)))
```

declares that Scheme number objects may be coerced to intervals whose lower and upper bounds are equal to that number. After this declaration, `interval-able?` will return true on numbers, and `->interval` will make intervals out of numbers.

`(defhandler-coercing operation handler coercer)` The given generic operation must be binary. Defines handlers for the given generic operation that have two effects: `handler` is invoked if that operation is given two arguments of the type corresponding to `coercer`; and if one argument is of that type and the other has been declared coercable to that type it will be so coerced, and then handler will be invoked. For example:

```
(defhandler-coercing generic-+ add-interval ->interval)
```

declares that intervals should be added by `add-interval`, and that anything `interval-able?` can be added to an interval by first coercing it into an interval with `->interval` and then doing `add-interval`. This subsumes

```
(defhandler generic-+ add-interval interval?  interval?)
```

`defhandler-coercing` may only be called after a call to `declare-coercion-target` defining the appropriate coercer and coercability tester procedures (but the various specific coercions may be declared later).

8.3 The Partial Information Generics

`(equivalent?  info1 info2) ==> #t or #f`

The `equivalent?` procedure is used by cells to determine whether their content has actually changed after an update. Its job is to ascertain, for any two partial information structures, whether they represent the same information. As a fast path, any two `eqv?` objects are assumed to represent equivalent information structures. The default operation on `equivalent?` returns false for any two non-`eqv?` objects.

A handler for `equivalent?` is expected to accept two partial information structures and return `#t` if they represent semantically the same information, and `#f` if they do not.

The built-in `equivalent?` determines an equivalence relation. Extensions to it must maintain this invariant.

`(merge info1 info2) ==> new-info`

The `merge` procedure is the key to the propagation idea. Its job is to take any two partial information structures, and produce a new one that represents all the information present in both of the input structures. This happens every time a propagator gives a cell some new information. Any two `equivalent?` information structures merge to identically the first of them. The default operation for `merge` on a pair of non-`equivalent?` structures that the handlers for `merge` do not recognize is to assume that they cannot be usefully merged, and return `the-contradiction`.

A handler for `merge` is expected to accept two partial information structures and return another partial information structure that semantically includes all the information present in both input structures. The handler may return `the-contradiction` to indicate that the two given partial information structures are completely mutually exclusive.

`merge` is expected to determine a (semi-)lattice (up to equivalence by `equivalent?`). That is

- associativity:

```
(merge X (merge Y Z))  ~  (merge (merge X Y) Z)
(equivalent? (merge X (merge Y Z)) (merge (merge X Y) Z)) ==> #t
```

- commutativity:

```
(merge X Y)  ~  (merge Y X)
(equivalent? (merge X Y) (merge Y X)) ==> #t
```

- idempotence:

```
(X ~ Y) implies (X ~ (merge X Y))
(or (not (equivalent? X Y)) (equivalent? X (merge X Y))) ==> #t
```

`(contradictory? info) ==> #t or #f`

The `contradictory?` procedure tests whether a given information structure represents an impossible situation. `contradictory?` states of information may arise in the computation without causing errors. For example, a TMS (see Section 7.7) may contain a contradiction in a contingent context, without itself being `contradictory?`. But if a `contradictory?` object gets to the top level, that is if a cell discovers that it directly contains a `contradictory?` state of information, it will signal an error and stop the computation.

A handler for `contradictory?` is expected to accept a partial information structure, and to return `#t` if it represents an impossible situation (such as an empty interval) or `#f` if it does not.

8.3.1 The Full Story on Merge

The description of `merge` as always returning a new partial information structure is an approximation. Sometimes, `merge` may return a new partial information structure together with instructions for an additional effect that needs to be carried out. For example, when merging two propagator cells (see Section 7.4), the new information is just one of those cells, but the two cells also need to be connected with propagators that will synchronize their contents. For another example, in Scheme-Propagators, if a merge produces a TMS (see Section 7.7) that contains a contingent contradiction, the premises that contradiction depends upon must be signalled as a nogood set (that this requires signalling and is not just another partial information structure is a consequence of an implementation decision of TMSes in Scheme-Propagators).

The fully nuanced question that `merge` answers is

> "What do I need to do to the network in order to make it reflect the discovery that these two information structures are about the same object?"

In the common case, the answer to this question is going to be "Record: that object is best described by this information structure". This answer is represented by returning the relevant information structure directly. Another possible answer is "These two information structures cannot describe the same object." This answer is represented by returning `the-contradiction`. Other answers, such as "Record this information structure and connect these two cells with synchronizing propagators", are represented by the `effectful` data structure, which has one field for a new partial information structure to record, and one field for a list of other effects to carry out. These instructions are represented as explicit objects returned from `merge` rather than being carried out directly because this allows recursive calls to `merge` to modify the effects to account for the context in which that

`merge` occurs. For example, if a merge of two cells occurs in a contingent context inside a merge of two TMSes, then the instructions to connect those two cells must be adjusted to make the connection also contingent on the appropriate premises.

`(make-effectful info effects)` Constructs a new effectful result of merge, with the given new partial information structure and the given list of effects to carry out. If the resulting effectful object reaches the top level in a cell, those effects will be executed in the order they appear in the list.

`(effectful-info effectful)` Returns the new information content carried in the given effectful object.

`(effectful-effects effectful)` Returns the list of effects that this effectful object carries.

`(effectful? thing)` Tells whether the given object is an effectful object.

`(->effectful thing)` Coerces a possibly-effectless information structure into an effectful object. If the `thing` was already effectful, returns it, otherwise wraps it into an effectful object with an empty list of effects.

`(effectful-> effectful)` Attempts to coerce an effectful object into an explicitly effectless one. If the given effectful object was not carrying any effects that would have any effect when executed, returns just the information structure it was carrying. Otherwise, returns the given effectful object.

`(effectful-bind effectful func)` Runs the given `func` on the information content in the given `effectful` object, and reattaches any effects. The effectful object may actually be a partial information structure without explicit effects. The func may return a new partial information structure or a new effectful object. The overall result of `effectful-bind` is the information returned by the call to `func`, together with all the effects in the original effectful object, and any effects in the return value of the `func`. The former effects are listed first.

`(effectful-list-bind effectfuls func)` Like `effectful-bind`, but accepts a list of effectful objects, and calls the `func` on the list of their information contents.

There are two reasons why this matters to a user of the system. First, callers of `merge` (for example recursive ones in contexts where a new partial information structure is defined that may contain arbitrary other ones) must be aware that `merge` may return an `effectful` object. In this case, it is the responsibility of the caller to `merge` to shepherd the effects appropriately, adjusting them if necessary. For example, the `merge` handler for two pairs recursively merges the cars and cdrs of the pairs. If either of those recursive merges produces effects, the pair merge forwards all of them. Here is the code that does that:

```
(define (pair-merge pair1 pair2)
  (effectful-bind (merge (car pair1) (car pair2))
    (lambda (car-answer)
      (effectful-bind (merge (cdr pair1) (cdr pair2))
        (lambda (cdr-answer)
          (cons car-answer cdr-answer))))))

(defhandler merge pair-merge pair? pair?)
```

N.B.: The car merge and the cdr merge may both produce effects. If so, these effects will be executed in FIFO order, that is, car effects first, then cdr effects. This order is an arbitrary decision that we as the designers of Scheme-Propagators are not committed to. All effects built into Scheme-Propagators are independent, in that their executions commute.

Scheme-Propagators has two built-in effect types: `cell-join-effect`, defined in `core/cells.scm`, instructs the system to make sure two cells are joined by synchronizing propagators; `nogood-effect`, defined in `core/contradictions.scm`, instructs the system to record that a list of premises constitutes a nogood set. (The error that the system signals when discovering a toplevel contradiction is not an effect in this sense).

Second, a new partial information structure may want to have some side-effect when merged. This must be accomplished through returning an appropriate `effectful` object containing appropriate instructions. New types of effects can be defined for that purpose. For example, the built-in TMSes are added to the system through this mechanism.

The handling of effects is extensible through two generic procedures.

`(execute-effect effect)` The `execute-effect` procedure is used by cells to actually execute any effects that reach the top level. A handler for `execute-effect` should execute the effect specified by the given effect object. The return value of `execute-effect` is not used.

`(redundant-effect? effect) ==> #t or #f` The `redundant-effect?` procedure is used to determine which effects will predictably have no effect if executed, so they may be removed. For example, synchronizing a cell to itself, or synchronizing two cells that are already synchronized, are redundant effects. Detecting redundant effects is important for testing network quiescence.

The default operation of `redundant-effect?` is to return `#f` for all effects, which is conservative but could lead to excess computation in the network. A handler for `redundant-effect?` is expected to return `#t` if the effect will provably have no consequence on any values to be computed in the future, or `#f` if the effect may have consequences.

If an effect is generated by a `merge` that occurs in a contingent context in a TMS, the TMS will modify the effect to incorporate the contingency. This mechanism is also

extensible. To teach TMSes about making new effects contingent, add handlers to the generic operation `generic-attach-premises`.

`((generic-attach-premises effect) premises) ==> new-effect` The `generic-attach-premises` procedure is used by the TMS machinery to modify effects produced by merges of contingent information. A handler for `generic-attach-premises` must return a procedure that will accept a list of premises and return a new effect, which represents the same action but appropriately contingent on those premises. In particular, the consequences of the action must be properly undone or made irrelevant if any premises supporting that action are retracted. For example, the instruction to join two cells by synchronizing propagators is made contingent on premises by causing those synchronizing propagators to synchronize contingently.

8.4 Individual Propagator Generics

Most primitive propagators are actually built from generic Scheme functions. Those propagators can therefore be extended to new partial information types just by adding appropriate methods to their Scheme generic operations. This is what we did in the interval example.

`(generic-foo argument ...) ==> result` A generic procedure for carrying out the `foo` job over any desired partial information inputs, producing an appropriately partial result. `generic-abs`, `generic-square`, `generic-sqrt`, `generic-not`, `generic-pair?`, and `generic-null?` accept one input. `generic-+`, `generic--`, `generic-*`, `generic-/`, `generic-=`, `generic-<`, `generic->`, `generic-<=`, `generic->=`, `generic-and`, `generic-or`, `generic-eq?`, `generic-eqv?`, `generic-expt`, and `generic-switch` accept two inputs.

Don't forget to teach the propagators what to do if they encounter a partial information structure on one input and a different one on another --- if both represent states of knowledge about compatible ultimate values, it should be possible to produce a state of knowledge about the results of the computation (though in extreme cases that state of knowledge might be `nothing`, implying no new information produced by the propagator).

8.5 Uniform Applicative Extension of Propagators

Also, almost all primitive propagators are wrapped with the `nary-mapping` wrapper function around their underlying generic operation. This wrapper function is an implementation of the idea of applicative functors [1], so if your partial information structure is an applicative functor, you can use this to teach most propagators how to handle it.

The propagators wrapped in `nary-mapping` are exactly the strict propagators. This includes all the built-in propagators except `:deposit`, `:examine`, `:cons`, `:car`, and `:cdr`

because those operate on cells rather than their contents, and `:amb` because it essentially has no inputs.

`((binary-map info1 info2) f) ==> new-info` The generic procedure `binary-map` encodes how to apply a strict function to partial information arguments. `binary-map` itself is generic over the two information arguments, and is expected to return a handler that will accept the desired function `f` and properly apply it. For example, consider contingent information. A strict operation on the underlying information that is actually contingent should be applied by collecting the premises that both inputs are contingent on, applying the function, and wrapping the result up in a new contingency object that contains the result of the function contingent upon the set-union of the premises from both inputs:

```
(define (contingency-binary-map c1 c2)
  (lambda (f)
    (contingent
     (f (contingent-info c1) (contingent-info c2))
     (set-union (contingent-premises c1) (contingent-
premises c2)))))

(defhandler binary-map contingency-binary-map contingency? contin-
gency?)
```

Note that the information inside a contingency object may itself be partial, and so perhaps necessitate a recursive call to `binary-map`. This recursion is handled by the given function `f`, and need not the invoked explicitly in handlers for `binary-map`.

A handler for `binary-map` is expected to accept two partial information structures and return a procedure of one argument that will accept a binary function. It is free to apply that function as many or as few times as necessary, and is expected to produce the appropriate result of "mapping" that function over the information in the input partial information structures to produce a new partial information structure, encoding all the appropriate uncertainty from both inputs. The given function `f`, for example as a result of `(nary-mapping generic-switch)`, may return `nothing` even when both of its inputs are non-`nothing`.

The `nary-mapping` wrapper works by repeated use of `binary-map` on arguments of arity greater than two. For unary arguments, `nary-mapping` invokes `binary-map` with a bogus second argument. Therefore, handlers for `binary-map` must handle applications thereof that have your new partial information structure as one argument, and a raw Scheme object as the other (this is a good idea anyway, and saves the trouble of writing handlers for an explicit `unary-map` operation).

8.6 Interoperation with Existing Partial Information Types

A new partial information structure may interact with an existing one in two ways:

- as arguments to merge or to binary propagators
- by containment (of and by)

The first is in general handled by making sure that `merge`, `binary-map`, and all appropriate individual propagator generic operations have methods that can handle any combinations that may arise. Often, the way to deal with two information structures of different but compatible types is to realize that one of them can be seen as an instance of the other type. The coercion machinery (see Section 8.2) allows one to declare when this situation obtains so that `defhandler-coercing` does the right thing. The specific touch points for this are the type testers and coercers of the existing partial information types:

Type	Predicate	Coercer
Nothing	nothing?	--
Raw Scheme object	various	--
Numerical interval	interval?	->interval
Propagator cells	cell?	--
Scheme pairs	pair?	--
Propagator closures	closure?	--
Contingency object	contingent?	->contingent
TMS	tms?	->tms
Contradiction	contradictory?	--

Notes:

- The `nothing` information structure defines methods on `merge` and the propagators that do the right thing for any other object, so does not require any additional effort.
- TMSes automatically coerce to TMS any object that is declared coercible to a raw contingency object.

For example:

```
(declare-coercion interval? ->contingent)
```

allows raw intervals to be seen as TMSes. This has the effect that if a binary operation (either `merge` or a primitive propagator subject to `nary-mapping`) encounter a TMS on one input and an interval on the other, it will coerce the interval to a TMS containing exactly that interval contingent on the empty set of premises, and then operate on those two structures as on TMSes.

The second kind of interoperation is handled by correctly dealing with merge effects (see Section 8.3.1). If you make a new partial information structure that contains others, you must make sure to handle any merge effects that may arise when recursively merging

the partial information your structure contains. If you make a new partial information structure that may need to have effects performed on merge, you should return those as appropriate merge effects in an `effectful` structure, and, if you need to create new kinds of effects in addition to the built-in ones, you should extend the generic operations `execute-effect`, `redundant-effect?`, and `generic-attach-premises` (Section 8.3.1).

9 Making New Primitive Propagators

Almost all definition of new primitive propagators can be handled correctly either by `propagatify` or by `define-propagator-structure` (see Section 7.5). We discuss the lower-level tools first, however.

9.1 Direct Construction from Functions

The fundamental way to make your own primitive propagators is the procedure `function->propagator-constructor`. It takes a Scheme function, and makes a propagator construction procedure out of it that makes a propagator that does the job implemented by that Scheme function. The propagator constructor in question takes one more argument than the original function, the extra argument being the cell into which to write the output. So the result of `function->propagator-constructor` is a diagram-style procedure (complete with (most of) the debugging information, and the constant conversion). The return value of `function->propagator-constructor` can be put into a cell, just same way that a Scheme procedure can be the value of a Scheme variable. For example, you might define:

```
(define-cell p:my-primitive (function->propagator-constructor do-it))
```

where `do-it` is the appropriate Scheme function.

Something important to pay attention to: `function->propagator-constructor` wraps the given function up into a propagator directly, and it is up to the function itself to handle any interesting partial information type that might come out of its argument cells. Notably, `nothing` might show up in the arguments of that function when it is called. Therefore, it may be appropriate the make the function itself generic, and/or wrap it in `nary-mapping`.

For example, let us walk through the implementation of the provided primitive `p:and` in `core/standard-propagators.scm`. First, we make a generic version of the Scheme procedure `boolean/and` to serve as a point of future extension:

```
(define generic-and (make-generic-operator 2 'and boolean/and))
```

Then we wrap that generic procedure with `nary-mapping` to make it process all partial information types that have declared applicative functor behavior, and then we give the result to `function->propagator-constructor` to make a propagator constructor:

```
(define-cell p:and
  (function->propagator-constructor (nary-mapping generic-and)))
```

Another detail to think about is metadata. `function->propagator-constructor` can supply all the metadata that the debugger uses except the name of your function. If your function is generic, the generic machinery already expects a name; otherwise, you need to supply the name yourself, with `(name! your-function 'some-name)`.

9.1.1 Expression Style Variants

Once you've made a diagram-style propagator constructor, you can make a variant that likes to be applied in expression style with `expression-style-variant`. For example, `e:and` is actually defined as:

```
(define-cell e:and (expression-style-variant p:and))
```

9.2 Propagatify

All that wrapping in `nary-mapping`, and naming your propagator functions with `name!`, and calling `expression-style-variant` to convert them to expression-style versions can get tedious. This whole shebang is automated by the `propagatify` macro:

```
(propagatify +)
```

turns into

```
(define generic-+ (make-generic-operator 2 '+ +))
(define-cell p:+
 (function->propagator-constructor (nary-mapping generic-+)))
(define-cell e:+ (expression-style-variant p:+))
```

The easy syntax covers the common case. You can also specify an explicit arity for the generic operation to construct (because sometimes `propagatify` will guess wrong). The above is also equivalent to `(propagatify + 2)`. Sometimes you may want to avoid constructing the generic operation. That can be done also:

```
(propagatify + 'no-generic)
```

becomes

```
(define-cell p:+
 (function->propagator-constructor (nary-mapping +)))
(define-cell e:+ (expression-style-variant p:+))
```

Finally, in the case where you want completely custom handling of partial information, even the `nary-mapping` can be avoided with

(propagatify-raw +)

which becomes

```
(define-cell p:+ (function->propagator-constructor +))
(define-cell e:+ (expression-style-variant p:+))
```

Note that `propagatify` follows the naming convention that the Scheme procedure `foo` becomes a generic procedure named `generic-foo` and then turns into propagators `p:foo` and `e:foo`.

9.3 Compound Cell Carrier Construction

`p:cons` is an interesting propagator, because while it performs the job of a Scheme procedure (to wit, `cons`), it operates directly on the cells that are its arguments, rather than on their contents. Other compound data structures can be made partial information that behaves like pairs using `define-propagator-structure`.

`(define-propagator-structure type constructor accessor ...)` Declares that additional Scheme data structures are partial information like pairs, and defines appropriate propagators that handle them. For example:

```
(define-propagator-structure pair? cons car cdr)
```

defines the propagators `p:pair?`, `e:pair?`, `p:cons`, `e:cons`, `p:car`, and `e:cdr` (and also makes pairs a partial information structure).

Defining `p:cons` to operate on its argument cells constitutes a decision to follow the "carrying cells" rather than the "copying data" strategy from the propagator thesis.

9.4 Fully-manual Low-level Propagator Construction

Finally, when the thing you want your propagator to do is so low-level and interesting that it doesn't even correspond to a Scheme function, there's always the `propagator` procedure. This is the lowest level interface to asking cells to notify a propagator when they change. `propagator` expects a list of cells that your propagator is interested in, and a thunk that implements the job that propagator is supposed to do. The scheduler will execute your thunk from time to time --- the only promise is that it will run at least once after the last time any cell in the supplied neighbor list gains any new information. For example:

```
(define (my-hairy-thing cell1 cell2)
  (propagator (list cell1 cell2)
    (lambda ()
      do-something-presumably-with-cell1-and-cell2)))
```

The `propagator` procedure being the lowest possible level, it has no access to any useful sources of metadata, so you will need to provide yourself any metadata you want to be able to access later. For an example of how this facility is used, see the implementations of `function->propagator-constructor` and `delayed-propagator-constructor` in `core/propagators.scm`.

10 Debugging

There is no stand-alone "propagator debugger"; if something goes wrong, the underlying Scheme debugger is your friend. Some effort has, however, been expended on making your life easier.

In normal operation, Scheme-Propagators keeps track of some metadata about the network that is running. This metadata can be invaluable for debugging propagator networks. The specific data it tries to track is:

- The names (non-unique but semantic) of all the cells and propagators. This is in contrast with the unique but non-semantic object hashes of all the cells and propagators that MIT Scheme tracks anyway.
- Which propagators are connected to which cells.
- Whether the connections are input, output, or both.

To make sure that your network tracks this metadata well, you should use the high level interfaces to making cells, propagators, and propagator constructors when possible (`define-cell`, `let-cells`, `define-propagator`, `propagatify`, etc). Any gaps not filled by use of these interfaces must either be accepted as gaps or be filled by hand.

In order to use the metadata for debugging, you must be able to read it. Inspection procedures using the metadata are provided:

name the name of an object, or the object itself if it is not named

cell? whether something is a cell or not

content the information content of a cell

propagator? whether something is a propagator or not

propagator-inputs the inputs of a propagator (a list of cells)

propagator-outputs the outputs of a propagator (a list of cells)

neighbors the readers of a cell (a list of propagators)

cell-non-readers other propagators somehow associated with a cell (presumably ones that write to it)

cell-connections all propagators around a cell (the append of the neighbors and the non-readers)

You can use these at least somewhat to wander around a network you are debugging. Be advised that cells are represented as Scheme entities and propagators are represented as Scheme procedures, so neither print very nicely at the REPL.

If you find yourself doing something strange that circumvents the usual metadata tracking mechanisms, you can add the desired metadata yourself. All the metadata collection procedures are defined in `core/metadata.scm`; they generally use the `eq-properties` mechanism in `support/eq-properties.scm` to track the metadata, so you can use it to add more. In particular, see the definition of, say, `function->propagator-constructor` or `define-propagator` for examples of how this is done.

11 Miscellany

11.1 Macrology

Sometimes you will need to make something that looks like a macro to Scheme-Propagators. The macro language of Scheme-Propagators is Scheme. For example:

```
(define (my-diagram x y z)
  (p:+ x y z)
  (p:- z y x)
  (p:- z x y))
```

`my-diagram` is a Scheme-Propagators macro that, when given three cells, wires up three arithmetic propagators to them. This simple example of course gains nothing from being a macro rather than a normal compound propagator, but using Scheme as a macro language lets you do more interesting things:

```
(define (require-distinct cells)
  (for-each-distinct-pair
   (lambda (c1 c2)
     (forbid (e:= c1 c2)))
   cells))
```

This `require-distinct` uses a Scheme iterator to perform a repetitive task over a bunch of Scheme-Propagators cells.

This is quite convenient, but sometimes one wants the debugging data provided by `define-propagator`. This is what `define-propagator-syntax` is for. Just change `define` to `define-propagator-syntax`:

```
(define-propagator-syntax (require-distinct cells)
  (for-each-distinct-pair
```

```
(lambda (c1 c2)
  (forbid (e:= c1 c2)))
cells))
```

11.2 Reboots

The procedure `initialize-scheduler` wipes out an existing propagator network and lets you start afresh:

```
build lots of network
...
(initialize-scheduler)
(run) --- nothing happens; no propagators to run!
```

This is the lightest-weight way to restart your Scheme-Propagators session. You can of course also restart the underlying Scheme or just reload Scheme-Propagators if you need to blow away your state.

11.3 Compiling

It turns out that `make-cell` and `cell?` are also MIT Scheme primitives, so if you want to compile your Scheme-Propagators code with the MIT-Scheme compiler, be sure to put

```
(declare (usual-integrations make-cell cell?))
```

at the top of your source files. Also, of course, you need to be suitably careful to make sure that the defined macros are available to the syntaxer when it processes your file. See `support/auto-compilation.scm` for how I do this, and, say, `core/load.scm` for how I use the compiler.

11.4 Scmutils

The Scmutils system built by Gerald Jay Sussman and friends for thinking about physics can be very useful for many purposes. Among other things, it knows about units and dimensions, about symbolic algebra, about solving systems of equations, etc. Scheme-Propagators runs in Scmutils just as well as in MIT Scheme. Some Scheme-Propagators examples that depend upon the ability to manipulate symbolic expressions and solve symbolic systems of equations are included.

11.5 Editing

We edit code in Emacs. You should edit code in Emacs too. Emacs of course has a Scheme mode; nothing more need be said about that here.

If you are going to edit any parenthesized source code in Emacs, Paredit mode is an option you should not overlook.

In addition to the above, we find it very useful to have Emacs highlight and indent some of the Scheme-Propagators macros we have defined the same way as their Scheme analogues; notably `define-propagator` and `let-cells`. Sadly the Emacs Scheme mode does not do this by default, so you need to tweak the Emacs config to do that. The file `support/scm-propagators.el` contains a dump of the relevant portion of my Emacs configuration.

There is at present no Emacs mode for Scheme-Propagators as distinct from Scheme.

11.6 Hacking

Scheme-Propagators is a work in progress. Be aware that we will continue to hack it. Likewise, feel free to hack it as well --- let us know if you invent or implement something interesting. May the Source be with you.

11.7 Arbitrary Choices

Several language design choices affecting the structure of Scheme-Propagators appeared arbitrary at the time they were made.

11.7.1 Default Application and Definition Style

Diagram style application was picked as the default over expression style when applying cells whose contents are not yet known, and for defining compound propagators when the style is not specified more clearly. The main rationale for this decision was an attempt to emphasize the interesting property of Scheme-Propagators and the propagator programming model. The unusual expressive power of fan-in that the propagator model offers can be taken advantage of only if at least some of one's code actually has fan-in, and writing code with fan-in requires diagram style.

11.7.2 Locus of Delayed Construction

There was a choice about where to put the delaying of pieces of propagator network that should be constructed only conditionally. Every recursion traverses an abstraction boundary and a conditional statement every time it goes around. Every recursion must encounter at least one delay barrier every time it goes around, or the construction of the network may generate spurious infinite regresses. But where should that barrier go? There were three plausible alternatives: the first idea was to put the barrier around the application

of recursive compound propagators; the second was to generalize this to put it around the application of all compound propagators; and the third was to capture the bodies of conditional expressions like `p:if` and delay only their construction. During most of the development of Scheme-Propagators, we were using option 1, on the grounds that it sufficed and was easy to implement. Doing this had the effect that in order to actually make a proper recursive propagator, one had to manually "guard", using a hand-crafted pile of `switch` propagators, all the i/o of a recursive call to prevent it from being expanded prematurely. For example, a recursive factorial network written in that style would have looked something like:

```
(define-propagator (p:factorial n n!)
  (let-cells ((done? (e:= n 0)) n-again n!-again)
    (p:conditional-wire (e:not done?) n n-again)
    (p:conditional-wire (e:not done?) n! n!-again)
    (p:* (e:factorial (e:- n-again 1)) n-again n!-again)
    (p:conditional-wire done? 1 n!)))
```

with the added caveat that it would need to be marked as being recursive, so the expansion of the internal factorial would be delayed until it got some information on its boundary (which would be prevented from happening in the base case by the `conditional-wire` propagators). As the system matured, we decided to write a series of macros (`p:when`, `p:unless`, `p:if`, and their expression-style variants) that automated the process of constructing those `conditional-wire` propagators. On making these macros work, we realized that adjusting `p:when` and company to delay their interior would be just as easy as delaying the opening of abstractions. At that point we decided to switch to doing it that way, on the grounds that, since `if` is special in all other computer languages, so it might as well be special here too, and we will leave the operation of abstractions relatively simple. (Partial information makes abstractions complicated enough as it is!) This has the further nice feature that it sidesteps a possible bug with delayed abstractions: if one wanted to create a nullary abstraction, automatic delay of its expansion would presumably not be what one wanted.

11.7.3 Strategy for Compound Data

The decision to go with the carrying cells strategy for compound data felt, while not really arbitrary, at least enough not forced by the rest of the design to be worth some mention. The topic is discussed at length elsewhere, and the available options detailed; so here we will just note why we ended up choosing carrying cells. For a long time, copying data seemed like the right choice, because it avoided spooky "action at a distance"; and merges did not require changing the structure of the network. The downside of copying data, namely the cost of the copying, seemed small enough to ignore. Then we tried to write a program for thinking about electrical circuits.

The specific killer part of the electrical circuits program was that we tried to equip it with observers that built a data structure for every circuit element containing its various parameters and state variables, and for every subcircuit a data structure containing its circuit elements, all the way up. When this program turned out to be horribly slow, we realized that copying data actually produces a quadratic amount of work: every time any circuit variable is updated, the whole chain of communication all the way from resistor to complete breadboard is activated, and they repeat merges of all the compounds that they had accumulated, just to push that one little piece of information all the way to the toplevel observer. In addition, these summary structures turned out to be less useful for debugging than we had hoped, because the updates of the summary structures would be propagator operations just like the main computation, so when the latter would stop for some strange reason, we always had to wonder whether the summaries were up to date.

Carrying cells seemed an appealing solution to both problems. If the summaries carried cells instead of copying data, then updates to those cells would not have to trouble the whole pipe by which the cells were carried, but would just be transmitted through those cells. Also, if we played our cards right, we should have been able to arrange for exactly the cells where the computation was actually happening to be the ones carried all the way to where we could get them from those summary structures, so that the summaries would always be up to date with the underlying computation. But what about the pesky fact that merging structures that carry cells requires side effects on the network? What if that merge is contingent on some premises because the cell-carriers are in some TMS?

That was when merge effects were invented. We realized that merging really should have legitimate side effects on the network, but should package those effects up in manipulable objects that it returns, instead of trying to just execute them. So the question that merge answers was changed from

> What is the least-commitment information structure that captures all the knowledge in these two information structures?

to

> What needs to be done to the network in order to make it reflect the discovery that these two information structures are about the same object?

The latter nicely subsumes the former: a normal merge is just the answer "record in the appropriate cell that the object of interest is described by this information structure". So everything fell into place. The strange `set!` in the most basic definition of the cell is, indeed, an effect that needs to be performed on the network to acknowledge the discovery that two particular information structures are about the same object. The even stranger error signalled on contradiction is an effect too: the thing that needs to be done to the network to reflect the discovery that two completely incompatible information structures describe the same object is to crash. And now both merging cells carried by compound structures and signalling nogoods by TMSes become perfectly reasonable, respectable citizens of the propagator world; and they can interoperate with being contingent by the

enclosing TMS modifying the effects to reflect the context in which they were generated before passing them on up out of its own call to merge.

With that change of perspective on merging, a whole chunk of problems suddenly collapsed. Cells could be merged with a simple "link these two with (conditional) identity propagators". Therefore compound data could be merged by recursively merging their fields, regardless of whether they were carrying cells or other partial information structures. Closures fell into place --- they were just a particular kind of compound data, and merged the way compound data merges. Closures had been a conceptual problem for the copying data view of the world, because closures really felt like they wanted to able to attach their interior propagators to cells closed over from the enclosing lexical environment; but for that, it seemed that the lexical environment would need to be a cell-carrying data structure. But now that carrying cells works, there is no problem. It was on that wave of euphoria that the carrying cells strategy rode into its current place as the standard way to make compound structures in the propagator world. Carrying cells certainly still feels cleaner and nicer than copying data; but it may be that copying data really could still be made to work in all the scenarios where carrying cells is currently winning. We just decided not to pursue that path.

And on the note of copying data being preferable because it preserves locality, maybe `cons` really should be the locality-breaking object.

12 How this supports the goal

We started with the goal of making it easier for people to build systems that are additive. A system should not become so interdependent that it is difficult to extend its behavior to accommodate new requirements. Small changes to the behavior should entail only small changes to the implementation. These are tough goals to achieve.

Systems built on the Propagator Model of computation can approach some of these goals.

A key idea is to allow fan-in, merging partial results. A result may be computed in multiple ways, by complementary processes. There may be multiple ways to partially contribute to a result; these contributions are merged to make better approximations to the desired result. Partial results can be used as a base for further computations, which may further refine known values or partially determine new ones. So we can make effective use of methods that give only part of an answer, depending on other methods to fill in missing details. This ability to absorb redundant and partial computations contributes to additivity: it is easy to add new propagators that implement additional ways to compute any part of the information about a value in a cell.

The Propagator Model is intrinsically parallel. Each component may be thought of as continually polling its neighbor cells and doing what it can to improve the state of knowledge. Any parallel system will have race conditions, but the paradigm of monotonically accumulating information makes them irrelevant to the final results of a computation.

A propagator network can incorporate redundant ways to compute each result. These can contribute to integrity and resiliency: computations can proceed along multiple variant paths and invariants can be cross-checked to assure integrity.

Dependency tracking and truth maintenance contribute to additivity in a different way. If you want to add a new pile of stuff, you don't need to worry too much about whether or not it will be compatible with the old: just make it contingent on a fresh premise. If the addition turns out to conflict with what was already there, it (or the offending old thing) can be ignored, locally and dynamically, by retracting a premise. Dependency tracking also decreases the amount each module needs to know about its interlocutors; for example, instead of having to guess which square root a client wants, the `sqrt` routine can return both of them, contingent on different premises, and let the eventual users decide which ones they wanted.

Dependency tracking is natural in the propagator model. By contrast with a traditional computational model, the propagator model has no defined order of computation, except as dictated by the data flow. Thus, no spurious dependencies arise from the ordering of operations, making the real dependencies easier to track.

A propagator program is analogous to an electrical circuit diagram, whereas a program written for a more traditional model is more like a system diagram: the intellectual viewpoint of the Propagator Model of computation is not the composition of functions, as in traditional models, but is rather the construction of mechanisms. The difference is profound:

1. circuit models are multidirectional; system diagrams compute from inputs to outputs.
2. circuit models are abstractions of physics; system diagrams are abstractions of process.

The circuit diagram viewpoint gives us powerful ways to think. We can modify a circuit diagram by clipping out a part, or by installing a different version. We can temporarily carry information from one place to another using a clip lead. We have lots of places to connect various kinds of meters for monitoring and debugging.

Now for something completely different! Scheme-Propagators is built with dynamically-extensible generic operations. The pervasive `merge` operation, as well as all the primitive propagators, are generic, and this makes it easier for us to add new forms of partial information. Adding new forms of partial information is a way to extend the capabilities of the propagation infrastructure to novel circumstances. With appropriate foresight, new partial information structures can interoperate with old, and additively extend the capabilities of existing systems --- a way to teach old dogs new tricks. Of course, such power is dangerous: if a network that depends on commutativity of multiplication of numbers meets an extension to square matrices, we will get wrong answers. But then, cross-checking across complementary methods, together with dependency tracking, simplifies the task of debugging such errors.

References

[1] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[2] Alexey Radul. *Propagation Networks: A Flexible and Expressive Substrate for Computation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 2009. http://hdl.handle.net/1721.1/49525.

[3] Alexey Radul and Gerald Jay Sussman. The Art of the Propagator. CSAIL Tech Report MIT-CSAIL-TR-2009-002, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2009. http://hdl.handle.net/1721.1/44215.