Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2009-053 November 3, 2009

Propagation Networks: A Flexible and Expressive Substrate for Computation

Alexey Radul

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

Propagation Networks:
A Flexible and Expressive Substrate for Computation

by

Alexey Andreyevich Radul

B.S., Massachusetts Institute of Technology (2003)
M.Eng., Massachusetts Institute of Technology (2005)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHVSETTS INSTITVTE OF TECHNOLOGY

September 2009

Author . . . . . .
Department of Electrical Engineering and Computer Science
September 4, 2009

Certified by . . . . . .
Gerald Jay Sussman
Panasonic (Matsushita) Professor of Electrical Engineering
Thesis Supervisor

Accepted by . . . . . .
Professor Terry P. Orlando
Chair, Department Committee on Graduate Students

Propagation Networks:
A Flexible and Expressive Substrate for Computation

by

Alexey Andreyevich Radul

Submitted to the Department of Electrical Engineering and Computer Science
on September 4, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

In this dissertation I propose a shift in the foundations of computation. Modern programming systems are not expressive enough. The traditional image of a single computer that has global effects on a large memory is too restrictive. The propagation paradigm replaces this with computing by networks of local, independent, stateless machines interconnected with stateful storage cells. In so doing, it offers great flexibility and expressive power, and has therefore been much studied, but has not yet been tamed for general-purpose computation. The novel insight that should finally permit computing with general-purpose propagation is that a cell should not be seen as storing a value, but as accumulating *information about* a value.

Various forms of the general idea of propagation have been used with great success for various special purposes; perhaps the most immediate example is constraint propagation in constraint satisfaction systems. This success is evidence both that traditional linear computation is not expressive enough, and that propagation is more expressive. These special-purpose systems, however, are all complex and all different, and neither compose well, nor interoperate well, nor generalize well. A foundational layer is missing.

I present in this dissertation the design and implementation of a prototype general-purpose propagation system. I argue that the structure of the prototype follows from the overarching principle of computing by propagation and of storage by accumulating information—there are no important arbitrary decisions. I illustrate on several worked examples how the resulting organization supports arbitrary computation; recovers the expressivity benefits that have been derived from special-purpose propagation systems in a single general-purpose framework, allowing them to compose and interoperate; and offers further expressive power beyond what we have known in the past. I reflect on the new light the propagation perspective sheds on the deep nature of computation.

Thesis Supervisor: Gerald Jay Sussman
Title: Panasonic (Matsushita) Professor of Electrical Engineering

To my wife Rebecca,
without whose patient critique my thesis would be far worse,
without whose enduring support it would not even be possible, and
without whose unfailing love nothing else I do would be possible either.

Acknowledgments

It is a great pleasure to thank the people who helped me with my graduate career.

First, my supervisor Gerry Sussman. He has been a fantastic friend and mentor to me for nigh on eight years. I first met Gerry when I took his class on classical mechanics as a sophomore, and in retrospect I have been in orbit, captured by his intellectual gravity, ever since. Admittedly, it must have been a high eccentricity orbit, because my distance from the center has varied considerably over the years; nonetheless, it was with Gerry that I began the adventure that is graduate school, and it is with Gerry that I end it. It has been fantastic, and I cannot even begin to enumerate what I owe him.

My thesis committee, Guy Steele Jr., Patrick Winston, and Tom Knight, has also been invaluable. I am particularly indebted to Guy for his meticulous and insightful comments as my dissertation neared completion.

Not only graduate school but life in general would have been completely different without Gregory Marton's unfailing friendship. Through him I have learned as much about kindness and generosity as about research and science.

Arthur Gleckler taught me how to engineer software. My internships with him have shaped the way I think about the nature of making programs continue to behave the way I want them to, and his continued friendship outside of work has helped me make *me* continue to behave the way I want me to.

No set of acknowledgments can be complete without my mother, Tanya Khovanova, and my brother, Sergei Bernstein. They have helped me, influenced me, and supported me beyond measure.

There are of course many more people whose influence on me deserves eloquence than I have here ink or paper to record it. I want to thank Alan Bawden, Jake Beal, Larry, Holly and Taylor Campbell, Mitchell Charity, Joyce Chen, Sue Felshin, Chris Hanson, Sagar Indurkhya, Boris Katz, Norm Margolus, Julie Norville, Jim Psota, François René Rideau, Jillian Saucier, and Julie Sussman.

Finally, pride of last place, my longtime friend and now wife, Rebecca. Gratitude does not cover what you have done for me. The chance to dedicate this dissertation to you is but a small offering in comparison to how I have flourished under the light of your friendship and your love.

My graduate career in general, and this work in particular, have been sponsored in part by a National Science Foundation Graduate Research Fellowship, by the Disruptive Technology Office as part of the AQUAINT Phase 3 research program, by the Massachusetts Institute of Technology, by Google, Inc., and by the National Science Foundation Cybertrust (05-518) program.

Contents

List of Figures

Chapter 1

Time for a Revolution

REVOLUTION is at hand. The revolution everyone talks about is, of course, the parallel hardware revolution; less noticed but maybe more important, a paradigm shift is also brewing in the structure of programming languages. Perhaps spurred by changes in hardware architecture, we are reaching away from the old, strictly time-bound expression-evaluation paradigm that has carried us so far, and looking for new means of expressiveness not constrained by over-rigid notions of computational time.

This dissertation introduces, explains, illustrates, and implements *general purpose propagation*. Propagation is an established computational idea, but using it to structure general-purpose programming can bring a dramatic new increase in expressiveness. The key idea of propagating *mergeable, partial information* allows propagation to be used for general-purpose computation. The second key idea of making the merging mechanism *generic* offers a new kind of modularity and composability. Finally a more technical contribution comes in the form of a structure for carrying *dependencies*, and a presentation of the fantastic consequences of combining dependencies with propagation.

The structure of the dissertation is as follows: after a more complete introduction in the present chapter, Chapter 2 lays out the guiding design principles of a propagation system based on these ideas, and Chapter 3 implements it. Chapter 4 develops the dependency machinery and illustrates some of its direct consequences. Then Chapter 5 studies the expressive power offered by propagation, in particular with dependency tracking, and compares propagation to other expressive programming techniques. Chapter 6 looks at building a complete programming system out of the basic fabric of computation worked out in the rest of the dissertation, and finally Chapter 7 reflects on computing as seen through the propagation lens.

1.1 Expression Evaluation has been Wonderful

Evaluation of expressions is the fundamental paradigm underlying almost all modern programming languages. A program is a series of expressions, evaluated either for their value or for their effect. Each expression is of some kind (constant, func-

tion call, definition, etc.), and is either atomic or has subexpressions. Evaluation proceeds by structural recursion on the expressions, evaluating the atomic expressions directly, and evaluating the compound expressions by appropriately combining the values of their subexpressions.

This paradigm is exemplified perhaps most clearly by `eval`/`apply` interpreters for Lisp dialects like Scheme [Abelson et al., 1996]. `Eval` interprets each expression, and recurs on the subexpressions; `apply` handles primitives and abstracted compounds (which involves recursive calls to `eval` on the bodies of those abstractions). The variations on the theme are endless—some are interpreted, some are compiled, some have type systems, some lack garbage collectors, and the essential paradigm is clearer in some than in others—but this fundamental recursion runs through nearly everything we think of as programming.

In order that it be clearer which way I intend to move forward from expressions, let me take a paragraph to look backward. Just as most high-level languages follow the expression paradigm, assembly languages follow another paradigm. The assembly language paradigm is a loop that executes instructions in sequence (and some instructions interrupt the sequence and cause the executor to jump somewhere else). On the one hand, this is easier to think about and implement, to the point that this is what we collectively chose to implement in our hardware. On the other hand, the only difference between instructions and expressions is that instructions are all atomic—there are no arbitrary combinations of instructions acting on the results of subinstructions. So expressions generalize instructions, and are therefore a more flexible and expressive paradigm; and the investment in learning to program with expressions rather than instructions pays immeasurable dividends in all aspects of computing.[1]

1.2 But we Want More

A wonderful engine of practical and theoretical progress though the expression evaluation paradigm has been, computer science as a field is reaching for even more. The constraint satisfaction community (e.g., [Tack, 2009]) seeks to solve simultaneous systems of constraints; this involves thinking about different constraints and different possible values as needed. The logic programming community (e.g., [Nilsson and Małuszyński, 1995]) promises automated reasoning over declared logical relations; this involves deducing in whatever order is best. The functional reactive programming community (e.g., [Cooper, 2008]) offers elegant responsiveness to changing inputs, alleviating worries about what piece of state to update in response to what action. And we all worry about the impending "concurrency problem".

[1] Except performance on microbenchmarks!

1.3 We Want More Freedom from Time

Why? Why is evaluation of expressions, which has served us so faithfully for half a century, not good enough for all these people? The fundamental problem is time. Time in evaluation-land is fundamentally constrained. The successful evaluation of an individual expression, or, equivalently, the successful production of an individual value, inescapably marks a point in time. The work done to produce that value came *before*; the work that will be done using that value comes *after*.[2] That inevitable limitation manifests as the sequencing of operations by the evaluator. Since the evaluator works by structural recursion on the expressions that comprise a program, the structure of the program text constrains the flow of time through that program. Thus, while a program must serve as the description of *what* we want accomplished, its shape excessively constrains the process, and therefore *how* the computer accomplishes it.

Each thing I alluded to in Section 1.2 is either a worry about or an escape from the rigidity of time. The "concurrency problem" worries about how to live with many parallel threads of time instead of just the one that the normal `eval`/`apply` loop provides. Constraint solvers examine their constraints in whatever order seems auspicious, and as many times as needed. Logic programs consider their clauses in an order determined by their search, and are free to change their mind and backtrack. Functional reactive systems react to whatever event happens next. In each case, time doesn't want to be a simple line: Maybe it's several communicating threads, maybe it needs to bounce around the constraints or deductions at will, maybe it needs to start and stop in response to user actions, but in each case the orderly progression of `eval`/`apply` time is a yoke.[3]

1.4 Propagation Promises Liberty

Fortunately, there is a common theme in all these efforts to escape temporal tyranny. The commonality is to organize computation as a network of interconnected machines of some kind, each of which is free to run when it pleases, propagating information around the network as proves possible. The consequence of this freedom is that the structure of the aggregate does not impose an order of time. Instead

[2]Lazy evaluation is based on the observation that merely passing some value as an argument to a function, or merely storing a reference to that value in a data structure, does not constitute using it until the program does something sufficiently primitive with that value (like trying to print it to the screen, or trying to hand it to the floating point unit so it can be added to 1.7320508075688772). This relaxes Time's stranglehold somewhat, but evidently not enough. We will return to the relationship between propagation and normal and applicative order evaluation in Section 6.2.4.

[3]The fact that separate subexpressions of the same expression may perhaps be evaluated in an unspecified order [Kelsey et al., 1998], or even in parallel [Allen et al., 2005], means that time is not entirely linear, but it is still at least piecewise-linear, with specific (relatively uncommon) points of uncertainty or branching. In fact, the very existence of `call-with-current-continuation` is evidence that something is amiss, because that procedure promises to return a reification of the *entire future!*

the implementation, be it a constraint solver, or a logic programming system, or a functional reactive system, or what have you is free to attend to each conceptual machine as it pleases, and allow the order of operations to be determined by the needs of the *solution* of the problem at hand, rather then the structure of the problem's *description*.

Unfortunately, the current state of the art of this organizational paradigm is somewhere between a vague generalization and an informal design pattern. On the one hand, every system that needs not to be a slave to time does something philosophically similar, but on the other hand, they all constitute different detailed manifestations of that common idea. What's worse, each manifestation is tailored to its specific purpose, each is complex, and none of them compose or interoperate. We have an archipelago of special-purpose propagation systems, drowning in a sea of restrictive linear time.

In this dissertation I demonstrate a *general-purpose* propagation system. I show that propagation subsumes evaluation (of expressions) the same way that evaluation subsumes execution (of instructions). I show that with appropriate choices of what to propagate, the general purpose system can be specialized to all the purposes listed above, and then some. I show how basing these specializations upon a common infrastructure allows those systems to compose, producing novel effects. Finally, I will show how the propagation lens casts a new light on several major themes of computer science.

I believe that general-purpose propagation offers an opportunity of revolutionary magnitude. The move from instructions to expressions led to an immense gain in the expressive power of programming languages, and therefore in understanding of computers and productivity when building computing artifacts. The move from expressions to propagators is the next step in that path. This dissertation merely lifts one foot—who knows what wonders lie just around the corner?

Chapter 2

Design Principles

EXPOUNDING on the advantages to be gained from building a propagation infrastructure is all good and well, but let us proceed now to consider how one should be built. The purpose of the present chapter is to present and motivate several design principles. These principles guide the detailed choices of the propagation prototype presented in Chapter 3 and used and discussed in the remainder of this dissertation; they furthermore constitute my recommendation for how general-purpose propagation systems should be built, once granted that the aim is to gain the expressive power that flows from freedom from Time.

While historically these design principles emerged over time through experiments with one variation on the propagation theme after another, now that they are discovered they form a coherent whole, and flow, in retrospect, from our stated purpose. I can therefore afford to present them on their own merits, with the promise that they turned out to actually work as my evidence of their validity, rather than documenting the tortuous intellectual path, full as it was of unexplored side avenues, that led here. My purpose in so doing is the greatest clarity of exposition, rather than any claim that we were able to get away with building the prototype in as Waterfall [Royce, 1970] a style as it is being presented.

2.1 Propagators are Asynchronous, Autonomous, and Stateless

We want to break free of the tyranny of linear time by arranging computation as a network of autonomous but interconnected machines. How should we architect such a network? What do we want to say about the nature of the machines?

To be explicit about the events in the system, let us say that the machines are not connected to each other directly, but through shared locations that can remember things which interested machines can read and write. Such locations are traditionally called **cells**. As the machines communicate with each other and perform their various computations, information will propagate through the cells of the network. The machines are for this reason traditionally called **propagators**. See, for example, Figure 2-1.

The purpose of all this is to avoid too early a commitment to the timing of

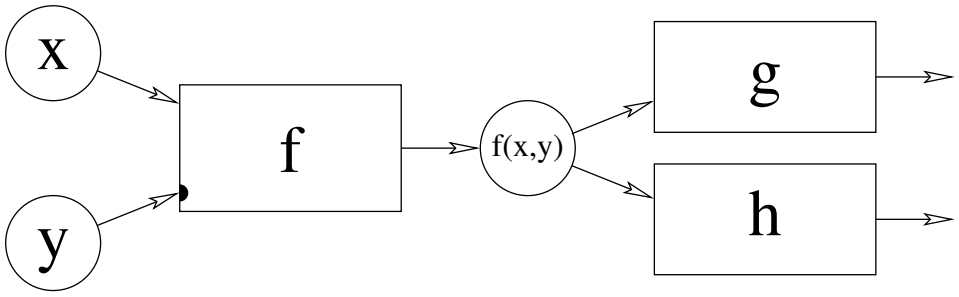


Figure 2-1: A sample fragment of a propagator network. Information about x and y has just propagated through f; now g and h have an input they can work on. The black semicircle marks y as the second argument of the (presumably non-commutative) function f.

events in our computers, so that the computers can decide for themselves in what order to do things, guided by what they discover when performing the tasks we assign to them. We should consequently not build into our infrastructure any timing commitments we can possibly avoid.

Let us therefore posit that our propagators are of themselves memoryless. The idea of memory is inextricably entwined with the idea of time, so let us push all memory out of our computing propagators, and into the cells, who are dedicated to dealing with memory anyway. We lose no generality by doing this, because we can always equip any machine that needs to appear stateful with a private cell to hold its state. With this commitment, the actions of any propagator are determined entirely by the contents of the cells it examines, whereas the contents of any cell are determined entirely by the actions of the propagators that write to it.[1]

Let us likewise posit that our propagators are autonomous, asynchronous, and always on—always ready to perform their respective computations. This way, there is no notion of time embedded in questions of which device might do something when, for they are all always free to do what they wish. Since the cells are the system's memory, it is the cells' responsibility to provide interlocks appropriately to prevent any nasty memory corruption problems, and to prevent propagators from seeing the contents of an individual cell in an inconsistent state.[2]

This dissertation takes these ground rules as axiomatic. Doubtless there are other philosophical foundations on which to rest an exploration of computer ar-

[1]Of course, not all of the computing world can be so nicely arranged. There are keyboards, hard disks, displays, network ports, robot sensors and actuators, and many other things that don't look like propagators that read only their cells and cells that receive input only from their propagators. Something else will have to happen at the boundaries of this system. What that something else should be, exactly, is wide open for future research; but we will yet revisit this topic, in a technical way in Section 6.6, and in a broader philosophical way in Section 7.3.

[2]It is important that these are perfectly local locks. We are not trying to enforce any global consistency notions—those will have to be emergent properties of the network. But we do want local consistency from the lowest-level primitives in the system.

chitectures that break free of an excessive dependence on time, but I have chosen to study these. I hope to convince you in the rest of this text that these principles cause no uncomfortable restrictions, and allow systems that are both elegant and expressive—more expressive, as promised, than those following the expression evaluation strategy.

2.2 We Simulate the Network until Quiescence

One might say that the very purpose of programming is to avoid having to physically construct the machine that is notionally performing one's computation, and instead merely to simulate that machine on just one physical universal computer.

Let us therefore posit further that as our objective is expressive programming, we are explicitly interested in computation by networks of *simulated* autonomous interconnected machines. We therefore will not restrict our attention only to propagators that could be implemented directly as physical devices.

Because we are after independence from time, we will not[3] use these devices to engineer systems with a particular evolution or sequence of states. Rather, our aim is to build systems whose steady state, after the transient consequences of adding the inputs have died down, is predictable; and to express whatever computations we want in the mapping between those inputs and those steady states.

The objective of our simulation of such a system is then the faithful computation of such a steady, or **quiescent**,[4] if you will, state, rather than the faithful reproduction of the intermediate states that lead up to it. As a consequence of that, of course, the simulation is very interested in detecting when such a quiescent state obtains.

We will simulate our primitive propagators with corresponding Scheme programs. We will presume that a single run of such a program suffices to model everything such a propagator will do on given inputs (excluding any possible consequences of its modifying its own inputs). A propagator can therefore be assumed to be quiescent if its inputs have not changed after the last time it started to run. Detecting the quiescence of a network, or a region thereof, thus reduces to detecting whether or not the contents of cells have changed; it is natural to make the cells responsible for this.

---

[3] By and large, but see Section 7.3.

[4] By "quiescent" I mean "no propagator will add any more information to any cell without further input from outside the network." This is only the simplest idea of "steady state." One could also imagine various propagator networks that asymptotically approach a certain attractor state but never fully reach it. Arranging such a network might also be useful, but quiescence will suffice to make an interesting enough system for the purpose of this dissertation. This text will therefore satisfy itself with quiescence, except to touch on tantalizing alternatives in Section 6.4.

2.3 Cells Accumulate Information

I am accepting the above propagator design principles because they worked (well enough to produce this dissertation, anyway); I am taking them as axioms because I have only the vague generalizations provided above as evidence that they are the only principles that would have worked. The first contribution of this work is that they *entail* a design principle applicable to cells, which in fact constitutes a shift of perspective from how such systems have been built previously.

The unquestioned assumption of previous generations of propagation systems has been that cells hold values—complete, finished, fully computed answers. This is a natural enough idea: a variable in a normal programming language is a thing that holds a value, so why should a cell in a propagator system be any different? In fact, I suspect that this fundamental, unquestioned idea is the underlying cause of the Babel of different, complex, incompatible special-purpose propagation systems that exist today.

I propose, in contrast, that we should think of a cell as a thing that accumulates *information about* a value. The information can perhaps be incomplete: some propagator may tell a cell something that is on the one hand useful but on the other hand does not determine that cell's value unconditionally. If so, the cell should readily offer that knowledge to other propagators that may ask, and stand ready to incorporate any improvements or refinements of that information, from any source. Instead of always demanding finished products of our computations, we are willing to accept "works in progress," which are both potentially already useful but also open to further improvement.

Accumulating information is a generalization of the idea of holding values, because a value can always be interpreted as the information that says "I know exactly what this value is, and it is x;" and the absence of a value can be interpreted as information that says "I know absolutely nothing about what this value is." This is not an abstract generalization: it solves the architectural problems that have plagued propagation systems in the past, and eliminates the core reason for their variations and complexities.

The particular knowledge representation can and should vary from problem to problem. Varying them is a way to adapt the propagation system to exhibit different behaviors and serve different purposes; but the contract for knowledge representations defines a module boundary separating the complexity of each particular representation from the core propagation substrate. Making the different information structures interact with each other usefully can still be a hard problem, but this architecture at least gives them a common medium they can share, making composition and compatibility that much easier.

The basic philosophical reason why cells must accumulate incrementally refinable information is that computation in propagation networks is essentially multidirectional. Since several propagators can point into the same cell, there is no reason to demand that just one of them be responsible for producing the whole of that cell's

contents. In fact, the whole point of building propagation systems is to capitalize on the possibility of multidirectionality—and to do that, each cell must be able to accept and merge all the contributions of all the propagators that might write to it. But don't take my word for it: let's have a concrete look at the consequences of assuming that cells store values.

2.3.1 Why don't cells store values?

Why do we need to do this accumulating information thing? Does the "cells store values" architecture really lead to inevitable trouble? What goes wrong?

Things go wrong when one starts trying to implement cells that hold values, and discovers oneself doing kludgerous things. Let us imagine, in our mind's eye, writing a procedure for adding a value to a cell:

```
(define (add-value cell value)
  (if (empty? cell)
      (begin
        (set-cell-value! cell value)
        (note-cell-change cell))
      ???))
```

If the cell is empty, then we add the value, and notify the simulator that this cell's value changed, so that it knows that the propagators that read that cell have not quiesced yet. But what should we do if we are adding a value to a cell that already has a value?

Our unquestioned assumption doesn't help here. The rule is supposed to be "a cell holds a value," but if there are competing values, the rule doesn't say anything about which value should win or what should happen. So, in the absence of principled guidance, we have to make an arbitrary choice.

Arbitrary choices are dangerous: one might unwittingly choose an option that specializes the system to one's particular purpose, and others might make the same choice differently, leading to incompatibilities. But we are courageous programming language designers, so that danger does not stop us.

Since we're writing a piece of code that embodies an arbitrary choice, we would like that code to be as simple as possible. If something sufficiently simple works and does everything we want, that is a powerful affirmation—simplicity and elegance go hand in hand with correctness and generality [Ockham, 1340]. Perhaps our principle even dictated that solution after all, but we were just not smart enough to see it at first.

2.3.2 Because trying to store values causes trouble

So what is the simplest way we can fill in those `???` question marks? The three simplest options don't involve examining the present or the incoming value. They are:

A. Drop the second value on the floor,

B. Overwrite the first value, or

C. Forbid the event (signal an error).

Let us examine each in turn.

Option A

What can we do with a second value without looking at it? Option A: Drop it on the floor. The code is simple indeed; we replace the question marks with a noop:

```
(define (add-value cell value)
  (if (empty? cell)
      (begin
        (set-cell-value! cell value)
        (note-cell-change cell))
      'ok))
```

Sadly, this doesn't work. This strategy causes practical trouble if more than one propagator might write a value to a cell, and if those propagators might disagree. For example, we might have liked to build a constraint system where the constraints were implemented by attaching mutually-inverse propagators to the same cells. But if we did that, and wired up a single cell to be part of more than one constraint (which is presumably the point), our constraint system might fail to enforce the constraints. Take, for instance, the sum constraint shown in Figure 2-2. Suppose the one, two, and six were added to the cells of the constraint before any of the constraint's propagators got a chance to run. Then when the constraint's propagators do run, the cells will drop the competing values those propagators will try to put there, and the constraint will go unenforced.

Now, we could try to solve this problem of constraint non-enforcement somewhere other than the cells. For instance, we could try to make sure that the network never sees an inconsistency like this by always running the whole thing to quiescence between supplied inputs. That's a terrible idea, because it imposes horrendous global constraints on the timings of things that happen; and what if the conflicting inputs are produced by some yet other program, rather than a human user? Where then are the boundaries where contradictions can be detected?

Or, alternately, we could try to build constraints that actually constrain with a network construction discipline such that whenever two propagators could disagree, there was some third propagator around to look at both their outputs and compare them, and arbitrate between them. But imposing such a discipline is also complexity added to the fundamental infrastructure, and this extra complexity is far greater than just making the cells a little smarter. So that's not a good way out either.

The philosophical reason the "Drop the second value on the floor" strategy is not a good idea is that now cells, and therefore the network taken as a whole, can ignore what propagators do under some circumstances (namely the circumstance that the cell already has a value). This violates the "always-on machines" vision,

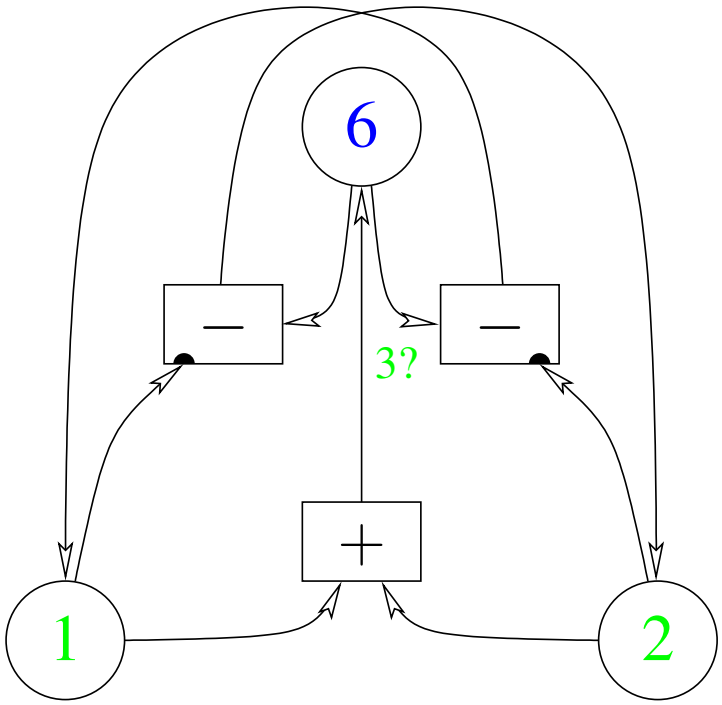


Figure 2-2: If cells drop second values, “constraints” don’t constrain.

and thus causes endless trouble.

Option B

What else can we do with a second value without looking at it? Option B: Always overwrite the first value. The code to implement that is even simpler:

```
(define (add-value cell value)
  ;; Don't even have the if
  (set-cell-value! cell value)
  (note-cell-change cell))
```

Unfortunately, always overwriting is no good either. Consider, for example, a loop in the network, like in Figure 2-3. Even though there is nothing more to discover after the third cell’s value has been filled in, the network continues to appear active, writing limitless repetitions of the same thing to the various cells. If the cells do not notice that the new incoming values are redundant, no one will be able to tell that the system is stable and the simulator should stop running these computations.

Previous efforts have considered solving this artificial instability in other places, for instance by adjusting the simulator not to rerun propagators that go “backward”. This amounts to predicting all the loops in the network structure, and trying to arrange that they are not repeated excessively. Needless to say that approach doesn’t scale—there’s no good general way to identify all loops in advance, and even if there were, there still wouldn’t be any good general way to know which way is

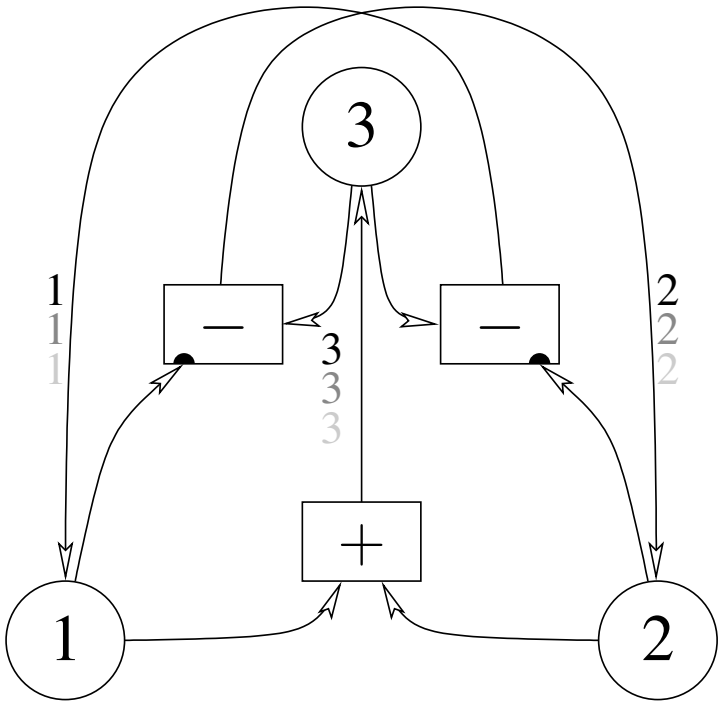


Figure 2-3: If cells always overwrite, loops buzz.

"backward". Trying to play a game like that also introduces additional unnecessary complexity into the scheduler, and therefore into the system's notion of time. And even if one could get the network to stabilize if the propagators agree, there would still be oscillations or race conditions if the propagators disagreed.

The fundamental reason overwriting the old value is not a good idea is that it reintroduces time, because now something *changes*: there was an old value that used to be there in the past, but when the new value gets written, the old value is *gone*. So now there's time, and you start worrying about the scheduler, and you've defeated the purpose.

Option C

What else can we do with a second value without looking at it? Option C: Forbid it. This code is also simple,

```
(define (add-value cell value)
  (if (empty? cell)
      (begin
        (set-cell-value! cell value)
        (note-cell-change cell))
      (error "Contradiction")))
```

but this is not a good idea either, because it just pushes the problem somewhere else. Now you either have to play with the scheduler again, or all the propagators have to do something like option A or option B, or some other such loss happens.

If the cells are the repositories of state, then dealing with multiple requests to store state is their job.

Out of options

And that's it. Since none of that works, you have to look at the values to decide what to do with them. This is a bit scary: cells are supposed to be value-storing machines, but it seems they can't just accept their values as sealed boxes—they have to look inside. But we are courageous programming language designers, so that little fear does not stop us. What, then, is the next best thing to do? We have an equality test.

```
(define (add-value cell value)
  (if (empty? cell)
      (begin
        (set-cell-value! cell value)
        (note-cell-change cell))
      (if (equal? value (current-value cell))
          'ok
          (error "Contradiction"))))
```

Well, that looks better: the buzzes stop, and the constraints still constrain. Unfortunately, we have now baked in the notion of equality. This is not good; our propagation infrastructure is now assuming something about the data it is pushing around: that the right way to resolve conflicts is with an equality test. We have made another arbitrary choice.

That choice turns out to be wrong.[5] Suppose we want to augment our network with dependency tracking, as [Steele Jr., 1980] has and as we will in Chapter 4. Then the cells should hold a value augmented with its justification. But what do we do if we get two identical values with different justifications? Surely that is not really a contradiction, but our equality test betrays us. So we need something more elaborate.

Or suppose instead we want to use our network for constraint satisfaction by domain reduction, as [Tack, 2009] has and as we will in Section 5.3. Then the cells will hold some representation of the domain of some variable. So what do we do if a domain gets narrowed down? We don't want to compare the domains for equality, that is certain. Again we need something more elaborate. And this time it's something different.

Or suppose instead we want to use our network for functional reactive programming, as [Cooper, 2008] has and as we will in Section 5.5. Then the cells will contain values that may be stale or fresh. So what do we do if some cell contains a stale value and a different fresh value shows up? Again comparing them for equality is the wrong thing. Again we need something more elaborate. And this time it's something yet different again.

---

[5] As all arbitrary choices eventually do, if one varies the purpose against which they are evaluated.

Having opened the Pandora's box [Athanassakis, 1983] of arbitrary decisions, we are faced with more and more of them, each tugging in different directions. Whatever each implementor's actual purpose is, each propagation system will end up being specialized to that purpose, and none of them will compose or interoperate well with one another. Starting with cells that store values leads into a maze of differences and incompatibilities.

2.3.3 And accumulating information is better

We solve this pile of problems at a stroke by changing the mindset. Instead of thinking of a cell as an object that stores a *value*, think of a cell as an object that stores *everything you know about a value*. Suddenly, everything falls into place. "Empty" cells stop being special: that's just a cell that doesn't know anything yet. "Second values" stop causing problems: that's just more information. The grotty equality test becomes a test for redundancy of new information, and the contradiction from different values becomes a contradiction from completely irreconcilable information. We will see later that dependency tracking, and many many other things, fall neatly under the accumulating information umbrella; the point of this dissertation is that propagating and accumulating partial information is enough for a complete, expressive, and flexible model of computation.

It is important that this be *accumulating* partial information: it must never be lost or replaced. A cell's promise to remember everything it was ever told[6] is crucial to gaining freedom from time. After all, if a cell could forget something that mattered, then all its neighbors would have to run before it forgot. But they can only remember things by writing them to other cells, which in turn can forget, so *their* neighbors must read them within a certain amount of time... and there is Time again. So cells must not forget.

The idea of accumulating partial information is a module boundary. It separates the essential core of propagation from the various complexities of dealing with the things being propagated. If one wants to propagate numbers, one will still need to test them for equality to detect collisions. If one wants to track dependencies, one will still need to wrangle truth maintenance systems. If one wants to do something else, one will still need to write the code that does it. But now these problems are separated from the core of propagation, and from each other, so they can be built additively and speak a common propagation language. That language can allow them to be written so as to compose and interoperate—it is a more expressive foundation on which we can create a higher level of abstraction.

[6]Except when the system can prove that particular things will never matter again. This is analogous to garbage collection in modern memory-managed systems.

Chapter 3

Core Implementation

BY CONSIDERING in some depth the overarching design principles we will follow in building our propagation infrastructure, we have prepared ourselves to actually build one. It is built in software, in fact as a prototype embedded in MIT/GNU Scheme [Hanson et al., 2005], because that's the easiest way for me to think about things like this. The construction of this prototype was too difficult an exercise in exploration for me to have undertaken the corresponding exercise in engineering; it will consequently win no prizes for speed, but I expect an implementation of reasonable alacrity to be a Simple Matter of Programming (with all the attendant technical but not ideological challenges).

This chapter motivates a particular detailed core propagation infrastructure with a series of examples, and shows the interesting portions of its implementation with a series of Scheme code fragments (the remaining details are recorded in Appendix A). We will reflect at the end of the chapter on why the result is more flexible than the evaluation of expressions, and the next chapter will then expand on that with the first of a series of illustrations of the greater expressive power gained by treating propagation as fundamental to computing.

The presentation of this prototype is structured as a gradual buildup. We begin with the simplest possible thing in Section 3.1; we observe that even this is interesting in Section 3.2; we vary the simple thing in Section 3.3, finding that our design principles from Chapter 2 do not lead us astray; and we generalize that variation in Section 3.4 to reach the promised generality and flexibility. Remember as you read that this chapter presents the basic fabric of computation that I am proposing; Chapters 4 and 5 are about why this fabric is interesting, and it is only in Chapter 6 that I will get down to the business of sewing the Emperor's [Andersen, 1837] new clothes out of it.

3.1 Numbers are Easy to Propagate

To recap, our computational model is a network of autonomous machines, each continuously examining its inputs and producing outputs when possible. The inputs and outputs of a machine are shared with other machines so that the outputs

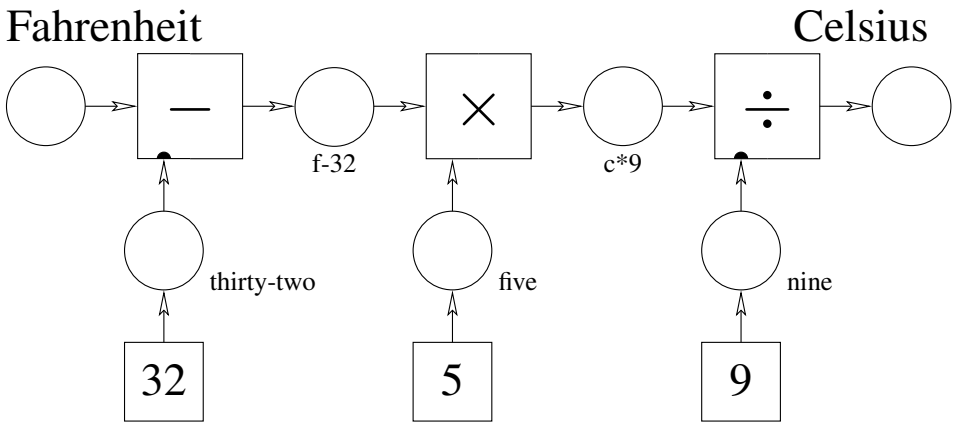


Figure 3-1: A Fahrenheit to Celsius converter network

of one machine can be used as the inputs for another. The shared communication mechanism is called a **cell** and the machines that they interconnect are called **propagators**. As a consequence of this viewpoint computational mechanisms are naturally seen as wiring diagrams.

Since we have been talking in abstract terms so far, you must be yearning for a concrete example. Let's start with something simple: the classic [Steele Jr., 1980] problem of converting temperatures from Fahrenheit into Celsius. There won't be any stunning insights from this section, but we will lay the foundation on which everything else can be built.

If we have a Fahrenheit temperature in cell f, we can put the corresponding Celsius temperature in cell c by the formula $c = (f - 32) * 5/9$. The wiring diagram of a network for computing this formula is shown in Figure 3-1. We can describe this wiring diagram in a conventional programming language (Scheme) by the following ugly code:

```
(define (fahrenheit->celsius f c)
  (let ((thirty-two (make-cell))
        (f-32 (make-cell))
        (five (make-cell))
        (c*9 (make-cell))
        (nine (make-cell)))
    ((constant 32) thirty-two)
    ((constant 5) five)
    ((constant 9) nine)
    (subtractor f thirty-two f-32)
    (multiplier f-32 five c*9)
    (divider c*9 nine c)))
```

Considered as a way to compute $c = (f - 32) * 5/9$, this code is of course overly verbose, but it does correspond directly to the wiring diagram we are trying to represent. We have chosen, for this prototype, to represent wiring diagrams by Scheme procedures that will attach copies of said wiring diagrams to given boundary cells. Thus `multiplier`, `divider`, and `subtractor` are primitive, one-

propagator "wiring diagrams"—Scheme procedures (to be defined forthwith) that will attach a fresh propagator of the corresponding kind to whatever cells we wish. Likewise `constant`, also awaiting momentary definition, is a wiring diagram generator: for any fixed value `x`, `(constant x)` stands ready to equip a cell with a propagator that will emit `x`. The last primitive this example uses is `make-cell`, whose job of making a fresh cell is the simplest of description, but as we shall soon see, the longest of implementation.

The propagator wiring diagram is very low level, and in that sense more analogous to assembly language than to the expression languages that we are used to. Working with the low-level descriptions, however, clarifies the infrastructure we are building, for it is not obscured with syntactic sugar. This is also simpler for the nonce, because there is no need to implement that sugar yet.

To use this network we need to use `make-cell` to make cells to pass in to this network constructor. We also need to use the cell operation `add-content` to give the Fahrenheit cell a value, and then `content` to get the answer from the Celsius cell:[1]

```
(define f (make-cell))
(define c (make-cell))

(fahrenheit->celsius f c)

(add-content f 77)
(content c)
25
```

So this simple propagator network gave the right answer. The final state of the network is shown in Figure 3-2. This network is like a paper plate that's not being watched by an environmentalist: you only use it once. There is no way to "take the 77 out" and use these same propagators again to convert another temperature—intentionally! The 77 is Information and, per our design principle in Section 2.3.3, Information must never be forgotten. It so happens that the 77 is *complete* information, not open to refinement, so this network is now done. That's not a problem: we can always make more copies of it. In a sense, this network is like the execution of a bit of program, not like its source code. We will consider our options for arranging reuse of networks (or their wiring diagrams) when we consider abstraction in Section 6.2. Until then, we will study propagator networks that behave like program executions; they are fascinating enough!

In the remainder of this section, we will implement the machine that makes this simple network structure go; and we will show off something cool about it in Section 3.2. Then in the rest of this chapter, we will expand this infrastructure to

[1] For the sake of clarity, here and henceforth I will omit the hooks into my software simulator that my implementation technically requires. Those hooks add nothing to the important ideas, and would clutter the presentation with (however little) needless detail. I invite the interested reader to peruse Appendix A.2 for the full exposition; particularly, any attempt to reproduce these examples on this implementation would be well served by attending to that Appendix.

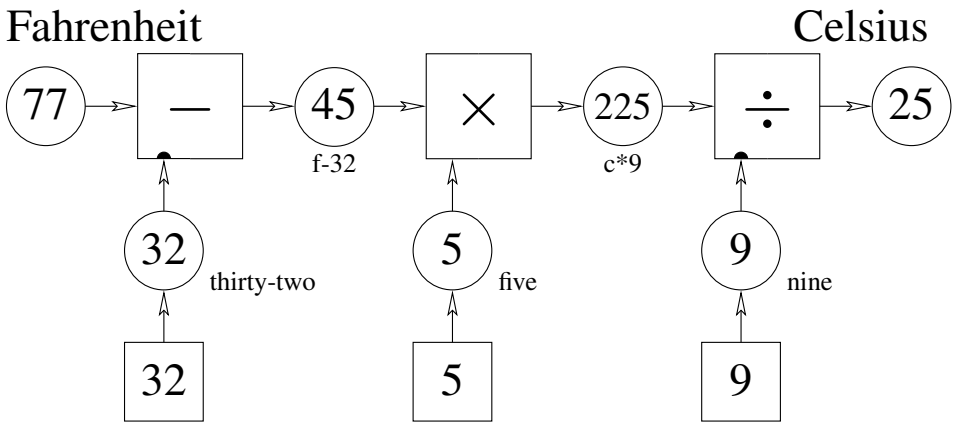


Figure 3-2: A Fahrenheit to Celsius network that has converted 77F to 25C

propagate *partial* information as well—first a specific kind in Section 3.3, just to get the idea, and then generalize to all kinds in Section 3.4—whereupon we will be ready to spend Chapters 4 and 5 building all sorts of systems on this foundation.

It is worth noting that even this simplest of structures (if augmented with an appropriate collection of the facilities discussed in Chapter 6) already recovers expression evaluation as a special case. After all, an expression maps perfectly well into a tree of propagators, one for each subexpression, with a cell for each intermediate value. Our notation for such a tree may be clumsier than the expressions we are used to programming with, but building too familiar a syntax may trap us in habitual modes of thought and obscure the fundamental expressiveness of propagation. Rather than going down that path, and rather than looking to the seams and stitches that await in Chapter 6, let us instead weave the fabric this demo has promised, and see in the following sections and chapters what makes it worthwhile.

An interlude on simulation

What does it take to make this basic example work? The zeroth thing we need is a means to simulate our propagator network in Scheme, as discussed in Section 2.2. The simulation is done by a software job queue. The scheduler is not very sophisticated because this is a prototype whose purpose is to study the essential nature of propagation. A production system based on these principles might dedicate hardware to some propagators, or timeshare batches of propagators on various cores of a multicore processor, or do arbitrarily fancy distributed, decentralized, buzzword-compliant acrobatics. We don't want to worry about any of that now. We want, rather, to learn how to design propagator networks that get the right answers independently of such matters, and thus give the performance gurus free rein to optimize, parallelize, distribute, and comply with buzzwords to their hearts' content. We will gain our expressive power even without sophisticated execution strategies.

This dissertation, therefore, uses a relatively boring software scheduler that

maintains a queue of pending propagators, picking them off one by one and running them until there are no more. The code for that scheduler is not worth presenting here, so it is banished to Appendix A.2. We will return to interesting questions about scheduling in Section 6.4; let us just summarize the matter here by listing the promises this scheduler gives us:

- Propagators are run in an arbitrary, unpredictable order. This is important because we don't want to accidentally depend on the order of their execution.
- Once a propagator is run, by default it will not run again unless it gets queued again.
- Every propagator that is queued will eventually run.[2]
- Every execution of every propagator, including every read and every write of every cell, is atomic. This is an accident of the scheduler's being single-threaded, but it does save us the confusion of cluttering our exploratory code with interlocks.

This relatively loose set of demands on the scheduler follows directly from our stated objective. The scheduler is the custodian of time in our simulation. Since the whole point of studying these systems is to program without being unduly constrained by time, the scheduler must not impose any particular flow of time on our networks. These properties are the closest I could get to running lots of asynchronous little machines on one big computer (without killing myself with random uninteresting multithreading bugs).

In order to maintain the illusion of always-on machines, we will also need to impose a discipline on our cells and propagators. Each primitive propagator will need to promise that a single execution is enough for it to do everything it will do with its currently available inputs. In other words, the software propagator must accomplish, in its one execution, everything the autonomous device it represents would have done with those inputs. This has the effect that the propagator will not need to be run again unless its inputs change. The cells will also need to maintain the invariant that every propagator that might do anything if it were run, that is, one whose inputs have changed since the last time it ran,[3] is indeed queued to run.

Given those promises and that discipline, the job queue becomes a quiescence detector—the non-quiesced propagators are exactly the queued ones. Therefore,

---

[2]Unless the network itself enters an infinite loop. Thoughts about mitigating that possibility are deferred to Section 6.4.

[3]This is counted from when the propagator started running, because it might change its own inputs, in which case it should be eligible for rerunning immediately. In this context, the word "idempotent" is used of any propagator that will not do anything if it is immediately rerun, i.e., one that should not be eligible for rerunning even if it did modify its own inputs. Knowing that a particular propagator is idempotent can lead to a useful optimization, but this system does not require idempotence in general.

when the queue is empty, the simulation can stop and return control to the host Scheme, knowing that it has produced the desired quiescent state.

Making this work

Besides such a simulator, what else does it take to make this basic example work? We need only cells and primitive propagators. We don't need anything else yet, because we are borrowing it all from the host Scheme.

Cells first—the cells make sense by themselves, whereas propagators are harder to think about in a vacuum. The cells are the custodians of state. As such, their job is to remember something. If the propagators really were always on, that would be all. Since our simulator's queue needs to be kept up to date, somebody also needs to requeue all the propagators that are looking at any given cell when that cell's content changes. Since the cell itself is the keeper of its own content, it is in the perfect position to be that someone. Therefore, in addition to its content, each cell also maintains a registry of the propagators that are watching it, and alerts them when its content changes. We impose a discipline on the propagators that they must register themselves with the right cells.

Since a freshly created cell has no sensible content until someone adds some, we define a distinguished object to indicate that a cell contains nothing. Using a distinguished object rather than, say, a separate boolean flag may seem like an arbitrary decision now, but it foreshadows the final design we will settle on later.

```
(define nothing #(*the-nothing*))

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

The actual interface to our cells consists of three functions. `Content` just extracts the cell's current content (possibly returning `nothing` if that's what the cell's content is); `add-content` adds some content to the cell; and `new-neighbor!` asserts that a propagator should be queued when that cell's content changes. It is important that there is no `remove-content` and no `alter-content`. Our design principle from Section 2.3.3 mandates it, and for good reason. This way, the scheduler need not promise to run that cell's neighbors within any particular amount of time—once content has been put into a cell, that cell will never forget it. This fact is critical to liberating ourselves form the tyranny of time.

I arbitrarily chose to implement the cells as closures in the message-accepter style. For simplicity of exposition, I am only supporting a single, specific partial information style in this particular code. This will let me get the propagation system up and running with a minimum of immediate fuss, and then we will generalize it in Section 3.4.

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

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

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

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

There is no great magic going on here. The most interesting of these operations is `add-content`. The logic of what `add-content` does follows from the meaning of a cell's content. In these cells, the designated marker `nothing` means "I know nothing about the value that should be here," and any other value x means "I know everything about the value that should be here, and it is x." In this simple information regime, only four things can happen when someone calls `add-content` on a cell: adding nothing to the cell says "I don't know anything about what should be here," so it's always ok and does not change the cell's content at all. Adding a raw value to a cell amounts to saying "I know that the content of this cell should be exactly x." This is fine if the cell knew nothing before, in which case it now knows its content is x, and alerts anyone who might be interested in that. This is also fine if the cell already knew its content was[4] x, in which case the addition taught it nothing new (and, notably, its neighbors don't need to be alerted when this happens). If, however, the cell already knew that its content was something other than x, then something is amiss. The only resolution available in this system is to signal an error. `Add` or `add` not; there is no `alter`.

The `new-neighbor!` operation adds the given propagator to the cell's neighbor

[4]Equality is a tough subject. We sweep it under the rug for now, but that's ok because it will become irrelevant later.

list. These are the propagators that will need to be queued when the cell's content changes. To be conservative, the cell also queues the new neighbor when the attachment occurs, on the off chance that said neighbor is interested in what the cell has already.

Propagators

With the cells done, we can finally talk about propagators. We are trying to simulate always-on machines. The best we can do in Scheme is to periodically run procedures that do what those machines would do. I chose to ask the user to close those procedures over the relevant cells manually, so that by the time the scheduler sees them they require no further arguments. (We'll see this happen in a minute.) To bless such a nullary procedure as a propagator, all we have to do is to attach it as a neighbor to the cells whose contents affect it, and schedule it for the first time:

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

This procedure arranges, using `alert-propagator`, for the procedure `to-do` to be run at least once, and asks, using `new-neighbor!`, for each cell in the `neighbors` argument to have `to-do` rerun if that cell's content changes. Remembering that each call to `new-neighbor!` also alerts the propagator, one might wonder "why all the alerting?" It is done as a matter of defensive programming, and since an alert means "ensure this propagator will run sometime," alerting the same propagator many times doesn't actually cost anything significant.

So that's how we make general propagators, but to get anywhere we also need some specific ones. One common kind of propagator propagates the result of running a normal Scheme function. The `function->propagator-constructor` procedure makes a primitive propagator constructor like `adder` or `multiplier`. These are the little bits of wiring diagram we have used before to build bigger wiring diagrams like `fahrenheit->celsius`; they are procedures that will, given cells for the function's inputs and output, construct an actual propagator.

```
(define (function->propagator-constructor f)
  (lambda cells
    (let ((output (car (last-pair cells)))
          (inputs (except-last-pair cells)))
      (propagator inputs                ; The output isn't a neighbor![5]
        (lambda ()
          (add-content output
            (apply f (map content inputs))))))))
```

This particular function imposes the convention that a propagator accepts its output cell last; it expects `f` to actually be a function (but see Section 6.6 for dis-

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

cussion of what to do if you want `f` to have side effects whose timing you want to rely upon), and to accept as many arguments as the eventual propagator has input cells. This is not the only way to make primitive propagators, but it is the only one we need for the examples we are dealing with right now.

No one made, and indeed no one should make, any guarantees that the input cells above will all have non-`nothing` contents before that propagator is run. Since our Scheme primitives are not natively equipped to handle the `nothing` token, we must therefore take care of it ourselves. Fortunately, most functions should return `nothing` if any of their inputs are `nothing` (for instance, adding 2 to a completely unknown value ought very well to have a completely unknown result) so we can do this very systematically. In our current situation, the simplest thing to do is to wrap the Scheme primitives in something that will check for `nothing`s before calling them

```
(define (handling-nothings f)
  (lambda args
    (if (any nothing? args)
        nothing
        (apply f args))))
```

and hand that wrapped function to `function->propagator-constructor`. This strategy for handling nothings will prove too simple by Section 3.4, but we will wait until then to upgrade it.

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

```
(define adder (function->propagator-constructor (handling-nothings +)))
(define subtractor (function->propagator-constructor (handling-nothings -)))
(define multiplier (function->propagator-constructor (handling-nothings *)))
(define divider (function->propagator-constructor (handling-nothings /)))
;;; ... for more primitives see Appendix A.3
```

Constants also turn out quite nicely,

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

and that's all we need to make our temperature conversion example work. We will need other things for a full programming system; for instance we haven't dealt with the moral equivalent of Scheme's `if`. But we don't need them yet, and they will fit in smoothly enough when we get to them in Chapter 6. Rather than worrying about that, let's play with this base, simple as it is, and see what we can make out of it.

3.2 Propagation can Go in Any Direction

The preceding example wasn't much to write home about. So we can reproduce small expressions as propagator networks—so what? An evaluator is perfectly good for expressions, and we haven't even shown how networks handle recursion

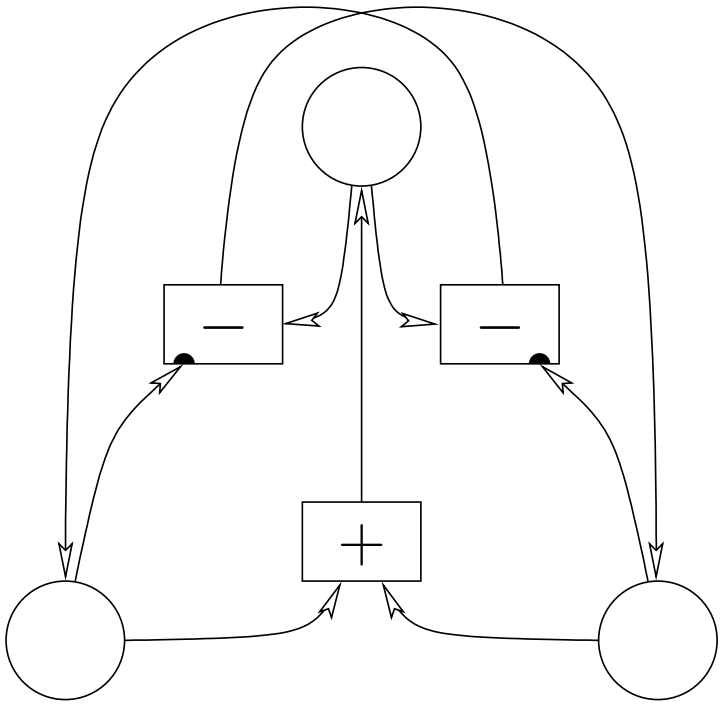


Figure 3-3: A constraint composed of mutual inverses

or compound data, never mind higher order functions. Well, recursion can wait until Section 6.2, compound data until Section 6.3, and higher order networks until future work. We don't need them yet. We can already show off something that's easy to do with propagators but painful to do with expressions.

One of the original motivations for examining propagators is that multidirectional constraints [Steele Jr., 1980, Borning, 1979, Steele Jr. and Sussman, 1980] are very easy to express in terms of unidirectional propagators. For example, the constraint "a is the sum of b and c" can be used to deduce a from b and c (by adding them) or b from a and c (by subtracting c from a) or c from a and b (by subtracting b from a). That's actually hard to express with an expression, but nothing could be easier to write down with propagators:

```
(define (sum x y total)
  (adder x y total)
  (subtractor total x y)
  (subtractor total y x))
```

This network fragment looks like Figure 3-3. It works because whichever propagator has enough inputs will do its computation. It doesn't buzz because the cells take care to not get too excited about redundant discoveries. Products and many other constraints work analogously:

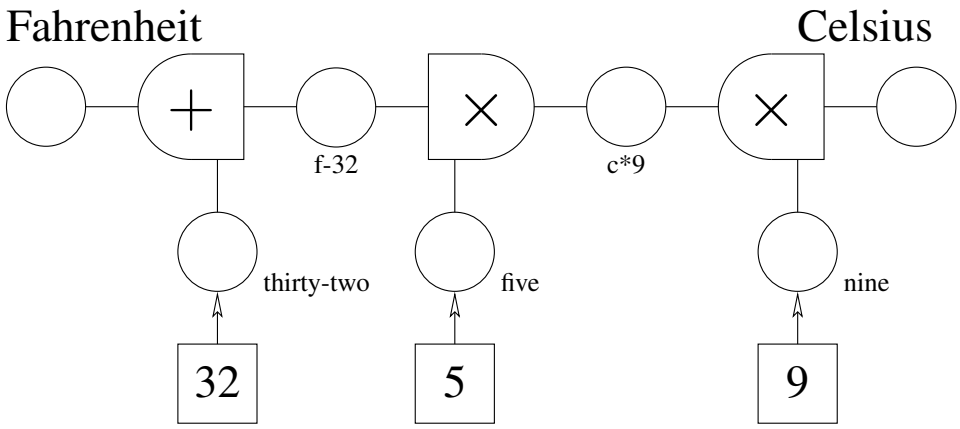


Figure 3-4: A multidirectional temperature converter. Now the + and × boxes each look like Figure 3-3 inside, and can compute in whatever direction the available data permits. That's why they don't have arrowheads on their connections.

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

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

;;; ...
```

Having done that, we can readily upgrade our `fahrenheit->celsius` converter into a constraint that can convert in either direction, dynamically choosing which to do based on which temperature happens to be available at the time. The procedure and the network diagram (Figure 3-4) are almost identical.

```
(define (fahrenheit-celsius f c)
  (let ((thirty-two (make-cell))
        (f-32 (make-cell))
        (five (make-cell))
        (c*9 (make-cell))
        (nine (make-cell)))
    ((constant 32) thirty-two)
    ((constant 5) five)
    ((constant 9) nine)
    (sum thirty-two f-32 f)
    (product f-32 five c*9)
    (product c nine c*9)))
```

And indeed, if we attach it to some cells and feed it, say, a Celsius temperature, it will gleefully produce the corresponding Fahrenheit temperature for us.

```
(define f (make-cell))
(define c (make-cell))

(fahrenheit-celsius f c)

(add-content c 25)
(content f)
77
```

This does not yet constitute a full constraint solving system, because it will get confused by circularities in the constraints (which amount to simultaneous equations). We will return to that problem later (Section 5.3), but for now even this facility can already be useful if it could integrate seamlessly with general programming, and it is already painful to try to build directly in expression-land. Just think about what you would need to do: every time you call it, it would need to check whether it got the Celsius or the Fahrenheit; it would need to contain both the Fahrenheit to Celsius formula and the Celsius to Fahrenheit formula;[6] and its caller would need to figure out whether they got the Celsius or the Fahrenheit as the answer.

If you wanted to extend an expression-based Fahrenheit-Celsius converter to also handle Kelvin, you'd be in an even bigger world of pain (for instance, there would now be six possible combinations of available input and desired output). But if you have the propagation infrastructure available, the Kelvin fragment just hooks on as a separate piece (yielding a combined network that looks like Figure 3-5):

```
(define (celsius-kelvin c k)
  (let ((many (make-cell)))
    ((constant 273.15) many)
    (sum c many k)))

(define k (make-cell))

(celsius-kelvin c k)
(content k)
298.15
```

and even produces the right Kelvin temperature from an input that was already there before it was attached!

Being able to propagate information in any order and direction is extremely powerful. It is interesting and useful in some cases even with complete information, such as the presence or absence of an incontrovertible number, but it really shines when combined with the ability to merge *partial* information that comes in from multiple directions. Let's see that now.

---

[6] If you built the bidirectional temperature converter out of tri-directional arithmetic primitives, you would not need to invert the formula completely—just reverse the order in which the arithmetic is tried. If you didn't even want to do that by hand, you would need to write a loop that tried all the arithmetic operations to see which one could do its computation, and then you'd be quite close to propagator territory.

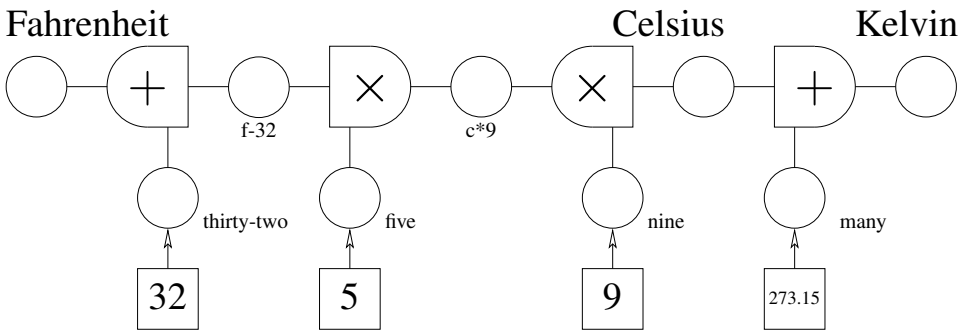


Figure 3-5: A three-way temperature converter. This is an additive extension of the two-way converter in Figure 3-4.

3.3 We can Propagate Intervals Too

When converting temperatures, we have already taken advantage of the big way the propagation model of computation differs from the expression-evaluation model. The difference is that a single cell can get information from multiple sources, whereas the return value of an expression can come from only one source—the expression. That's why the network can convert from Celsius to Fahrenheit just as well from Fahrenheit to Celsius: all the intermediate cells are happy to receive values from either direction, and the propagators will push information whichever way they can.

More generally, however, if information can enter a cell from multiple different sources, it is natural to imagine each source producing some *part* of the information in question, and the cell being responsible for combining those parts. This is the key idea of partial information that we are developing throughout this dissertation, which we are now seeing in action for the first time.

Let us imagine a network whose cells contain subintervals of the real numbers, each interval representing the set of possible values to which the cell's content is known to be restricted. The operations in the network perform interval arithmetic. In this case, if there is some redundancy in the network, a cell can get nontrivial constraining information from multiple sources; since each source authoritatively asserts that the cell's value must be within its limits, the net effect is that the intervals need to be intersected [de Kleer, 1976, de Kleer and Brown, 1992].

To make this example concrete, consider the famous [Calandra, 1961] problem of measuring the height of a tall building by means of a barometer. A great variety of solutions, based on sundry physical principles, [Calandra, 1968] are known; one method, attributed [Mikkelson and Mikkelson, 2007] to Neils Bohr, is to drop it off the roof and time its fall. Then the height h of the building is given by $h = \frac{1}{2}gt^2$, where g is the acceleration due to gravity and t is the amount of time the barometer took to hit the ground. We implement this as a propagator network (just for fun, it includes some uncertainty about the local g):

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

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

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

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

This has the same form as the temperature conversion; the interesting difference is that we are propagating intervals instead of numbers.

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

```
(define (similar-triangles s-ba h-ba s h)
  (let ((ratio (make-cell)))
    (product s-ba ratio h-ba)
    (product s ratio h)))
```

and we can try it out:

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

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

Different measurements lead to different errors, and the computation leads to a different estimate of the height of the same building. This gets interesting when we combine both means of measurement, as in Figure 3-6, by measuring shadows first and then climbing the building and dropping the barometer off it:

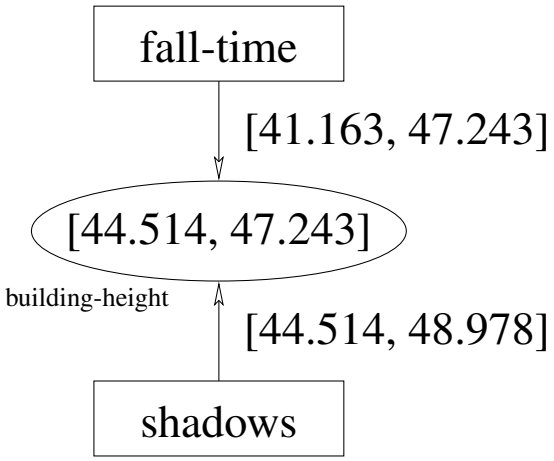


Figure 3-6: Partial answers from different sources merge at a cell. The merged result incorporates all the information available in both partial results.

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

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

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

Because the two experiments are completely independent, this much can be simulated perfectly well in an expression language with an explicit merging step like

```
(intersect-intervals
 (similar-triangles ...)
 (fall-time ...))
```

Putting the merge into the cell itself is perhaps a bit more elegant. It also naturally leads to a more incremental computation—if measuring shadows could take a long time (waiting for a break in the clouds?) a client needing the height of the building can start work with a partial result from the other method, and perhaps refine it when the shadow measurement finally becomes available. This is something of a gain over what we can do with expressions, but not yet very much.

The real advantage of letting the cells merge information is that it lets us build systems with a much broader range of possible information flows. In fact, we snuck that into this example already. Since we built our barometer networks out of constraints, the refined information available about the height of the building propagates backward, and lets us infer refinements of some of our initial measurements!

```
(content barometer-height)
#(interval .3 .31839)
;; Refining the (make-interval 0.3 0.32) we put in originally

(content fall-time)
#(interval 3.0091 3.1)
;; Refining (make-interval 2.9 3.1)
```

Indeed, if we offer a barometer (presumably different from the one we dropped off the roof) to the building's superintendent in return for perfect information about the building's height, we can use it to refine our understanding of barometers and our experiments even further:

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

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

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

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

Here we have just seen multiple ways of solving the same problem augment and reinforce each other; and even better, the *results* of the various methods tell us more about each other's *inputs*. This is a toy example, of course, but this general sort of thing happens all the time in science, and in human thought generally, but takes a great deal of effort to arrange on conventional computers.

This example already shows the power of propagating mergeable information, so we should go ahead and look at the infrastructure that runs it; but we'll start seeing what it can *really* do after Section 3.4, in which we show a general system that can handle numbers, intervals, and lots of other interesting things.

Making this work

Unfortunately, the code we had in Section 3.1 doesn't run this barometer example out of the box. The first reason is that those propagators don't operate on intervals. Therefore, we need some code—Figure 3-7—to actually do interval arithmetic, and we need to make some propagators that expect intervals in input cells and write intervals to output cells. We can just reuse `function->propagator-constructor`, but wrap the interval arithmetic rather than the normal number arithmetic:

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

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

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

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

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

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

Figure 3-7: Deadly boring code for interval arithmetic.[7] Nothing fancy here, especially since we are assuming that all our intervals have positive bounds (and only implementing multiplication, because the examples don't add intervals).

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

The second reason the barometer example doesn't work on intervals out of the box is that the cells from Section 3.1 signal an error whenever any propagator adds anything to a nonempty cell that is not equal to what's already there. So to make this example work, we need to alter our cells to accept intervals, and merge them by intersection rather than checking that they are equal. The relevant change to `make-cell` is marked by `**`.

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

```
(define (make-cell)
  (let ((neighbors '()) (content nothing))
    (define (add-content increment)
      (cond ((nothing? increment) 'ok)
            ((nothing? content)
             (set! content increment)
             (alert-propagators neighbors))
            (else                       ; **
             (let ((new-range
                    (intersect-intervals content
                                         increment)))
               (cond ((equal? new-range content) 'ok)
                     ((empty-interval? new-range)
                      (error "Ack! Inconsistency!"))
                     (else (set! content new-range)
                           (alert-propagators neighbors)))))))
    ;; ... the definitions of new-neighbor! and me are unchanged
  ))
```

This interesting, interval-specific merge code is what allowed our example network to accept measurements from diverse sources and combine them all to produce a final result that was more precise than the result from any one measurement alone.

Sadly, this version won't run the example from Section 3.1 as written, because now it doesn't handle straight numbers anymore—intervals are again a particular, special-purpose partial information structure, with different behavior from the one we used to convert temperatures. But notice how similar the code is, and how little we had to change to move from one to the other. The scheduler, in particular, is untouched. This is a hint that the partial information principle is the right thing. Now, when we generalize to working with numbers *and* intervals in Section 3.4, we will find that we have a system of immense power and flexibility. This is the base on which we will fulfil the promises made in Chapter 1.

3.4 Generic Operations let us Propagate Anything!

In Section 3.1 we propagated numbers, and we saw this was good. In Section 3.3 we propagated intervals, and we saw this was good. But the numbers were separated from the intervals, and we could not propagate the one through the same networks as the other. Let us now merge the two designs, and generalize them, and forge one infrastructure to propagate them all.

Whenever one needs to do the same thing to different kinds of data, generic operations [Kiczales et al., 1999][8] come to mind. There are two places where we had to make changes to move from accommodating numbers to accommodating intervals. We had to change the cell's behavior on receiving new information, and we

[8]Or polymorphism, *ad-hoc* [Strachey, 1967] if you like, or overriding [Gosling et al., 2005], or virtual method dispatch [Stroustrup et al., 1991]. . . . I don't know much about roses, but an idea by any other name works as well.

had to redefine the "primitive" arithmetic. We can make the two kinds of networks interoperate with each other (and with many interesting future kinds we will introduce later) by transforming those places into published generic operations, and allowing them to be extended as needed. This change gains us immense generality and flexibility, which we implement here and will proceed to exploit in the subsequent chapters.

First cells

The code we had to vary between propagating numbers and propagating intervals was the reaction to new information being added to the cell. Numbers we compared by equality, signaling errors when they differed. Intervals we merged by intersection, signaling errors when the intersection was empty. What do these things have in common that can become the contract of a generic procedure?

The commonality is that both are viewed as information about what's in the cell. A number is complete information: "The content of this cell is 5 and that's final." An interval is partial: "The content of this cell is between 4 and 6, but I don't know anything else." In both cases, errors arise when an incoming statement cannot be reconciled with what the cell already knows. For instance, if the intersection of two intervals is empty, then they represent mutually contradictory information, so a cell exposed to this is justified in signaling an error.

This story is amenable to being factored into a generic function, `merge`, whose methods depend upon the kind of partial information being tendered. The `merge` procedure takes over the job of merging the information available to a cell, but we leave the cell itself responsible for alerting its neighbors when things change or when contradictions arise, because that behavior is the same in all cases. The change needed for the new cell is marked by `***`:

```
(define (make-cell)
  (let ((neighbors '()) (content nothing))
    (define (add-content increment)     ; ***
      (let ((new-content (merge content increment)))
        (cond ((eq? new-content content) 'ok)
              ((contradictory? new-content)
               (error "Ack! Inconsistency!"))
              (else (set! content new-content)
                    (alert-propagators neighbors)))))
    ;; ... new-neighbor! and me are still the same
  ))
```

The contract of the generic function `merge` is that it takes two arguments, the currently known information and the new information being supplied, and returns the new aggregate information. If the information being supplied is redundant, the `merge` function should return exactly (by `eq?`) the original information, so that the cell can know that the news was redundant and not alert its neighbors. If the new information contradicts the old information, `merge` should return a distinguished value indicating the contradiction, so that the cell can signal an error. For symmetry and future use, if the new information strictly supersedes the old (i.e., if the old

information would be redundant given the new, but the new is not redundant given the old) `merge` is expected to return exactly (by `eq?`) the new information.[9]

Let us now construct a generic `merge` that obeys this contract for numbers and intervals, and which we will later extend to other partial information types. The particular generic operations facility I am using is incidental. I will describe it enough to make further inline code legible (if you really want to know the gory details, see Appendix A.1), but the interesting thing is what the particular methods are, and the fact that we can attach them independently of each other.

To create a new generic procedure, we use `make-generic-operator`; for this we supply it the procedure's arity, name, and default operation. The default operation will be called if no methods are applicable. For `merge`, we can let the default operation assume that both the content and the increment are complete information. In this case, they are consistent only if they are equal, and if they are not, we should produce a contradiction.

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

We will make the detection of contradictions a generic operation as well, so we can extend it later, but for now it need only check for the sentinel contradiction value.

```
(define the-contradiction #(*the-contradiction*))

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

To add methods to generic procedures we use `defhandler`. The procedure `defhandler` takes a generic procedure (or its name), a method, and a list of predicates, and extends the generic procedure to invoke the given method if the supplied arguments are all accepted by their respective predicates. This is a predicate dispatch system [Ernst et al., 1998].

It turns out that our sentinel value for "no content" fits very nicely into the generic partial information scheme, as a piece of information that says "I know absolutely nothing about the content of this cell". The `merge` methods for handling `nothing` are

---

[9] `Merge` is not entirely symmetric: if the first and second arguments represent equivalent information but are not `eq?`, `merge` must return the first rather than the second. This is a consequence of the asymmetry in the cells' treatment of their existing content versus incoming content. Having `merge` return the wrong one could lead to spurious infinite loops.

```
(defhandler merge
 (lambda (content increment) content)
 any? nothing?)

(defhandler merge
 (lambda (content increment) increment)
 nothing? any?)
```

Together with the default operation on `merge`, these methods replicate the behavior of our first cell (from Section 3.1).

To be able to store intervals in the cells of a network, we need only add a method that describes how to combine them with each other:

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

Interpreting raw numbers as intervals allows us to build a network that can handle both intervals and numbers smoothly.

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

(defhandler merge
 (lambda (content increment)
   (ensure-inside increment content))
 number? interval?)

(defhandler merge
 (lambda (content increment)
   (ensure-inside content increment))
 interval? number?)
```

Already, in a small way, we see the general-purpose propagation infrastructure beginning to let different subsystems (numbers and intervals, in this case) interoperate in the same network. After we finish building it, we will expand this *much* further: with a detailed disquisition on the tracking and uses of dependencies in Chapter 4, and a sweeping overview of all manner of ways to propagate in Chapter 5.

Then propagators

Now we want to upgrade our propagators to handle both numbers and intervals. Recall how in Section 3.1 we made a propagator that could handle numbers (and `nothing`s) with

```
(define multiplier
  (function->propagator-constructor (handling-nothings *)))
```

and how in Section 3.3 we made a propagator that could handle intervals (and `nothings`) with

```
(define multiplier
  (function->propagator-constructor (handling-nothings mul-interval)))
```

To generalize this, let us make propagators out of generic operations instead of the specific functions `*` and `mul-interval`. First, we have to define said generic operations:

```
(define generic-+ (make-generic-operator 2 '+ +))
(define generic-- (make-generic-operator 2 '- -))
(define generic-* (make-generic-operator 2 '* *))
(define generic-/ (make-generic-operator 2 '/ /))
(define generic-square (make-generic-operator 1 'square square))
(define generic-sqrt (make-generic-operator 1 'sqrt sqrt))
;;; ... for more generic primitives see Appendix A.5.1
```

Then we make propagators out of them:

```
(define adder
  (function->propagator-constructor (nary-unpacking generic-+)))
(define subtractor
  (function->propagator-constructor (nary-unpacking generic--)))
(define multiplier
  (function->propagator-constructor (nary-unpacking generic-*)))
(define divider
  (function->propagator-constructor (nary-unpacking generic-/)))
(define squarer
  (function->propagator-constructor (nary-unpacking generic-square)))
(define sqrter
  (function->propagator-constructor (nary-unpacking generic-sqrt)))
;;; ... remainder and details in Appendix A.5.1
```

We wrap the generic operations in a common wrapper to have a common mechanism to handle sufficiently uniform types of partial information. `Nary-unpacking` is a more general and extensible version of `handling-nothings`; and `nothing` is one of the partial information types we will handle uniformly (across all primitive operations) with `nary-unpacking`. The details, along with discussions of the relative merits of available approaches, are in Appendix A.5.1.

Finally, to let our propagators handle intervals as well as numbers, we need to attach interval arithmetic methods to the generic operations.[10]

---

[10]Besides methods implementing interval arithmetic, we also need methods that will promote numbers to intervals when needed. The details are in Appendix A.5.2.

```
(defhandler generic-* mul-interval interval? interval?)
(defhandler generic-/ div-interval interval? interval?)
(defhandler generic-square square-interval interval?)
(defhandler generic-sqrt sqrt-interval interval?)
;;; ... for the other interval methods, see Appendix A.5.2
```

Now that we've done all that work, it's almost time to reap the benefits. First we should verify that nothing broke—indeed, our interval arithmetic still works

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

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

and now it interoperates with raw numbers appearing in the network,

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

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

which get smoothly interpreted as intervals with equal upper and lower bounds where appropriate.

This is the culmination of the effort we set for ourselves in this chapter. We started with a structurally sound but limited propagation system for complete information in Section 3.1, and began benefiting from its multidirectionality already in Section 3.2. We verified its structural soundness by modifying it to operate on intervals, a specific kind of incomplete information, in Section 3.3. Now, seeing where and how the system had to be modified, we united numbers and intervals as methods of common generic procedures, so our machine works equally well with both, even at once.

The real advantage of doing this is that we've built a general infrastructure that can propagate and merge any style of partial information we care to teach it about. The plan for the rest of the dissertation is to spend Chapter 4 carefully working out a partial information structure that implements and uses dependency tracking and belief maintenance; then in Chapter 5, show that by appropriate choices of partial information we will be able to recover the advantages of constraint satisfaction, functional reactivity, probabilistic programming and others in a single unified framework; then with some additional infrastructural labor in Chapter 6 we will also be able to recover regular programming; and we will end with a system that supersedes evaluation, and gains the expressive power that we have been reaching for.

Chapter 4

Dependencies

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

Dependency decorations on data that record the justifications for the data give us a powerful tool for organizing computations. Every piece of data (or procedure) came from somewhere. Either it entered the computation as a premise that can be labeled with its external provenance, or it was created by combining other data. We can add methods to our primitive operations which, when processing or combining data that is decorated with justifications, can decorate the results with appropriate justifications. For example, the simplest kind of justification is just a set of those premises that contributed to the new data. A procedure such as addition can decorate a sum with a justification that is just the union of the premises of the justifications of the addends that were supplied. Such simple justifications can be carried without more than a constant factor overhead in space, but they can be invaluable in the attribution of credit or blame for outcomes of computations—in knowing what assumptions matter for a particular consequence.

By decorating data with dependencies a system can manage and usefully compute with multiple, possibly inconsistent world views. A world view is a subset of the data that is supported by a given set of explicit assumptions. Each computational process may restrict itself to working with some consistent world view. Dependencies allow a system to separate the potentially contradictory consequences of different assumptions, and make useful progress by exercising controlled incredulity.

If a contradiction is discovered, a process can determine the particular **no-**

good set of inconsistent premises. The system can then "chuckle", realizing that no computations supported by any superset of those premises can be believed; computations can proceed in worldviews that do not include the nogood set. This chuckling process, **dependency-directed backtracking**, [Stallman and Sussman, 1977, Lieberherr, 1977, Zabih et al., 1987], can be used to optimize a complex search process, allowing a search to make the best use of its mistakes. But enabling a process to simultaneously hold beliefs based on mutually inconsistent sets of premises, without logical disaster, is itself revolutionary.

Now that we have established a general mechanism for computing with various forms of partial information in Chapter 3, we are ready to consider dependency tracking as a particularly consequential form. As discussed above, dependency tracking is extremely useful in its own right, for provenance information, worldview separation, and search. We will also find in Chapter 5 that similar techniques will help us implement systems that may at first appear to require custom propagation mechanisms.

Dependency tracking is difficult to do and use well in evaluation-based systems, because the linear flow of time imposes spurious dependencies: everything that comes after any point in time seems to depend on everything that went before. But as we shall see in the rest of this chapter, in a propagation setting, dependency tracking is both natural and effective.

We will spend this chapter evolving a system that tracks and uses dependencies. We will start with simply tracking the dependencies of a single value in Section 4.1, and find that we can use the dependency information for end-to-end checks of what inputs actually mattered for a particular result. Then, in Section 4.2, we will implement separable worldviews by using **Truth Maintenance Systems**, [Doyle, 1978, McAllester, 1978, Forbus and de Kleer, 1993], to store multiple values with different justifications. We will extend our truth maintenance machinery to detect and report the causes of contradictions in Section 4.3, and use it to automatically search through a space of hypothetical premises in Section 4.4. This last is a general-purpose implementation of implicit dependency-directed backtracking [Zabih, 1987].

Observe that as we evolve our dependency tracking system below, we need make no changes to any of the code already presented, whether to cells, basic propagators, or the scheduler.[1] All our changes are just new partial information structures. The core propagation system is as modular and flexible as promised in the beginning. As a case in point, when the system needs to change its worldview, as may happen in search, it can do so directly through a partial information structure—no provision for this is necessary in the toplevel controller of the propagation system. This stands in stark contrast to traditional constraint satisfaction, where search is an additional, special-purpose external control loop commanding the propagation proper. We will discuss constraint satisfaction as a specialization

[1] Except, of course, for adding methods to published generic procedures, but that was the point.

of general-purpose propagation in Section 5.3.

4.1 Dependencies Track Provenance

We start with a relatively simple system that only tracks and reports the provenance of its data. How do we want our provenance system to work? We can make cells and define networks as usual, but if we add supported values as inputs, we want to get supported values as outputs. For example, we can label the measurements in our shadow measuring experiment with the metadata that they *are* from the shadow measuring experiment. We make up a premise named `shadows` to stand for that experiment, and say that the inputs are justified by that premise:

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

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

Indeed, our resulting estimate for the height of the building now depends on our `shadows` premise, which follows the consequences of our measurements of the barometer and the shadow through the `similar-triangles` computation. We can try dropping the barometer off the roof, but if we do a bad job of timing its fall, our estimate won't improve.

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

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

What's more, the dependencies *tell* us that it was a lousy timing job: the estimate computed through `fall-duration` and supported by `lousy-fall-time` gave the `building-height` cell no information it didn't already know from `shadows`, so that cell just kept its previous answer, with its previous dependencies. If we time the fall better, then we can get a finer estimate, which then *will* depend on the improved fall timing measurement.

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

This answer still depends on `shadows` because `better-fall-time` didn't completely replace it—the lower bound on the height is still determined by the `shadows` experiment. If we then give a barometer to the superintendent, we can watch the superintendent's information supersede and obsolesce the results of our measurements

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

Now that the super has given us precise data on the height of the building, the multidirectionality of the network lets us infer more about our original measurements, just as it did in Section 3.3. But now, the dependency tracker explicitly tells us which measurements are improved, and on what grounds:

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

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

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

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

In this case, the original `building-shadow` measurement was better than what we could infer about the shadow from the superintendent's extra data, but all our other measurements proved improvable.

I snuck something subtle into this example. We had arranged for the network to treat numbers and intervals as compatible forms of partial information in Section 3.4. Now we expect it to track the dependencies of both without flinching. Beyond combination, this is *composition*: the justification tracking is a partial information type that expects to operate on another partial information type, in this case the number or interval whose justification is being tracked. We will see shortly that this is accomplished by a recursive call to the `merge` function that defines the partial information abstraction.

Here values in cells can depend upon sets of premises. In this example each cell is allowed to hold one value, and the `merge` procedure combines the possible values for a cell into one most-informative value that can be derived from the values

tendered. This value is supported by the union of the supports of those values that contributed to the most informative value. The value is accumulated by combining the current value (and justifications) with the new value being proposed, one at a time.

A justified-intervals anomaly

This particular approach to accumulating supports is imperfect, and can produce spurious dependencies. We can illustrate this with interval arithmetic by imagining three values A, B, C being proposed in order. A possible computation is shown in Figure 4-1. We see that premise A is included in the justifications for the result,

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

Figure 4-1: A justified-intervals anomaly. Tracking dependencies too coarsely can lead to spurious dependencies being asserted.

even though it is actually irrelevant, because it is superseded by the value supported by C. This case actually occurs in the code example above—the dependency of `barometer-height` on `better-fall-time` is spurious, but retained for this reason. The deep reason why this happens is that an interval is really a compound of the upper and lower bounds, and it is profitable to track those dependencies separately; we will study that question, in its general form, in Section 6.3. This anomaly will also become less severe as a consequence of the worldview support we will add in Section 4.2.

Making this work

What do we need to do to make this dependency tracking work? We need to store the justification of each datum alongside that datum; we need to track the justifications properly through merging the data; and we also need to track the justifications properly through computing with the data. Storing is easy: we define a container data structure to store a value together with the dependencies that support it (called a `v&s` for `value&support`; see Appendix A.4).

The important thing is to describe how to merge the information contained in two such data structures; see Figure 4-2. The value contained in the answer must of course be the merge of the values contained in the two inputs, but sometimes we may get away with using only some of the supporting premises. There are three cases: if neither the new nor the old values are redundant, then we need both

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

(defhandler merge v&s-merge v&s? v&s?)

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

Figure 4-2: Merging supported values

their supports; if either is strictly redundant, we needn't include its support; and if they are equivalent, we can choose which support to use. In this case, we use the support of the value already present unless the support of the new one is strictly more informative (i.e., is a strict subset of the same premises).

Here, for the first time, we are using the partial information abstraction for composition (as I kept saying we would). Our dependencies don't necessarily have to justify complete *values*—they can justify any information we want, possibly partial. Merging `v&s`s recursively merges their contents with the same generic `merge`, and uses the results to decide what to do with the justifications.

If it so happens that two supported values contradict each other, we want to return an object that will be recognized as representing a contradiction, but will retain the information about which premises were involved in the contradiction. It is convenient to do that by using the cell's generic contradiction test; that way we can let `v&s-merge` return a supported value whose value is a contradiction, and whose support can carry information about why the contradiction arose.

```
(defhandler contradictory?
 (lambda (v&s) (contradictory? (v&s-value v&s)))
 v&s?)
```

This is also composition: `merge`ing two data can produce a contradiction object, which is also an information state. So we can store it in a `v&s`, and it becomes a contradiction whose reasons we know something about.

Finally, we need to upgrade our arithmetic primitives to carry dependencies

around, and to interoperate with data they find that lacks justifications by inserting empty dependency sets; see Appendix A.5.3. Now supporting individual values works, and we are ready to elaborate the dependency idea into supporting alternate worldviews.

4.2 Dependencies Support Alternate Worldviews

How can we make a machine that believes several different things at the same time? How can we teach it to reason about the consequences of one assumption or another separately, without getting mixed up about which conclusion is due to which source? Well, in the previous section, we used dependencies to *track* the consequences of our various barometric experiments—now we will use them to *segregate* those consequences, and to be able to reason about them in isolation.

We accomplish this separation by giving cells a partial information structure that holds more than one datum at a time, each justified by its own justification. Allowing multiple values makes it possible to efficiently support multiple alternate worldviews: a query to a cell may be restricted to return only values that are supported by a subset of the set of possible premises. Such a subset is called a **worldview**. **Truth maintenance systems** (TMSes) are much-researched [Doyle, 1978, McAllester, 1978, Forbus and de Kleer, 1993] devices that can be used to store multiple values with different justifications for just this purpose.

If we put TMSes in our cells, we can revisit the building-height problem, and see how dependencies and worldviews let us change our mind about which of our experiments to believe, and compute the consequences of one or another subset of our measurements:

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

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

Nothing much changes while there is only one source of information. Where (on page 53) we had a supported value, we now have a TMS with one supported value in it. The answer changes more dramatically when we add a second experiment:

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

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

Now the TMS remembers the deductions made in the first (`shadows`) experiment, as well as the consequences of both experiments together. The consequences of the `fall-time` experiment alone are not computed yet, because we chose to build our system not to compute them. The reasoning behind this design decision was that, in my experience, it was not a good idea to compute all the consequences of all the premises all the time; rather we just compute the strongest consequence of all the premises in the current worldview taken together. The current worldview at this point believes both `shadows` and `fall-time`, because we made believing everything the default, and that default has not yet been overridden.

In this particular system, we chose to make the worldview implicit and global. That works fine on a uniprocessor, but a more distributed propagator network might be better served by a more local notion of worldview, and by propagating changes thereto explicitly rather than letting them instantly affect the entire network.[2]

With an implicit global worldview, one can **query** a TMS, without additional input, to get the most informative value supported by premises in the current worldview:

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

As we will see later, `tms-query` is the lens through which (most) propagators view TMSes—this is what allows the network to work on just one worldview at a time, without having to work out all the myriad variations of what to believe and what not to believe until requested.

The answer above should remind the reader of the situation that obtained on page 54. We have again computed the best estimate of the height of our building from measuring various shadows and then dropping a barometer off the roof. The new and powerful development is that we also provide a means to manipulate which premises are in the current worldview. For our next example, removing the `fall-time` premise causes the TMS query to fall back to the less-informative inference that can be made using only the `shadows` experiment:

---

[2]It would probably be a good idea for waves of worldview changes to travel through the network "faster" than other changes, to minimize work spent on computing in worldviews that are no longer interesting. Section 6.4 discusses possibilities for a coherent and general mechanism that could be used to achieve this.

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

If we also remove the `shadows` premise, our poor system no longer has anything to believe at all! Therefore, `tms-query` dutifully reports that `nothing` can be concluded about the height of the building from the currently believed premises:

```
(kick-out! 'shadows)
(tms-query (content building-height))
#(*the-nothing*)
```

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

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

The consequences of `fall-time` without `shadows` are new: this is the first time we put the network into exactly that worldview. This is therefore a new deduction, and the full TMS remembers it:

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

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

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

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

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

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

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

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

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

On this simple example, we have illustrated a very powerful mechanism. Our network can believe various different pieces of information, and by keeping track of the reasons why it believes them, the network can keep them separate from each other, and keep straight the different consequences of its different beliefs. By manipulating the worldview—the set of premises the network believes at any one time—we can ask it to compute what it can supposing one set of beliefs or another, as we like.

This much is already useful, flexible, and powerful. But we will see in Section 4.3 that even mutually inconsistent beliefs do not cause any great trauma, and then we will empower our dependency system even further in Section 4.4 by teaching the machine to hypothesize new beliefs of its own, and to automatically search through the space of possible beliefs to find consistent ones.

The justified-intervals anomaly revisited

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

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

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

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

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

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

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

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

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

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

Making this work

The first component of making worldviews work is to define a suitable TMS to put into cells. The TMS we build here is optimized for simplicity and elegance rather than performance. Its only interesting technical advantage over the fine fare in [Forbus and de Kleer, 1993] is that it is functional: "adding a fact" to an existing TMS does not change it, but creates a new TMS that's just like the old one, but with that fact added. This feature is important because the cells expect to be able to compare an old TMS with a new, and having additions mutate the structure would serve that purpose ill.

Our TMS is a set of `v&s`s. In Appendix A.4 we define a `tms` record structure that contains a list of `v&s` records. These `v&s`s represent the direct deductions the surrounding system has added to the TMS, and any consequences thereof the TMS has deduced on its own. Asking the TMS to deduce all the consequences of all its facts all the time is perhaps a bad idea, so when we merge TMSes we assimilate the facts from the incoming one into the current one, and then deduce only those consequences that are relevant to the current worldview.

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

(defhandler merge tms-merge tms? tms?)
```

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

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

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

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

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

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

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

(define (v&s-believed? v&s)
  (all-premises-in? (v&s-support v&s)))

(define (all-premises-in? premise-list)
   (every premise-in? premise-list))
```

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

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

To support the implicit global worldview, we need a mechanism to distinguish premises that are believed in the current worldview from premises that are not. Premises may be marked with "sticky notes"—using a hash table to remember things about the premises without altering their identity. Appendix A.4 shows how this is arranged.

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

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

In order to let cells containing TMSes interoperate with cells containing other kinds of partial information that can be viewed as TMSes, we also augment our generic arithmetic operations. The sufficiently curious reader can find the details in Appendix A.5.4; the interesting point is that they avoid working on the entire contents of any given TMS by calling the `tms-query` procedure.

4.3 Dependencies Explain Contradictions

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

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

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

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

but it is appropriate for the system not to propagate consequences deducible in an inconsistent worldview, so the `barometer-height` remains unchanged:

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

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

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

and on that of the barometer.

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

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

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

As promised, contradictory beliefs are not traumatic. The deep reason contradiction handling works so well is that `the-contradiction` is just another partial information state, and our truth maintenance machinery operates on partial information by design. On the other hand, contradictions are not quite like the other kinds of partial information, because we chose to make them have a global effect: we wanted our system to stop computing things in a given worldview immediately, as soon as it discovered that worldview to be contradictory anywhere. We must therefore do some work to treat contradictions a little specially. This work is a consequence of our particular choice of making the worldview global.

Making this work

Because we want to pay special attention to contradictions, we will add consistency checks to key places. It suffices to modify just our existing truth maintenance system, and in just two places, marked ** below, to check for the consistency of computed results. The procedure `tms-merge` needs to check the consistency of the newly deduced consequence

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

(defhandler merge tms-merge tms? tms?)
```

and the procedure `tms-query` needs to check the consistency of the answer it is about to return.

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

Actually checking that something is consistent amounts to verifying that it is not contradictory. The interesting part is extracting the set of premises that support the contradiction, called a **nogood set**, so that something can be done with it.

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

The simplest `process-nogood!` procedure just aborts the process, preventing further computation in the worldview that is now known to be inconsistent, and giving the user a chance to adjust the worldview to avoid the contradiction:

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

This will be expanded next—we will teach the system to make its own hypotheses, and automatically retract them if they prove contradictory, thus building a dependency-directed search into our infrastructure.

4.4 Dependencies Improve Search

Implicit generate-and-test can be viewed as a way of making systems that are modular and independently evolvable. Consider a very simple example: suppose we have to solve a quadratic equation. There are two roots to a quadratic. We could return both, and assume that the user of the solution knows how to deal with that,

or we could return one and hope for the best. (The canonical `sqrt` routine returns the positive square root, even though there are two square roots!) The disadvantage of returning both solutions is that the receiver of that result must know to try his computation with both and either reject one, for good reason, or return both results of his computation, which may itself have made some choices. The disadvantage of returning only one solution is that it may not be the right one for the receiver's purpose.

A better way to handle this issue is to build backtracking into the infrastructure (as proposed, for instance, by [Floyd, 1967, Hewitt, 1969, McCarthy, 1963], and implemented, e.g., by [Abelson et al., 1996, Siskind and McAllester, 1993]). The square-root procedure should return one of the roots, with the option to change its mind and return the other one if the first choice is determined to be inappropriate by the receiver. It is, and should be, the receiver's responsibility to determine if the ingredients to its computation are appropriate and acceptable.[3] This may itself require a complex computation, involving choices whose consequences may not be apparent without further computation, so the process is recursive. Of course, this gets us into potentially deadly exponential searches through all possible assignments to all the choices that have been made in the program. As usual, modular flexibility can be dangerous.

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

We illustrate this idea with some information about the building's occupants we have gleaned, with due apologies to [Dinesman, 1968], from the chatty superintendent while we were discussing barometric transactions:

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

Should we wish to determine from this where everyone lives, we are faced with a search problem. This search problem has the interesting character that the

[3] Such ideas of end-to-end checks arise in many other circumstances as well, for instance testing data integrity in Internet transmissions. The basic reason why they are important is that the intermediate layers doing subordinate computation or transmission are at the wrong level of abstraction to check that they did everything right, and therefore the toplevel agents on either of the channel should verify and confirm with each other that things worked out well.

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

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

To make the example concrete, Figure 4-3 shows a direct encoding of the problem statement above as a propagator network.

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

Figure 4-3: The superintendent's puzzle

Observe the generators `one-of`, which guess floors where people live but reserve the right to change their guesses later, and testers `require` and `forbid` that point out situations that necessitate the changing of guesses. We will show their implementation in full detail before the section ends.

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

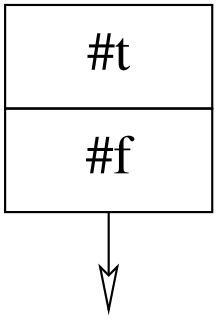


Figure 4-4: Conventional diagram of a `binary-amb`

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

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

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

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

Making this work

We start by adding a mechanism for making guesses. For simplicity, it will only guess a boolean value, namely either `#t` or `#f`. We can always bootstrap an arbitrary discrete guesser from a pattern of these (and in fact the `one-of` generator referenced in our `multiple-dwelling` example does that). The guessing machine is called `binary-amb` after `amb`, the ambiguity operator suggested by [McCarthy, 1963]. We adopt Figure 4-4 as the diagrammatic picture for this object.

The code implementing the `binary-amb` machine is in Figure 4-5. This machine works by manufacturing two new premises, and adding to its cell the guess `#t` supported by one premise and the guess `#f` supported by the other. It also creates an `amb-choose` propagator whose job is to watch the worldview and work to ensure that it believes exactly one of these premises at any time.

Premises accumulate reasons why they should not be believed (data structure details in Appendix A.4). Such a reason is a set of premises which forms a nogood set if this premise is added to it. If all the premises in any such set are currently believed, that constitutes a valid reason not to believe this premise. The `amb-choose` procedure checks to see whether it is consistent to believe either of its premises, and does so if possible. If neither of a guesser's premises can be believed, the guesser can perform a resolution step to deduce a contradiction that does not involve either

```
(define (binary-amb cell)
  (let ((true-premise (make-hypothetical))
        (false-premise (make-hypothetical)))
    (define (amb-choose)
      (let ((reasons-against-true
             (filter all-premises-in?
               (premise-nogoods true-premise)))
            (reasons-against-false
             (filter all-premises-in?
               (premise-nogoods false-premise))))
        (cond ((null? reasons-against-true)
               (kick-out! false-premise)
               (bring-in! true-premise))
              ((null? reasons-against-false)
               (kick-out! true-premise)
               (bring-in! false-premise))
              (else                     ; this amb must fail.
               (kick-out! true-premise)
               (kick-out! false-premise)
               (process-contradictions
                (pairwise-resolve reasons-against-true
                                  reasons-against-false))))))
    ((constant (make-tms
                (list (supported #t (list true-premise))
                      (supported #f (list false-premise)))))
     cell)
    ;; The cell is a spiritual neighbor...[4]
    (propagator cell amb-choose)))
```

Figure 4-5: A guessing machine

of its premises. After such a contradiction is signalled and an offending premise is disbelieved, the `amb-choose` will eventually be rerun, and may perhaps then be able to believe one or the other of its premises.

The `pairwise-resolve` procedure takes two lists of complementary nogood sets and produces a list of all the contradictions that can be deduced from them:

```
(define (pairwise-resolve nogoods1 nogoods2)
  (append-map (lambda (nogood1)
                (map (lambda (nogood2)
                       (lset-union eq? nogood1 nogood2))
                     nogoods2))
              nogoods1))
```

One list contains sets of premises that would suffice for a contradiction, if supplemented with the `true-premise`; the other contains those that would cause a contradiction if supplemented with the `false-premise`. The deducible contradic-

[4] As this code is written, the `amb-choose` propagator does not read input from anywhere, so it could get away without being attached to the cell, as in `(propagator '() amb-choose)`. That only works, however, because the worldview is implicit and global, and the `amb-choose` gets realerted behind the curtain every time to worldview changes. In a more honest system, one could imagine a local worldview stored in each cell, in which case `amb-choose` would have to read the worldview from the cell the `binary-amb` construct is attached to.

tions are all the unions of all pairs of sets from the two input lists. This constitutes a resolution of each of the nogoods represented by the `reasons-against-true` against each of those represented by the `reasons-against-false`.

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

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

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

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

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

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

Finally, we have the machinery to let a contradiction discovered in the network (by `check-consistent!`, see page 65) trigger an automatic change in the world-view.

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

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

The emergent behavior of a bunch of `binary-amb` propagators embedded in a network is that of a distributed incremental implicit-SAT solver based on propositional resolution. This particular SAT solver is deliberately as simple as we could make it; a "production" system could incorporate modern SAT-solving techniques from the abundant literature (e.g., [Abdelmeged et al., 2007]). The network naturally integrates SAT-solving with discovering additional clauses by arbitrary other computation. From the point of view of the SAT solver, the SAT *problem* is implicit (in the computation done by the network). From the point of view of the computation, the SAT solver's *search* is implicit. We will discuss this relationship more in Section 5.1.

Ambiguity Utilities

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

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

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

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

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

Upgrading a binary choice to an n-ary choice can be a simple matter of constructing a linear chain of binary choices controlling conditionals,[5] like Figure 4-6. For completeness, code for this follows:

---

[5] We are using the `conditional` box we will describe carefully in Section 6.1. Suffice it to say for now that it forwards one or the other of its inputs on to its output, based on the value of its control, and adds the dependencies of the control to the dependencies of the output it produces.

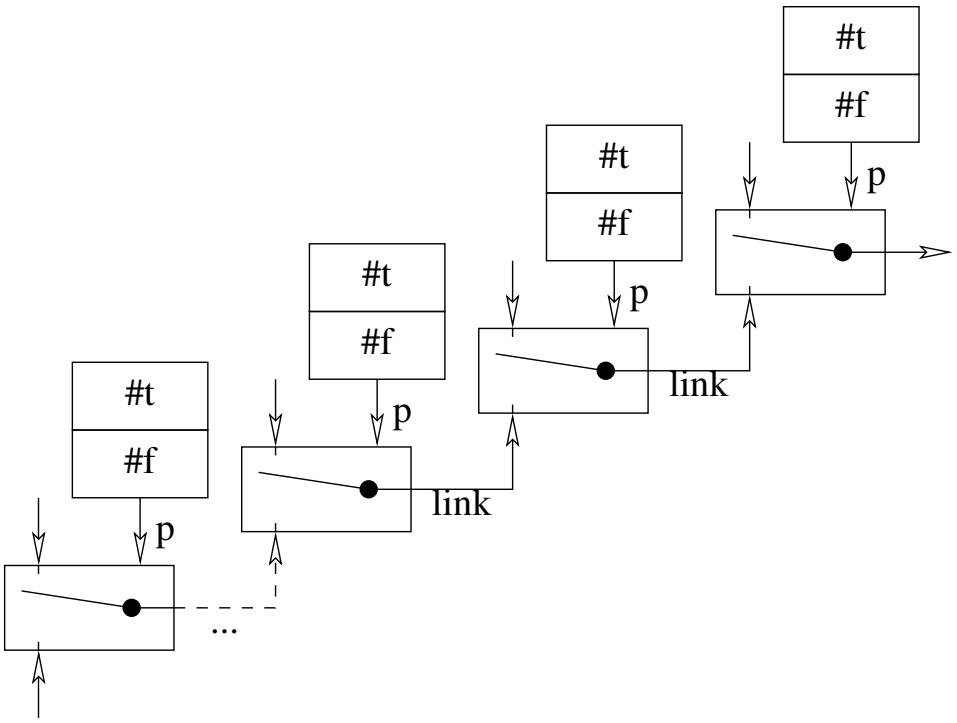


Figure 4-6: A linear chain turning binary guessers into an n-ary guesser. It consists of `binary-amb` boxes, drawn as in Figure 4-4, and `conditional` boxes, to be defined carefully in their own Section 6.1, and drawn as in Figure 6-4 on page 112.

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

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

At the beginning of this chapter, I promised to show how dependency decorations give us a powerful tool for organizing computation. In the course of this chapter we have seen, as promised, how dependencies can be used to track the provenance of data, and in so doing, allow us to examine alternate worldviews simultaneously, and finally, how these abilities make possible a sophisticated and efficient search. Though on one hand this chapter is much more special-purpose than the last, being entirely devoted to an examination of one particular partial information type—one among the myriads that could be plugged into this system—on the other hand, it is this example more than any other that shows the quantum leap of modularity that the partial information idea makes possible. It is generally thought that a propagation system needs a search algorithm to be built into the core scheduler: we have just shown how it can instead be associated with a composable partial information type and tuned to the needs of each problem separately, without modifying the core or the other information propagated by the network. This is one of the revolutionary aspects of this system: when we turn, in the next chapter, to the subject of the expressive power of propagation, we will see its consequences play out.

Chapter 5

Expressive Power

CONSIDER what we have accomplished so far: we have built the core of a general-purpose propagation infrastructure in Chapter 3. We have used it, without modification, to track dependencies and support multiple worldviews in Chapter 4. That was a very detailed and careful example of the generality and flexibility of the idea of propagation; it also showed that carrying around additional information with long-range identities (dependencies, in that instance) is a valuable design pattern for using propagation to do things that may seem surprising at first. Let us now look upon the extent of propagation's expressive power from a greater distance, so as to look over more ground. We will study the task of building a complete programming system on this expressive substrate in Chapter 6.

5.1 Dependency Directed Backtracking Just Works

Search strategies have interested artificial intelligence researchers since the inception of the field. Many and sundry have been invented; but I want to talk about one in particular because it benefits drastically from a propagation infrastructure. As such, it can perhaps serve as an example of a new class of search strategies that were very difficult to explore under the evaluation paradigm.

The strategy in question is called **dependency-directed backtracking**. We made our network do dependency-directed search in Section 4.4, but let us take a minute to examine what happened, and why this is not like the searches one is used to. To understand the idea, consider an abstract search problem. Suppose we have four binary choices a, b, c, and d that we can make independently, and we need to search to find an acceptable combination. We are used to thinking of searches as linear sequences of choices—first make one choice, then make another choice, and so on. We are therefore used to drawing such search spaces as trees like Figure 5-1: branch on how the first choice went, then in each branch, branch again on how the second choice went, etc. With four choices to make in our little example, that's $2^4 = 16$ branches to examine.

This kind of search strategy meshes wonderfully with the evaluation paradigm of computing. Evaluation forces an order on time, and the choices we have to

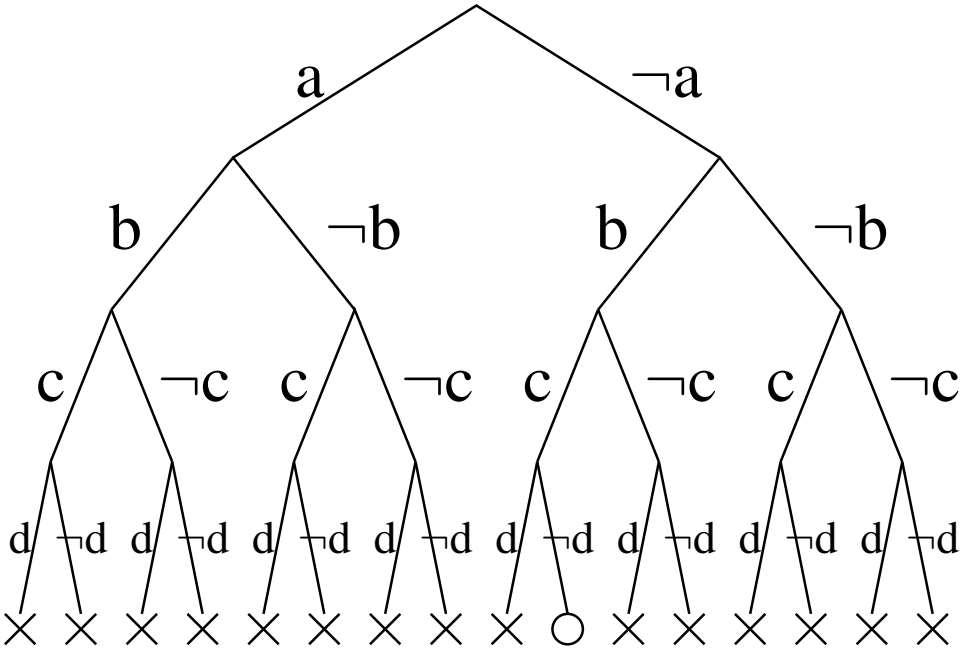


Figure 5-1: A search tree over the four binary choices a, b, c, d. Without further information, a search would have to examine all 16 combinations of choices to determine that $\{\neg a, b, c, \neg d\}$ was the only acceptable arrangement.

make fit into that order. We think of the search space as a tree, and our search program traverses that tree in, say, depth-first order. And every time a (leaf?) node of the tree is found unacceptable, the search program just sighs, backtracks to the last choice it made, and tries another guess. This process can in fact be made implicit with a device like `amb` [McCarthy, 1963, Abelson et al., 1996], (which can be implemented as a library for an evaluation-style language using continuations [Sitaram, 2004]), thereby cleanly separating the program from the search strategy.

But now suppose we want a more sophisticated search strategy. Suppose that we had a mechanism, such as the one we have been studying in Chapter 4, for tracking the consequences of each of our choices. In other words, suppose we could maintain dependencies such that whenever we found a node in the search space unacceptable, we had a list of which choices actually mattered for reaching that unacceptable state. For instance, suppose our search could discover, as it progressed, that each of the sets of choices $\{a, b\}$, $\{\neg b, c\}$, $\{\neg c\}$, and $\{\neg a, b, c, d\}$ was unacceptable, as in Figure 5-2. Then there are three ways we could make our search better.

First, instead of returning to an arbitrary (and perhaps irrelevant) choice (such as the chronologically last one made), we can return to the last choice that actually mattered for producing the observed failure. For example, if we explore the $\{a, b, c, d\}$ branch first, and discover that $\{a, b\}$ suffice to make it unacceptable, we can skip an entire subtree of our search. The left half of Figure 5-2 has three in-

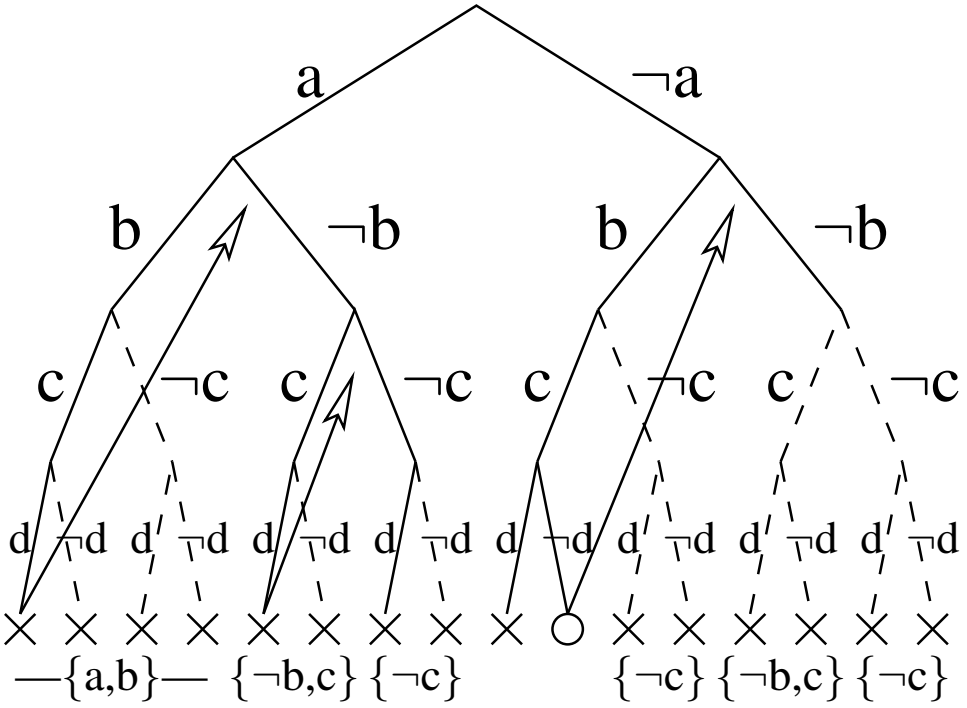


Figure 5-2: A search tree with dependency information explaining the reasons for failures. A search that uses the dependency information can get away with examining just five of the terminal positions, and can also avoid a fair amount of intermediate computation.

stances of this skipping. This improvement saves us from rediscovering the same contradiction again and again while we vary choices made after those that will force that contradiction to occur. This much is easy to implement with continuations.

Second, we can memorize the sets of choices that caused failures, and ask our choice nodes to avoid them in the future. For example, having discovered in the $\{a, \neg b, \neg c, d\}$ branch that $\{\neg c\}$ alone causes a contradiction, we can skip any future subtrees that include that choice. The right half of Figure 5-2 is dominated by two instances of this skipping. Such memorized choice sets are called **nogood sets**, or just **nogoods**. This would save us from rediscovering the same contradiction over and over again while we vary choices that were made before the ones that forced the contradiction, but which do nothing to forestall it. In the example, changing a to $\neg a$ does not save us from the contradiction $\{\neg b, c\}$ (or $\{\neg c\}$, for that matter). Now this improvement is suddenly a pain to implement with continuations, because when we backtrack past one of the choices in our nogood set, we must somehow remember about it, so that when we return to that choice point later in the search, we remember the nogoods we learned about it; in other words, our choices need to have explicit *identities*. That's actually very difficult [Zabih et al., 1987, Hanson, 2007], because the natural `amb` device makes a fresh choice point every time it is called, and if one invokes a continuation to backtrack past one of them, its identity will be forgotten,

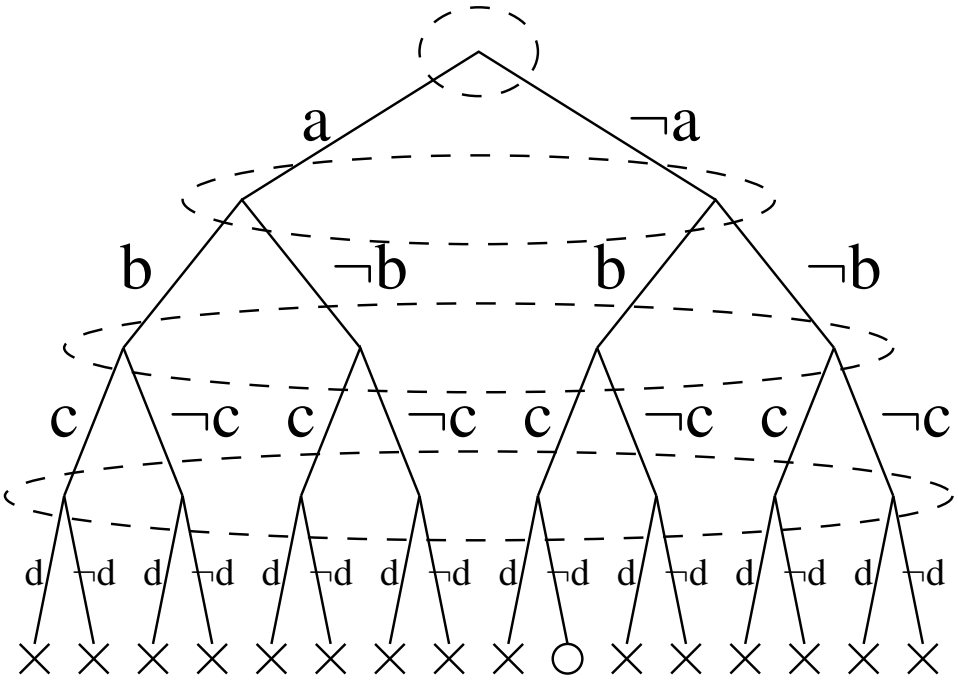


Figure 5-3: A search tree does not capture choice identity. A single choice turns into a whole level of the search tree, and its identity is not represented.

as shown in Figure 5-3.

Third, if we're tracking the dependencies anyway, then when we change a choice, we should be able to avoid recomputing anything that doesn't depend on that choice. This improvement is also very difficult to implement with continuations, because again, there is no good way to keep track of the fact that one is invoking the same continuation with different values, and have it "just remember" things it already computed that are not affected by the new value one is supplying.

All three of these kinds of search improvements can be especially important if the major cost in a computation is not the search itself, but some other computation that depends on the choices the search makes. For example, if we are doing symbolic algebra and guessing the positive or negative roots of (relatively few) quadratic equations, the actual symbolic manipulations can be far more expensive than the mechanics of searching. In such a case, avoiding even one spurious branch can be worth all the overhead of wrangling dependencies.

All these troubles with implementing dependency-directed search are due to the over-rigid notion of time that follows from the evaluation strategy. When one is evaluating, time is linear, and everything that comes after automatically depends on everything that came before. So even if you can track the dependencies and discover things that are independent, there is no way to go back and change your mind about only the computational consequences of a choice, rather than about everything that chronologically followed it.

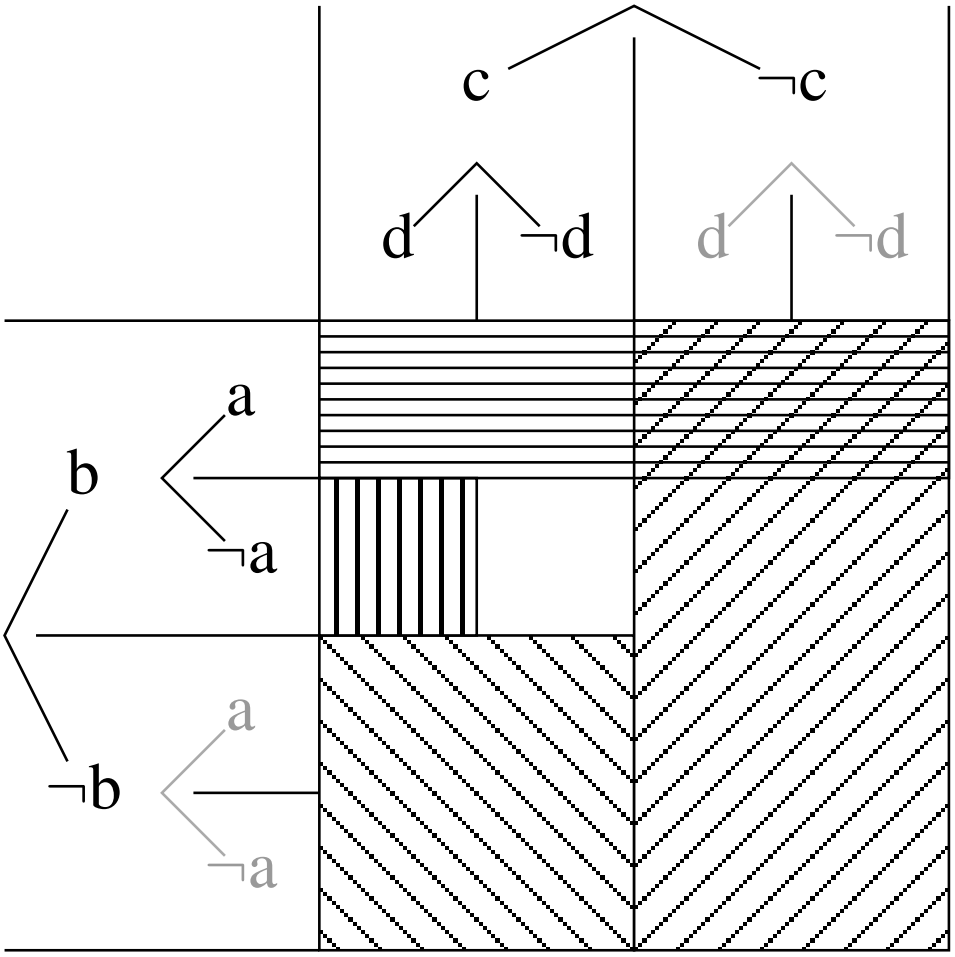


Figure 5-4: A search hypercube. Discovered nogoods can eliminate large chunks of the cube at once: Each of the nogoods $\{a, b\}$, $\{\neg b, c\}$, $\{\neg c\}$, and $\{\neg a, b, c, d\}$ is indicated with its own shading.

Moving to propagation instead of evaluation solves this problem. Propagation has a much more flexible intrinsic notion of time.[1] Implicit dependency-directed search can be achieved by propagating a truth maintenance data structure, as we did in Section 4.4.

Doing that removes the linear-branching structure of our search "tree". Each choice becomes an entity in its own right, instead of being a row of nodes in a search tree; so their identities are conserved. The result is less of a search tree and more of a search hypercube, where each choice is its own (Boolean) dimension, and discovered contradictions can eliminate entire subspaces. For our example, this is shown, as best a four-dimensional structure can be shown on a sheet of paper, in Figure 5-4.

[1]Like nearly all of our most powerful analytical metaphors, propagation technology turns time into space. We are used to thinking of space as having more dimensions than time, hence the flexibility.

The state of the search is now stored explicitly in partial information structures in the cells instead of implicitly in the control flow of the evaluation. When a choice needs to be changed, the partial information structure at that choice changes, and its consequences are propagated to those cells that depend on it. Other choices are automatically unaffected, and intermediate values that don't depend on the choice are automatically not recomputed. There are no spurious dependencies on the order in which choices are made, so those spurious dependencies don't cause any trouble.

Considered from the propagators' perspective, dependency-directed search is just a mechanism for solving long-range Boolean dependencies—if raw propagation of certain values is not enough to deduce the answer, make a hypothetical guess, and propagate its consequences, keeping track of which guess was responsible. If you hit any contradictions, send them back to the guess so it can be adjusted. One might think of this as a kind of reasoning by trial-and-error.

5.2 Probabilistic Programming Tantalizes

The idea of probabilistic programming has been the subject of much recent research [Ramsey and Pfeffer, 2002, Goodman et al., 2008, Radul, 2007]. The motivating observation is that many useful programs need to perform deductions under regimes of uncertain knowledge. Probability theory is the extension of classical logic to reasoning with propositions whose truth is uncertain, if the uncertainty can be quantified [Jaynes, 2003]. In effect, doing probability is like doing logic but with numbers attached. Since we have just been doing a lot of logic (and since I think probability is important), let's see what propagation networks can do for probability.

The laws of probabilistic reasoning are strict enough to implement once for the general case, and yet subtle enough that humans regularly misapply them in specific cases. Hence there is great potential benefit to embedding those laws in a programming system, so that human programmers can focus on writing programs in it, and the system itself can take care of obeying the rules of probability correctly. Many such systems have been built [Pfeffer, 2001, Milch et al., 2005]. They vary in their surface syntax and expressivity, but the basic idea is that an expression like

```
(if (flip 1/3)
    1
    (if (flip 1/2)
        2
        3))
```

represents both a random *process* (in this case, a process with one weighted coin flip, and possibly another fair coin flip depending on how the first flip went) and a *probability distribution* on the possible results of that process (in this case the uniform distribution on $\{1, 2, 3\}$). The system's job, then, is to allow the programmer to write as expressive a set of such programs as the system can support, and then to answer questions about the distributions those programs represent.

So what can propagation do for probabilistic programming? We should expect some benefit, because propagation is all about multidirectionality, and probability is essentially multidirectional. And indeed, just as in Sections 4.4 and 5.1 we saw how propagation allows for dependency-directed search for solving logical problems, so propagation allows for dependency-directed inference for reasoning about probabilistic programs. Propagation also interacts with probability in other ways, which, lacking space, I cannot dwell on, but which I will mention in passing.

Let us therefore consider the essentially multidirectional nature of probabilistic reasoning in some more depth. We will then be able to outline dependency-directed inference, which should be implementable directly on the propagation and dependency tracking infrastructure presented in this dissertation. Let us thereafter reflect on some further, more speculative potential interactions between probability and propagation.

Multidirectionality

So why is probabilistic programming essentially multidirectional? Probabilistic programming is essentially multidirectional because of evidence. The rules of probability (and of normal logic, for that matter) demand that observation of *effects* be able to yield information about *causes*. For example, suppose you come home one day and observe that the grass on your lawn is wet. Suppose, further, that in this idealized world, the only way the grass could have gotten wet was either for it to have rained or for the sprinkler to have turned on and sprinkled it [Russell and Norvig, 1995]. Then you can deduce that one of those two things must have happened. The wetness of the grass does not *physically* cause either rain or sprinkling—indeed the reverse—but wetness *logically* causes you to deduce something about rain and sprinkling.

To turn this into a probability problem, suppose we don't already know whether it rained or sprinkled, but we have some prior knowledge about how often those two things happen (and about their relationship to each other). Suppose we are interested in how our knowledge about the events of this particular day changes, based on our observation that the grass is wet. Then we can write this scenario down as a probabilistic program, for example this one:

```
(let ((rained? (flip 1/5))
      (sprinkled? (flip 4/5)))
  (let ((wet? (or rained? sprinkled?)))
    (observe! wet?)                    ; **
    (list rained? sprinkled?)))
```

Here we have an explicit model of the situation. We are positing that our prior knowledge (before observing the wetness of the grass) is that it rains 20% of the time, and that the sprinkler activates 80% of the time, independently of whether or not it rains. We are further positing that the grass is wet if and only if it either rained or sprinkled (or both), and, on the line marked $**$, that the grass is, indeed, wet.

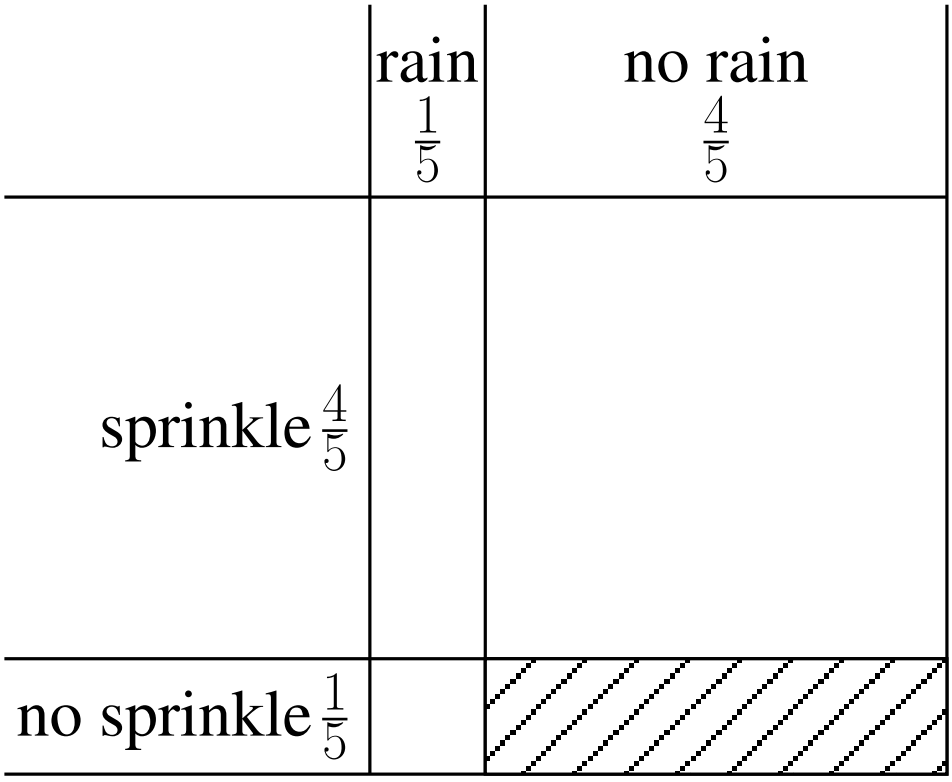


Figure 5-5: Deduction from observing that the grass is wet. The wetness of the grass eliminates the possibility that it neither rained nor sprinkled, and distributes the prior probability of that event among the others.

If we tasked a probabilistic programming system with producing the distribution of possible return values of that expression, that agree with the observation that the grass is wet, it should give us a moral equivalent of

```
(((#t #t) . 4/21)
 ((#f #t) . 16/21)
 ((#t #f) . 1/21))
```

Such an answer means that the probability that it rained and sprinkled is 4/21, the probability that it sprinkled and didn't rain is 16/21, and the probability that it rained but didn't sprinkle is 1/21 (and the probability that it neither rained nor sprinkled is zero, because then the grass would not have been wet). These situations and their relative weights look like Figure 5-5.

Observe how learning about the wetness of the grass raised the probability of it having rained from the prior 1/5 to the posterior 5/21, and the probability of the sprinkler having turned on from the prior 4/5 to the posterior 20/21. Of course, the grass being wet has no physical effect on the rain, but it gives us *information* about whether it rained or not, and the probability of rain having occurred *given our knowledge* goes up. So reasoning is multidirectional: besides obviously having

to reason *forward* along the direction of causality, from known causes to presumed effects, we must also be able to reason *backward*, from observed effects to hypothesized causes (and then to other effects of those causes, and to other causes of those other effects, as far as the deductions go).[2]

Inference

The point of probabilistic programming is to let programmers specify probability distributions by writing the programs that give rise to them, and then having the underlying system answer questions about those distributions. The process of contemplating a distribution in order to answer questions is called **inference**, and different algorithms for doing it are called **inference strategies**.

Since the laws of probability are multidirectional, every inference strategy must embody one or another way by which observing evidence affects the probabilities of causally related events (regardless of the direction of the causality links). In other words, there must be some communication "backward" along causality links—evidence about results must talk to beliefs about causes. How is that done now, and what can the idea of propagation contribute?

Several inference strategies have been implemented in various probabilistic programming systems to date. Some [Goodman et al., 2008, Milch et al., 2005] focus on answering questions by running the randomized process many times and counting the frequencies of events of interest. Such systems must either discard or cleverly avoid runs that disagree with the asserted observations. This discarding is the channel of communication between the evidence and its causes in these systems.

Other probabilistic programming systems, such as [Radul, 2007, Pfeffer, 2001], perform some sort of systematic search over the possible courses the randomized process could follow. Such systems must backtrack when they discover violations of asserted observations. This backtracking is the channel of communication between the evidence and its causes in these systems. In both cases, the channel is fairly narrow: the only thing the evidence can say is "I don't like this state. Try another." That's not very informative.

Propagator networks offer the possibility of a different kind of inference strategy. We have seen how to explicitly track dependencies in Section 4.4, and this gave a new, *dependency-directed* search strategy, similar to dependency-directed backtracking [Stallman and Sussman, 1977, Lieberherr, 1977, Zabih et al., 1987]. We can do the same for probabilistic inference. In fact, the `flip` device can function exactly like the `binary-amb` device, except that it must also remember the prior probability of making either choice. Then we can let the search proceed in a dependency-directed manner, exactly as it would if there were no probabilities involved. The product of the prior probabilities determines the prior probability of any particular consistent answer (the unshaded rectangles in Figure 5-5). The evi-

[2] In fact, probability theory is so useful exactly because it teaches us what the right answers are when reasoning from effects to causes; but that still leaves open the question of how to compute those answers.

dence eliminates some volume of prior probability space as impossible (the shaded region of Figure 5-5). The posterior probability of the consistent answers is normalized to account for that eliminated volume.

If we do probabilistic inference this way, we automatically gain a wider channel for backward communication. Instead of merely saying "I don't like this state," a piece of evidence can at the very least examine the dependencies of its inputs and indicate the choices that led to the bad state. That allows SAT-solving instead of blind enumeration—already a leg up.

Even better, one expects that in many cases the multidirectionality of propagation can be used to even greater effect: sometimes the evidence can lead directly to deductions about its immediate causes, without having to wait for a hypothetical value to come flying in. This is called **evidence pushing** [Pfeffer, 2001]. Since propagation is multidirectional inherently, we need not constrain ourselves to sending interesting computation only down a "forward channel" and using a particular, restricted "backward channel" for the necessary computations that go in other directions—we can compute whatever complex things we want in whatever directions we like.

Interactions

Even though the prior probabilities for how to make each choice are not strictly necessary for doing probabilistic inference by dependency-directed search, they can help guide the search. Specifically, the network can try to arrange to always be considering the most likely as-yet-unexplored set of choices. "Most likely" in this setting can mean either "highest prior probability" or "highest posterior probability given the nogood sets discovered so far."

On the other hand, probabilistic inference is a hairier problem than just search. The reason is that basic search (including the dependency-directed search presented in Section 4.4) is concerned with whether or not some answer is *possible* (i.e., consistent with all the requirements), whereas probabilistic inference asks also how *probable* something is. The difference is that whether a set of choices leads to a possible state depends only on that set of choices (and their direct consequences, and whether those consequences include a contradiction), whereas how probable that state is also depends on whether *other* choices lead to contradictions, because each thing that proves impossible makes everything that is still possible more likely.[3][4]

It may therefore make sense to implement dependency-directed inference by letting various deductions about the probabilities of things propagate too, rather than expecting them to be computed and communicated to the entire network in-

[3]To borrow Arthur Conan Dolye's [Doyle, 1890] immortal words:

> When you have eliminated the impossible, whatever remains, however [*a priori*] improbable, must be the truth.

[4]Complexity theorists would say that search is a SAT problem, while inference is a #SAT problem; would prove that the latter is certainly no easier; and would theorize that in fact it is strictly harder.

stantly the moment a contradiction is discovered. Then there will be two levels of propagation: propagation in the "object world," where the consequences of various premises are worked out and various nogood sets are detected; and propagation in a "strength-of-belief world," where the numerical consequences of those nogood sets on the posterior probabilities of various things flow around and update. The right way to do this is open to further research.

Probability theory has an interesting feature that is absent from the other partial information structures we have studied and will study in this dissertation. That feature is that it comes with its own rules for making decisions. If one has some objective, which is encoded in the effort to maximize some utility function, what is one to do? If one exactly knows the state of world (or at least the part that impinges on one's utility), one is in luck, for one can just pick the action that maximizes said utility. But what if not? What if one's information about the world is incomplete? Probability theory has an answer: maximize *expected* utility (where the expectation is taken over the probability distribution that represents the knowledge one *does* have about the world). Better, one need not always compute even the probability distribution completely: If the only things one does not know are sufficiently unlikely and make sufficiently little difference to one's utility, one can make a perfect decision without even having perfect information about one's knowledge of the world, never mind perfect information about the actual world.[5] This tantalizing character may prove very valuable when building systems that mix partial information with irreversible action.

5.3 Constraint Satisfaction Comes Naturally

The first thing many people think when they hear "propagator" is "constraint satisfaction". Indeed, the present work is closely related to the constraint satisfaction field [Apt, 2003], if subtly different.

Constraint solvers generally do incorporate propagation systems. Those propagation systems are, however, usually very special-purpose, tailored to run under that particular constraint solver, and to propagate the consequences of the particular kinds of constraints that solver deals with. For example, Waltz's work on analyzing line drawings [Waltz, 1972] contained a mechanism for enforcing arc consistency by propagation.

Furthermore, a constraint solver has an additional flow of control beyond propagation, namely a search. For both of those reasons, different constraint solvers don't play very well together, and don't compose and interoperate with other uses of propagation.

We will see in this section that the traditional constraint solver design is not

[5] In fact, one's decisions may be guided not by the demand to strictly maximize the utility function, but simply to ensure that it is above some level (perhaps 80% of the maximum?). In that case, one may be able to save far more computation, and still make decisions that provably follow the rule one has set for oneself.

the only possible one: we can build complete constraint solvers *inside* our general-purpose propagation infrastructure. We will need to use the dependency tracking we learned in Chapter 4 to subsume the search a constraint solver does into the propagation process, but we can do that *without altering the propagation infrastructure*, and without subjecting it to an external control loop that performs a search. Therefore, propagation can remain a common language with which such constraint solvers can talk to each other and to other propagating systems.

Special-purpose though constraint-satisfying propagation systems usually are, a great deal of research, e.g., [Borning, 1981, Konopasek and Jayaraman, 1984, Tack, 2009], has gone into improving them. I expect that much of it can be generalized to the present general-purpose propagation; other work on improving constraint satisfaction can be reused directly in the form of specific, well-worked-out propagators for specific purposes. We discuss some of those advances and their applicability to the present case in the "Importation" paragraph.

Description

Before diving into how we can do constraint satisfaction differently, a brief review of the standard approach is in order. The usual definition of constraint satisfaction is: Let there be some set of variables. For each variable, allow it to take on values from some domain.[6] Further, impose some set of constraints on the possible combinations of the values of the variables. Given all that, the constraint satisfaction problem is to find one (or all) assignment of values to variables, such that each variable really does take on a value from its domain, and such that all the constraints are satisfied; or to deduce that this is impossible.

For example, the popular Sudoku puzzles are readily expressible as constraint satisfaction problems. Each puzzle consists of a nine-by-nine grid with some digits written in, like Figure 5-6. The puzzle is to fill each remaining space in with a digit from 1 to 9 so that each row, each column, and each of the nine three-by-three subgrids has exactly one of each digit. The constraint satisfaction formulation follows: every empty cell is a variable, its domain is the set $\{1, \ldots, 9\}$, and the constraints are twenty-seven `all-different` constraints among nine squares each (some of which are variables, and some of which are already known).

In general, solving a constraint satisfaction problem requires search. One might consider, therefore, the following strategy for solving such a problem: enumerate all possible assignments of values to variables; check each of them for satisfying the constraints; and return those that do. Such a search is, of course, distressingly exponential, rendering this strategy impractical, so the usual approach is to interleave each step of the search with a propagation step, where the constraints are used to prune the search space by narrowing the domains of the variables. To do this, systems keep track of the "current domain" of each variable; use that information

[6]The domains need not be the same across variables; further, they need not necessarily be finite, but of course they do need to be finitely representable (e.g., a union of intervals over the real numbers).

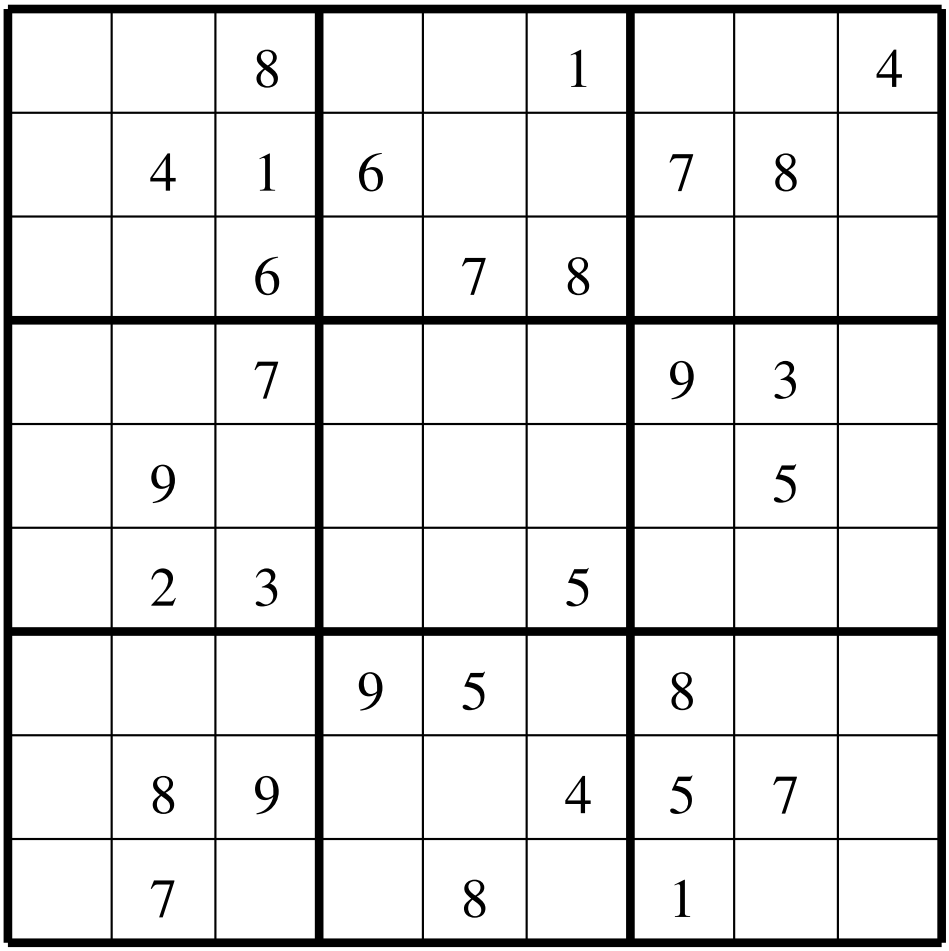


Figure 5-6: A Sudoku puzzle. The task is to complete the grid of squares with digits from 1 through 9 so that each row, each column, and each bold three-by-three square has exactly one of each digit.

in the propagation step to narrow the domains of other variables as much as possible; and then guess further narrowings of the domains in the search, every time propagation reaches a fixed point.

For example, if one were solving Sudoku, one might maintain the set of digits that may, as far as one currently knows, appear in a particular square. Then, of course, any such set can be diminished by removing digits that are already known to appear in any of the squares in the same row, column, or block as the square in question. Such removals may perhaps eliminate all but one of the options a square has; then more can readily be said about the possibilities open to its neighbors.[7] In principle, there could be long chains of deductions of this kind, traveling this

[7]This is just the simplest thing one can do to propagate information about an `all-different` constraint. Many algorithms have been developed, of varying simplicity, speed, and pruning strength; see, e.g., [Puget, 1998].

way and that over the problem space. There is no good way to sequence all such deductions that will occur in a problem before starting to solve the problem, so one needs some form of propagation to deduce all the consequences coherently. But then if it so chanced that one had exhausted one's ability to deduce, one must still fall back on search, to guess something about one of the squares, propagate the consequences of that guess, and backtrack if it leads to a contradiction.

Embedding

How can we build a constraint solver on the present propagation infrastructure? Well, any representation of a variable domain is a good candidate for a partial information structure. Let us represent every variable with a cell. Let us store the current domain of that variable as the partial information structure in that cell. For each constraint, let us make a propagator that narrows the domains of the relevant variables in accordance with the requirements of that constraint.[8] In the case of Sudoku, the partial information structures will be sets of the digits that can appear in the square. The propagators will extract some information of our choosing out of the fact that the numbers in appropriate groups of nine squares are supposed to be different, such as removing the 5 from the sets of values possible for all squares in the same row, column, or block as a known 5.

If we do this, our general-purpose infrastructure will implement the propagation step of a constraint satisfaction system. That much is good; but what about the search? The search is the place where something spectacular happens.

Domain representations are partial information structures. Truth maintenance systems from Section 4.2 are partial information structures that expect to contain partial information structures. What happens if we put the former inside the latter? Then a cell for Sudoku might contain a TMS like

$$\{4,5,8,9\}:A,\ \ \{4,5\}:A,B,\ \ \{2,4,5\}:B,\ \ \{2,4,5,8,9\}:\text{␣}$$

which says "I unconditionally know this square can only contain a 2, 4, 5, 8 or 9; and if I believe A then the 2 becomes impossible; and if I believe B then the 8 and the 9 become impossible." Then we can subsume the search, which is usually considered a process external to the propagation, into an emergent behavior of the propagation itself. Composable modularity strikes again.

Consider the following embedding of a constraint satisfaction problem into this infrastructure: Each variable is again a cell, but now it contains a truth maintenance system over possible "current domains". Each propagator is lifted appropriately to carry around the dependencies of the domains it operates on.[9] Further, attach a pile

[8] In general, there are many propagators that serve, theoretically, to enforce a given constraint. [Tack, 2009] studies their properties and the tradeoffs they expose at great length.

[9] Doing this well, without introducing spurious dependencies, is nontrivial and has great practical consequence. We are not for the present concerned with such details; the point here is that the propagation infrastructure here presented is general enough to implement a constraint satisfier without modification.

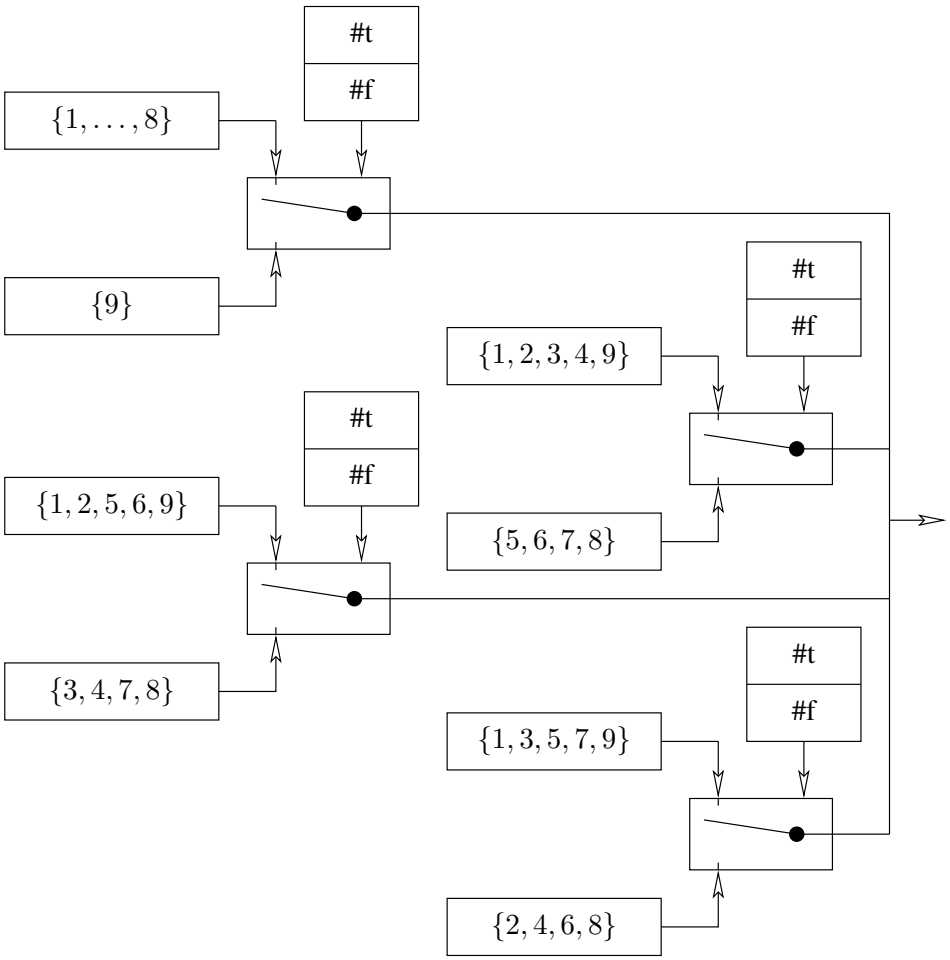


Figure 5-7: A collection of guessers that will guess a value between one and nine by guessing each binary digit separately.

of `binary-amb` propagators controlling a pile of switches[10] and constants, so as to guess all possible values for the variable in question. (For Sudoku, this might look like Figure 5-7.) Finally, claim that empty domains constitute contradictions, so that any set of guesses that leads to some value having an empty domain becomes a nogood.

Then if you run that propagator network, it will eventually settle down into a stable state that constitutes a solution to the constraint satisfaction problem it embodies. The propagators that implement the constraints will keep the system honest, by signalling contradictions and creating nogoods whenever the guesses are inconsistent. The guessers, however, will drive the system to a fully specified solution. The search that used to have to be an external mechanism becomes an emergent

[10]We will describe the `switch` propagator carefully in Section 6.1. Suffice it to say that a `binary-amb` controlling a switch effectively chooses whether to output some specific partial information (as opposed to choosing whether to output `#t` or `#f`).

consequence of guesses being made, nogoods being discovered, and guesses being retracted.[11]

This is power. By generalizing propagation to deal with arbitrary partial information structures, we are able to use it with structures that encode the state of the search as well as the usual domain information. We are consequently able to invert the flow of control between search and propagation: instead of the search being on top and calling the propagation when it needs it, the propagation is on top, and bits of search happen as contradictions are discovered. Even better, the structures that track the search are independent modules that just compose with the structures that track the domain information.

That inversion and modularity has a very nice consequence: by changing the partial information structures that implement the search, we can apply different searches to different problems, or different portions of the same system, while the propagation remains unchanged. In particular, the astute reader will have noticed that the search that emerges from the embedding presented is not the depth-first search that constraint satisfaction systems typically use, but rather a dependency-directed backtracking search. Composing partial information structures has given us a constraint satisfier with a more intelligent search[12] for free.

Implementing constraint satisfaction in a general-purpose propagation system (as opposed to tailoring a special-purpose one for it) also leads to various other generalizations. For instance, the full constraint problem need not be specified in advance: as computations occur, new cells and new propagators can be added to the network dynamically, and so can discover which portion of a potentially infinite constraint problem is actually relevant to the computation at hand. For another instance, since propagation recovers general programming, constraint programs can embed seamlessly into programs meant largely for other purposes—there is no need to explicitly call an external constraint solver. And symmetrically, our constraint solvers now have a chance to interact with and benefit from other uses of propagation.

---

[11] It is probably advantageous to ensure that new guesses are not introduced until after the constraints have all been enforced, as this prunes the space the search will have to explore. The system as presented has no way to ensure that, but the scheduler extensions described in Section 6.4 can allow it.

[12] Actually, the advantages of dependency-directed search are not exactly orthogonal to constraint propagation, but rather *replicate* much of its benefit. In fact, deducing things by propagating constraints can amount to just knowing how to schedule the guesses that a dependency-directed search engine would make so that it finds small nogoods fast. For example, in the case of Sudoku, observing a 5 in one square and deducing that 5 cannot appear in another square on the same row is completely equivalent to guessing 5 for that other square, checking the constraint, and deducing a nogood that bars that 5 henceforth. Not all constraints are like that, but perhaps standard constraint satisfaction lore bears some reexamination in that light.

On the other hand, by introspection, it seems that when I solve puzzles such as Sudoku, I *far* prefer deducing things that are certain to making hypothetical guesses. Is there a good reason for that, or do I just not have very much memory for storing alternate worldviews in my head? Or do I have hardware acceleration for looking for patterns that let me make deductions, but have to deal with hypotheticals and their consequences interpretively?

Importation
A great deal of the work in constraint satisfaction is on making constraint satisfiers fast rather than expanding the scope of the paradigm. As such, it is orthogonal to the present exposition, and some of it can be imported to make the general-purpose infrastructure more efficient.

One major area of research is on how best to schedule propagators, so as to minimize the total amount of work required to produce a solution. It would be interesting to try to generalize that work to general-purpose propagation. To take [Tack, 2009] as an example, Tack's work on propagation strength can be reused together with an informativeness-based scheduler as discussed in Section 6.4; and Tack's work on events and propagation conditions can be reused to improve the accuracy of queueing propagators for reexecution.[13]

Another major area of research in constraint satisfaction is in finding efficient propagators for particular constraints, e.g., [Puget, 1998]. This work can be reused directly, by implementing those propagators for use with the present infrastructure. It may, however, also be profitable to reconsider those algorithms in light of novel partial information types. For example, the infrastructure offers natural dependency-directed backtracking as a possible search, but to take the best advantage of this search, it would be appropriate to track the consequences of each choice very carefully. In the Sudoku example, for instance, it may be worth noting that some square can't have a 4 because of guess A but that it can't have a 5 at all, so that if that is used to deduce the location of the 5 in its row, that deduction will not carry a spurious dependency on guess A. Integrating dependency tracking well may necessitate a reworking of the traditional algorithms from the literature.

5.4 Logic Programming Remains Mysterious

On a high level, this work is similar to logic programming in that both attempt to build complete programming systems whose control flow is implicit. In logic programming, the implicit control flow is in two places: it is in the bidirectional deductions made by unification, and in the search process that attempts various clauses when searching for the refutation of the goal. In this work the implicit

[13] Specifically, the `make-cell` we settled on in Section 3.4 (page 45) uses the finest possible test for whether its content changed or didn't—`(eq? new-content content)`—to decide whether to requeue the propagators observing it. We can imagine instead a system where each propagator can register a procedure indicating whether it should be alerted based on what happened to the cell's content. Passing `(lambda (content new-content) (not (eq? content new-content)))` is equivalent to the default behavior, but many spurious re-propagations could perhaps be saved by using coarser predicates where appropriate. Tack's event systems can be recovered if it turns out that only a handful of specific predicates are needed in a particular region of the network; if more than one propagator registers using the same predicate, they can be stored together and the predicate tested only once. Further, if implications between those predicates are known in advance, more of the tests can be avoided. Tack also observes that performance can be improved if propagators can change their own scheduling conditions and replace themselves with more efficient, specialized versions as domains become more constrained.

control flow is the sequence of propagator activations (and reactivations) as data travels through a network.

In both cases, the implicit control story works well as long as there are no side effects. The introduction of side effects into logic programming systems forces a more specified search order, which in turn threatens to become part of the definition of the method, and undermine the gains of implicit control. The introduction of side effects into propagator systems is an interesting topic for future work; see Section 7.3 for preliminary thoughts on the subject. If the deleterious consequences of side effects can be avoided (or sufficiently mitigated), the present work can be a fresh attempt at implicit control, and therefore a more declarative style of programming.

The partial information lens shows an interesting relationship between differing orders of evaluation in conventional languages and the idea of logic programming. As discussed in Section 6.2.4, applicative order evaluation is the discipline of insisting on complete information about the argument list of a function before invoking it. In that sense, normal order is permitting oneself the freedom to invoke a function with only a certain kind of partial information about the argument list—the list structure itself is known exactly, but each argument may be completely unknown. Then, if the function turns out to need that argument, it waits until that argument is completely determined (and supplied to the function), and then proceeds.

Lazy functional languages like Haskell illustrate that normal order can be useful as a direct modification of "standard programming." I have argued in this dissertation, however, that mergeable partial information is the gateway to multidirectionality. How can we merge the "incomplete argument list" data structure? It's just a bunch of `cons` cells, so it merges by unification. Now that we know that, we can add multidirectionality to our hypothetical normal order language, merging the caller's and callee's information about a function's argument list by unification. The result is very similar to logic programming, in that it has one of the forms of implicit control that logic programming does, namely that the arguments to a function call serve as input or output channels according to what information is available.

Logic programming also contains an aspect of search, for deciding which clause to try when looking to refute a goal. This is a different sort of implicit control; and trying to replicate it on the general-purpose propagator infrastructure is an open avenue for further research. One of the roadblocks is that standard presentations of logic programming make the search chronological, whereas propagators have no single temporal stream on which to base a chronological search. Perhaps a more careful study of abstraction in the present context will yield fruit; perhaps the logic programming search is something like guessing which of several possible alternate function bodies to apply to any given call site.

An obvious difference between traditional logic programming and the present work is that this propagation infrastructure is applicable to all manner of partial information, and is not limited to logical assertions and structural unification. Con-

straint logic programming extends logic programming in similar directions, for instance, to algebraic identities or constraints. Since the essential nature of the search remains basically the same, figuring out how to build logic programming on propagators should yield constraint logic programming for free.

5.5 Functional Reactive Programming Embeds Nicely

Reactive systems like (graphical) user interfaces and interactive animations have traditionally been annoying to write. The essential problem is that the program must be able to respond to events initiated by the user, or the outside world generally. The program cannot predict the sequence in which events will occur, so must be able to react to any event at any time; control must therefore jump around between different "event handlers" at the world's whim, and chaos ensues if the programming environment tries to impose too rigid a notion of time.

Functional Reactive Programming approaches that problem by writing such reactive systems as pure functions from (presumably time-varying) inputs to (hence presumably time-varying) outputs, and letting a runtime update the intermediate and output values as appropriate when inputs change [Elliott and Hudak, 1997, Wan and Hudak, 2000, Nilsson et al., 2002, Cooper, 2008].

For example, if one wanted to animate a ball at the current location of the user's mouse, one might write

```
(draw (ball mouse-position))
```

If `ball` is a function that takes a position and produces a representation of a graphic (of a ball) suitable for rendering on screen, and `mouse-position` is the current position of the mouse pointer (which changes in response to the user moving the mouse), then the job of the reactive system is to automatically update the output of `(ball mouse-position)` when the mouse moves, and invoke appropriate graphics primitives to redraw it.

The wins are everything one would expect from going up a level of abstraction: greater clarity of expression, since one need not explicitly worry about state changes, ease of maintenance, better compositionality, etc. For instance, if one wanted to animate a ball that varied in size as a function of time and followed the mouse around, one could just write

```
(draw (scale (ball mouse-position) (cos time)))
```

and the system would automatically take care of redrawing it both when the mouse moved and when the clock ticked. Note how the programmer is saved the tedium of worrying about the order of whether the next interesting event will be a mouse movement or a clock tick: abstraction, composition.

Implementing functional reactivity

So, we're sold that functional reactive programming is a good idea, and we want to build ourselves a system that does it. We immediately find three desiderata:

1. If only a small number of inputs change at any one time, it would be nice to save effort by not recomputing the values of things that don't depend on them;

2. It would be nice to avoid glitches, which might occur if a fresh partial result is combined with a stale one somewhere; and

3. It would be nice if our implementation were as simple and general as possible.

A first cut at a functional reactive system might be to just rerun the entire inputs-to-outputs function every time any input changes. This will certainly work, it meets the second and third desiderata beautifully, and it's perfectly good for exploring the general functional reactive idea; but it would be nice if we could also save our computers some needless crunching by also meeting the first desideratum.

How can we avoid recomputing the consequences of inputs that didn't change? Since this is a dissertation about propagation, the answer I'm looking for is to propagate. Specifically, to propagate only the consequences of inputs that did change, and leave the unchanged ones alone.

Cooper, under Krishnamurthy, has done some very nice work on implementing functional reactive programming by propagation, in a system called FrTime [Cooper, 2008]. FrTime is built around a custom propagation infrastructure; it nicely achieves both non-recomputation and glitch avoidance, but unfortunately, the propagation system is nontrivially complicated, and specialized for the purpose of supporting functional reactivity. In particular, the FrTime system imposes the invariant that the propagation graph be acyclic, and guarantees that it will execute the propagators in topological-sort order. This simplifies the propagators themselves, but greatly complicates the runtime system, especially because it has to dynamically recompute the sort order when the structure of some portion of the graph changes (as when the predicate of a conditional changes from true to false, and the other branch must now be computed). That complexity, in turn, makes that runtime system unsuitable for other kinds of propagation, and even makes it difficult for other kinds of propagation to interoperate with it.

We will take a different approach. Rather than build a custom propagation engine just for functional reactivity, we will see a way to craft partial information structures to make our existing propagation infrastructure behave in a reactive way. This offers the great advantage of generality: we will achieve reactivity on a medium that can also propagate for other purposes; which will allow our reactive systems to interoperate seamlessly with other systems that can be expressed in the same medium; and in so doing we will generalize functional reactive program-

ming in a way that will give us a solution to the long-standing problem of multiple reactive read-write views for free.

Functional reactivity on general-purpose propagation

So, how do we build a reactive system on the general-purpose propagation infrastructure presented in this dissertation? All the locations are cells, of course, and all the computations are propagators. The question boils down to: What partial information structure captures time-variation the way we want? How should it merge, and how should propagators treat it?

One way suggests itself: Let our partial information structures say "as of time t, the value here is x." Let us merge these structures by letting later times take precedence over earlier times and overwrite them; and let us ask propagators to carry the timestamp from their inputs forward to their outputs.

Now we have a decision to make: what to do if a propagator has inputs with different timestamps? We could just let it do the computation and carry the latest timestamp forward, but that runs the risk of causing glitches. For instance, if somewhere in our program we had something like

```
(< time (+ 1 time))
```

and `time` increased by 1, we could get into trouble if the `<` propagator reran before the `+` propagator. Then `<` would see a fresh value for `time` but a stale value for `(+ 1 time)` and erroneously emit `#f`. FrTime solves this problem by promising to run the propagators in a topologically sorted order, but such a promise would make our scheduler too special-purpose.[14]

A propagator whose inputs have inconsistent timestamps should not compute, but wait until its other inputs refresh.[15] Unfortunately, this policy causes a different problem: if some user input changes and we write the new value with an updated timestamp, we can't just leave a (different) unchanged input be, because we have to update its timestamp to tell the propagators that it is still fresh. Sadly, that causes a cascade of timestamp updates; and while this approach does work, it amounts to recomputing the whole input-to-output function every time any input changes, just to verify that the timestamps are all current.

So what do we have? We have an approach that works. The system produces no glitches, because the explicit timestamps on every value prevent inconsistent values from different times from being used for the same computation. When the network has quiesced at each time step, the outputs correctly reflect the inputs for that time step. Unfortunately, we have not met the desideratum of non-recomputation. This

[14]We could perhaps fake it with partial information structures that stored the "height" of a particular cell in a particular network, but we will attack this differently in the present exposition.

[15]Unless it knows better: we can build accumulators like the output of an `integral` propagator by judiciously violating this rule, to wit letting the `integral` propagator absorb a `dx` input stamped with the current time into an accumulated output stamped with an earlier time (and overwrite the latter with a result stamped with the current time).

is because a value is considered stale if it does not bear the latest timestamp, and this is too strict a freshness requirement.

Avoiding recomputation by tracking dependencies

Can we do better? Yes, we can. One way is with a variation on the dependency tracking of Chapter 4. Give each time-varying input an identity and use per-input timestamps. That is, at each intermediate value, instead of just tracking a global timestamp, track which inputs, read at which times, led to it. For example, the `time` input cell might contain a partial information structure that says: "The value here is 17, because of the `time` input read at time 17." Then the cell for the value of the expression `(+ 1 time)` can contain a structure that says "The value here is 18, because of the `time` input read at time 17."

Now the rule is that a value is fresh if all the timestamps for all the inputs required for *that value* are at their latest, regardless of whether other inputs have changed since. For instance, after a clock tick, the + propagator will want to add "The value here is 19, because of the `time` input read at time 18" to a cell that already has "18 because of `time` at 17"; but "`time` at 18" proves that "`time` at 17" was stale, so the old 18 is forgotten and replaced with the new 19.

As before, a propagator whose immediate inputs are supported by an inconsistent set of user inputs should wait and do nothing, until the consequences of the update that caused the inconsistency propagate all the way. So the < in our example might see "18 because of `time` at 18" in one input and "18 because of `time` at 17" in the other input. Rather than computing that 18 is not less than 18 and producing a glitch, it should notice that one of its inputs promises that "`time` at 17" is stale, and so it should wait until the other input becomes fresh. On the other hand, if the mouse hasn't moved since time 2, then "`mouse-position` at time 2" will still count as fresh, and a computation depending on both the `mouse-position` and the `time` will be able to proceed, without having to wait for the `mouse-position` timestamp to catch up to the `time` timestamp.[16]

So what do we have now? Now, if one input changes but another does not, we need only update the new one, because the old one remains fresh even if we leave it alone. Therefore, computations that only depended on the old input do not need to be redone—we have achieved, at least to some degree, desideratum 1, without giving up too much of desideratum 2, and fully meeting desideratum 3 by reusing a general-purpose propagation system.

---

[16]There are of course more details to think about. If values are supported by multiple inputs, it may be that updates travel at different speeds and it turns out that two sets of supports prove each other stale (e.g., foo is due to A at 2 and B at 1, but bar is due to A at 1 and B at 2). Perhaps an explicit `stale` marker is in order? Also, how to merge values that are supported by different inputs at different times (like foo due to A at 4 and bar due to B at 2)? Especially if they contradict each other? But these are questions of policy. Which policy is good for what purpose, and how to intermingle them, is an avenue for future research; I believe the basic strategy is sound.

Granularity of identity matters

This approach to propagating reactive updates carries a different set of tradeoffs than the approach taken in FrTime. On the one hand, there is potential for glitches: if two inputs with different identities change at the same time, the consequences of one may reach a place before the consequences of the other, and that propagator may produce a value that depends on a set of inputs that never actually occurred in the real world. For example, this might happen if we wanted to represent the position of the mouse as two separate inputs `mouse-x` and `mouse-y`, rather than one compound input `mouse-position`: if the user moved the mouse diagonally, the consequences of, say, the x-coordinate might reach some propagator before the consequences of the y-coordinate. Then that propagator would be perfectly content to produce some result that depended on "`mouse-x` at 42" and "`mouse-y` at 33", which may represent a position where the mouse never went. The consequences of "`mouse-y` at 42" would eventually catch up and overwrite, but the situation would still constitute a glitch. This would look like moving diagonally by moving across first and then up.

This kind of glitch can be avoided with coarser input identities: if both the inputs in the above example are actually seen as aspects of the same input, and share an identity, then instead of producing the glitch, that propagator would have waited for all the consequences to reach it. But coarsening the premises costs more recomputation in principle: if one aspect can change even though the other does not, the updated timestamp will still need to be carried through to ensure consistency. For instance, if the user really did move the mouse exactly vertically, then separating `mouse-x` from `mouse-y` might have saved us the trouble of recomputing the consequences of `mouse-x`, such as a shadow from the pointer's position. Collapsing those into a single `mouse-position` forces us to update the timestamp on that shadow even though it's still the same. The extreme in this direction is to have all inputs share the same identity—this corresponds to the original scheme of using just one global timestamp. So here we have a tradeoff FrTime did not need to have: as we make premises finer, separating inputs from each other, we recompute fewer things, but start to risk more glitches in places where the consequences of different inputs converge.

We buy something for the price of this tradeoff, however: the basic propagation infrastructure did not have to be changed to suit our purpose, and therefore such a reactive system can compose with other uses for propagation. For example, we could build a reactive constraint solver: if there is some system of constraints with some time-varying inputs, the propagation for updating those inputs will interoperate seamlessly with the propagation for solving those constraints. Or we could build a reactive system with multiple read-write views of the same data. Such a thing is hard to do normally because it is essentially multidirectional, but it sits beautifully inside the essential multidirectionality of propagation.

For example, suppose we were trying to produce a color selection widget. It should of course allow the user to select their colors either as red-green-blue or

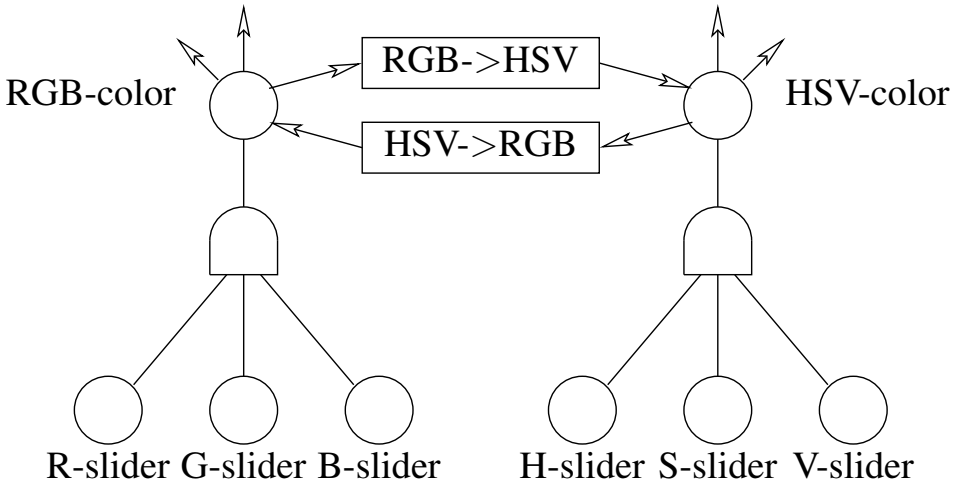


Figure 5-8: A network for a widget for RGB and HSV color selection. Traditional functional reactive systems have qualms about the circularity, but general-purpose propagation handles it automatically.

as hue-saturation-value as they please, and ideally update both views to show the current selection. The natural network to draw for this is in Figure 5-8; observe the converter circularity between the RGB cell and the HSV cell. That circularity reflects the property we wanted, that whichever input the user updates, the other will get modified accordingly. Unfortunately, traditional functional reactive systems have a hard time with this: there is no one function with distinct inputs and outputs that implements both conversions and picks which to do based on the input. FrTime, for instance, would not like this, because the update graph would not be acyclic.

The present propagation infrastructure, however, handles this RGB-HSV example seamlessly. If we just say that all six of the inputs share the same identity `color`, but ask the GUI toolkit to update the timestamps for only the RGB or HSV triplet, rather than all six of them, when the user moves one, we will achieve exactly the desired effect. If the user moves an RGB slider at time 2, data supported by "`color` at time 2" will start on the RGB side and propagate over to the HSV side, superseding all in its path. If the user then moves an HSV slider at time 5, data supported by "`color` at time 5" will start at the HSV side, and propagate over to the RGB side, proclaiming the staleness of the old RGB values and superseding *them* in their turn.

More interactions

Building reactive systems on top of a common propagation infrastructure can also have another benefit. The "Reactive" part of Functional Reactive Programming messes horribly with the flow of time. This confuses side-effects immensely, and,

because people don't want to think about that pain, the discipline is "Functional".[17] If, however, the ideas suggested in Section 7.3 can lead to a decent general story for side-effects in propagator networks, then perhaps reactive systems with a decent side-effect story can be built on top of those principles.

Finally, functional reactive programming research can shed valuable light on how to build general propagation systems that model time in the world; and in particular may offer a coherent story about accepting input. Imperative programming models time in the world by identifying it with time in the computer. That proves excessively burdensome when the computer needs to do things whose order does not have a natural correspondence with the world's time. Functional reactive programming advocates a different way to model time in the world, that in principle imposes fewer restrictions on the computer's internal behavior.

This dissertation, in contrast, makes no mention, barring a brief discussion in Section 6.7, of how a propagator network should accept input from the outside world; doing that well is left to future work. The prototype presented here accepts input by means of the user interacting with the Scheme read-eval-print loop in between runs of the propagator network's scheduler. A complete propagation-based programming system may therefore want to borrow ideas on how to model time from the functional reactivity community.

5.6 Rule-based Systems Have a Special Topology

A traditional rule based system [Hayes-Roth, 1985] (or blackboard system, as they are sometimes called [Erman et al., 1980]) is also an instance of a propagation process. In this case, the blackboard, or the assertion database, acts like a single cell, and each rule is a propagator that reads that cell and potentially adds stuff to it. The network structure looks like Figure 5-9. The partial information structure in this one cell is a big thing containing all the knowledge the rule system has discovered so far. It has the interesting feature, unlike other partial information systems discussed in this dissertation, that it not only can but is expected to grow without bounds, collecting ever more information.

By this analogy, a more general propagator network is like a blackboard system with a large number of blackboards that are segregated by topic, and each "rule" (i.e., propagator) registers to observe only those blackboards that it is interested in, and presumably only writes observations on a limited set of blackboards that are about things it can deduce something about. This decomposition can improve the performance of such a system by saving rules from having to search through large piles of assertions that predictably do not concern them, and, in principle, by distributing reasoning about disparate topics across processes or computers.

[17] With the addendum that drawing something on the screen is no longer a side effect, because after all, you can redraw it later as a reaction to something else. But "real" side-effects, like sending packets over the network, are not, in general, seamlessly integrated into functional reactive systems.

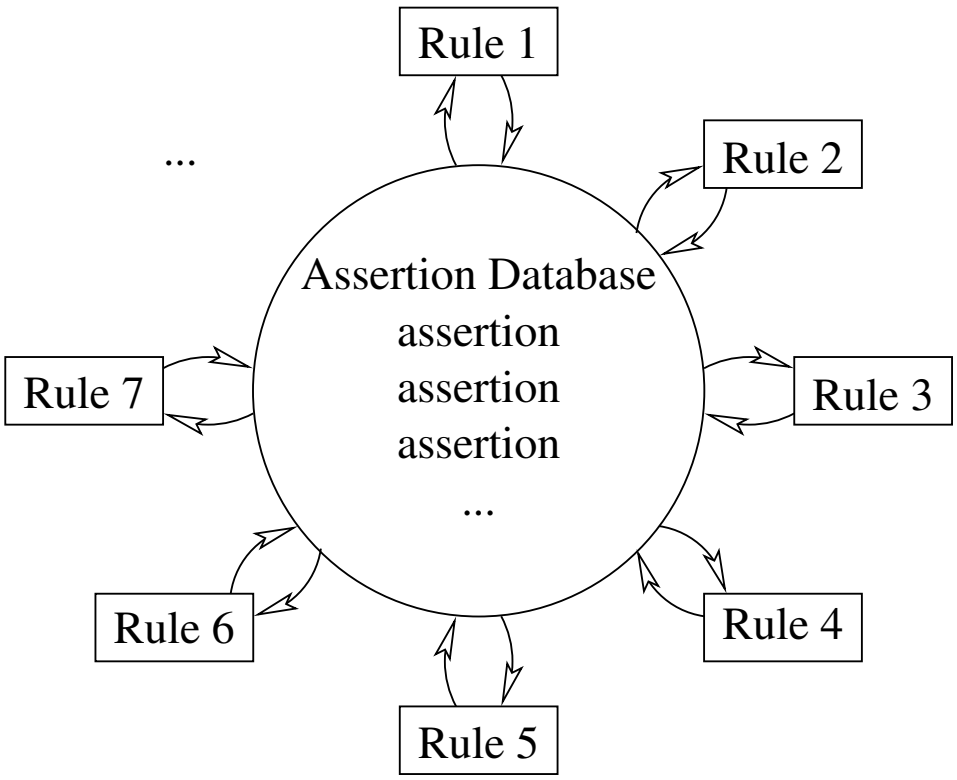


Figure 5-9: A rule-based system viewed as a propagator network looks like a big cell with a bunch of little propagators reading and writing it.

5.7 Type Inference Looks Like Propagation Too

Type inference can be formulated as generating and then solving constraints (on variables that range over the possible types) [Sulzmann and Stuckey, 2007]. These constraints can therefore be solved by a propagation process. Since the constraints tend to be sparse, because they come from a substrate which has a topology and local features, propagation is a very natural way to think about the process of solving them.

Consider as the simplest example type inference over simply-typed λ-calculus. Our calculus has

- constants with known types, like `#f`, `#t`, `0`, `1`, ..., `"mumble"`, ...[18]
- variables, like `f`, `g`, `x`, `y`, ...
- function applications like `(f 8)`, and
- function definitions like `(λ (x) (increment (double x)))` (which bind variables).

[18]Which can be treated as variables with known fixed bindings, if one seeks uniformity.

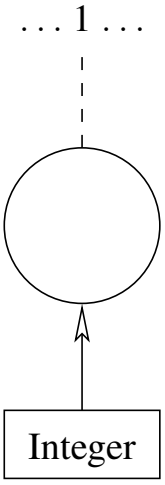


Figure 5-10: A type inference network fragment for a constant

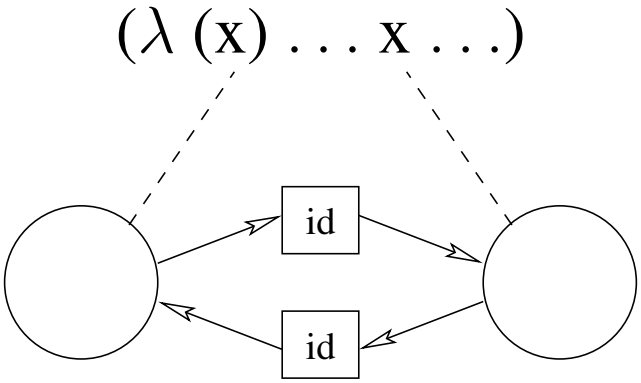


Figure 5-11: A type inference network fragment for a variable reference

We can write this type inference problem down as a propagator network as follows: Let there be a cell for every variable binding and for every expression. This cell holds (everything we know about) the type of that variable or the type of the object produced by that expression. Then add propagators for every occurrence of every construct of the language:

- For every constant expression, add a propagator that writes the type of that constant into the cell for that expression, per Figure 5-10.
- For every variable reference, add a pair of identity propagators between (the cell for the return type of) that variable reference and (the cell for the type of) the binding of that variable, per Figure 5-11. These identity propagators will copy everything we know about either type into the other.[19]
- For every function abstraction, add a propagator triplet that enforces the function definition rules, per Figure 5-12:
  - One writes (everything we know about) the argument type of the function into (the cell for) the type of the variable being bound;
  - One writes (everything we know about) the type of the function return into (the cell for) the type of the function's body; and

[19]This causes some interesting limitations in the presence of higher-order functions. The technical term for these limitations is that this type system is **monomorphic**. The literature has much to say about type systems that do not suffer from these restrictions, but they are too complex to think about for this example.

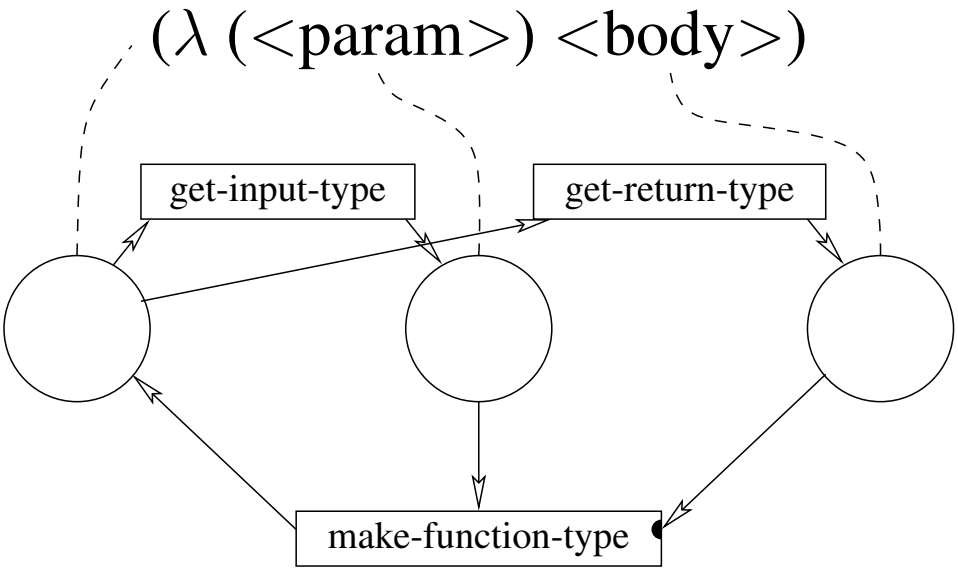


Figure 5-12: A type inference network fragment for a function definition

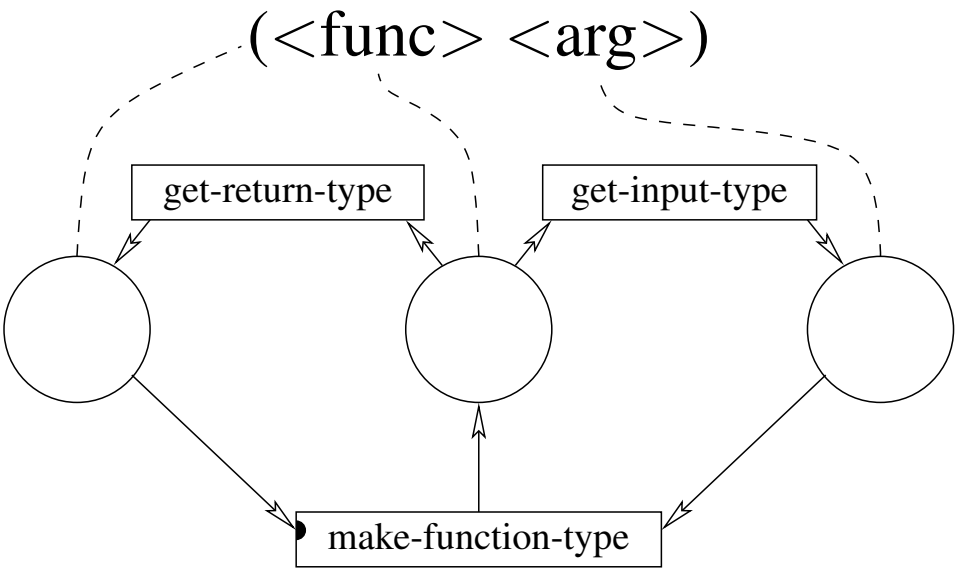


Figure 5-13: A type inference network fragment for a function application

  - One writes the function type derived from (everything we know about) the type of the bound argument and the type of the function body into (the cell for) the type of the function being abstracted.

- Finally, for every function application, add a propagator triplet (much like the previous one) that enforces the function application rules, per Figure 5-13:
  - One writes (everything we know about) the argument type of the func-

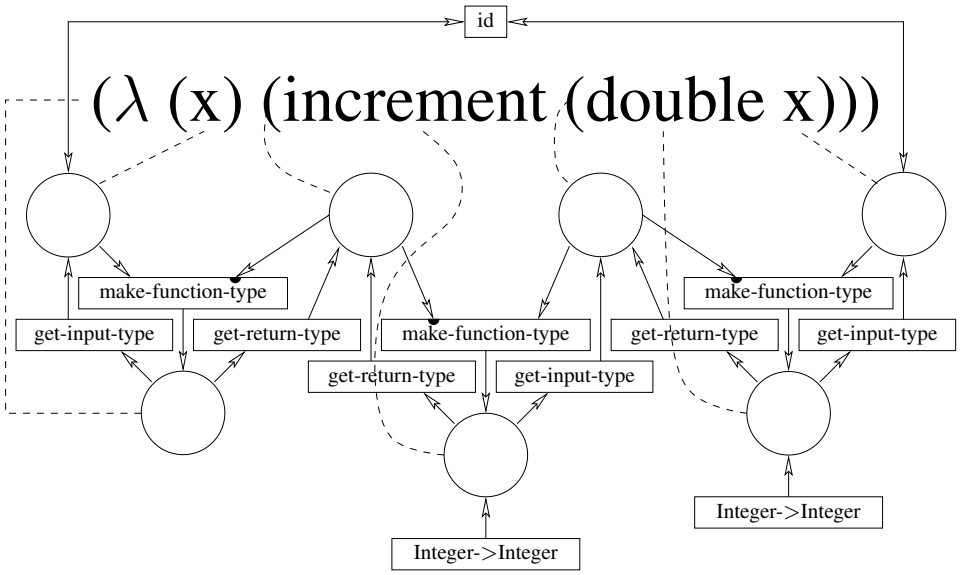


Figure 5-14: A complete sample type inference network

tion into (the cell for) the return type of the argument expression;

- One writes (everything we know about) the return type of the function into (the cell for) the return type of the application; and
- One writes the function type derived from (everything we know about) the type of the argument and the type of the return into (the cell for) the type of the function being applied.

For the simply-typed λ-calculus, equipping the cells with a simple partial information structure suffices. Specifically, each cell can hold either `nothing`, meaning that we don't know the type that goes there yet, or a primitive type like `Integer` or `Boolean`, or a compound function type like `Integer -> Boolean` or `Integer -> nothing` (the latter meaning that the function takes an integer but we don't yet know what it returns). To get it right, function types have to be treated like compounds per Section 6.3, and therefore merged by structural recursion.

Such a machine will be able to perform simple type inferences without any further elaboration. For example, a network constructed to infer the type of `(λ (x) (increment (double x)))` is shown in Figure 5-14 (assuming `increment` and `double` are both known to be functions from integers to integers). The progress of inference in that network is shown in Figures 5-15 through 5-18.

More elaborate examples will of course require more elaborate structures. For example, typing the identity function will need structures that can maintain type variables, and propagators that can hypothesize them when needed, and push them around to see what consequences they have.

Some standard descriptions of type inference cast it as a matter of producing a bunch of type equations first and then solving them all in one fell swoop, using

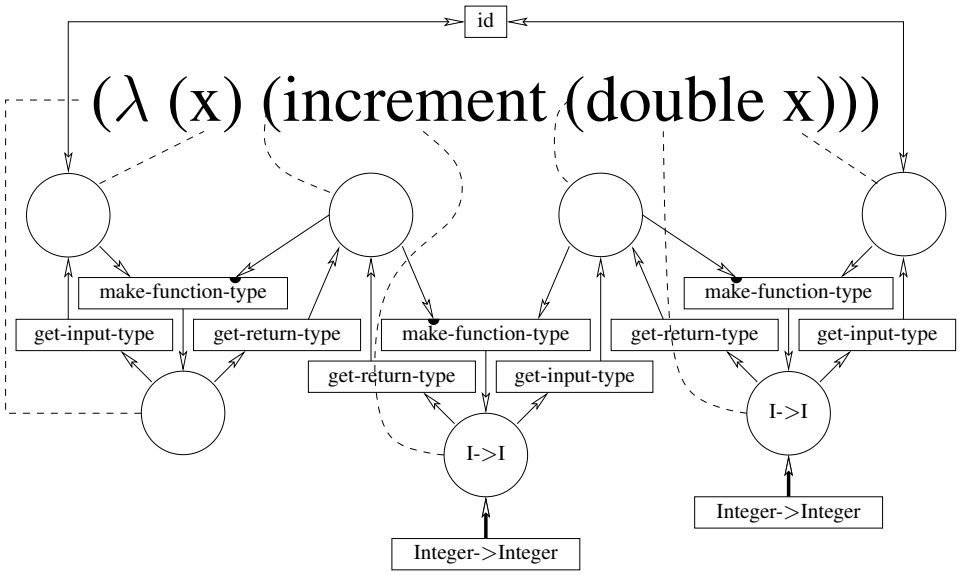


Figure 5-15: Sample type inference 1: The known types of constants are filled in.

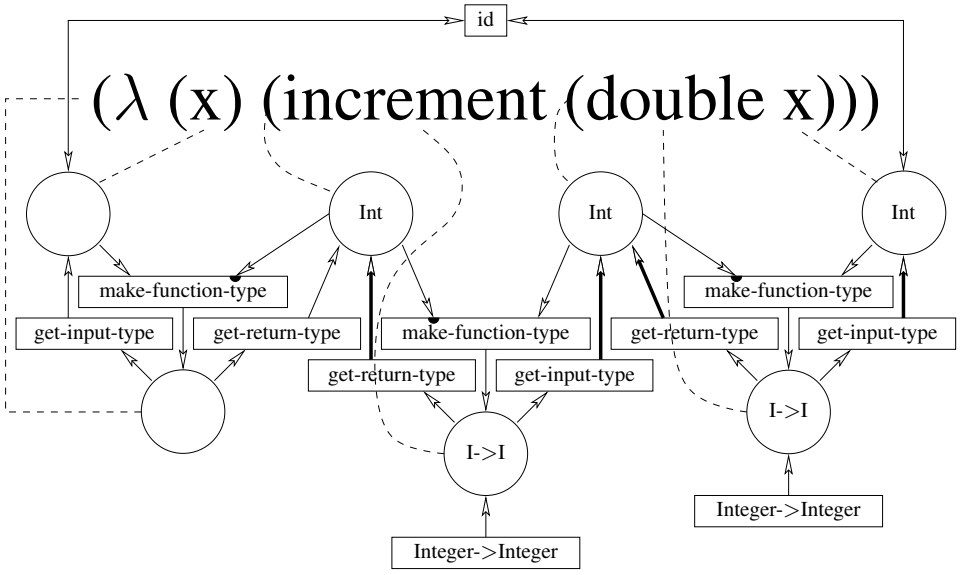


Figure 5-16: Sample type inference 2: The types of several expressions can be deduced immediately from the types of the functions that produce or consume them.

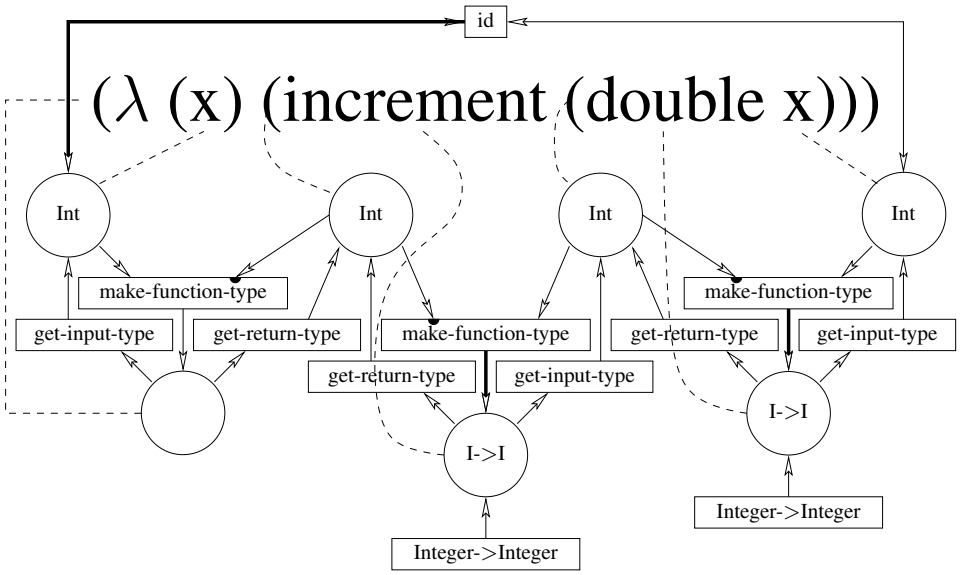


Figure 5-17: Sample type inference 3: The type of the parameter is inferred from how that parameter is used.

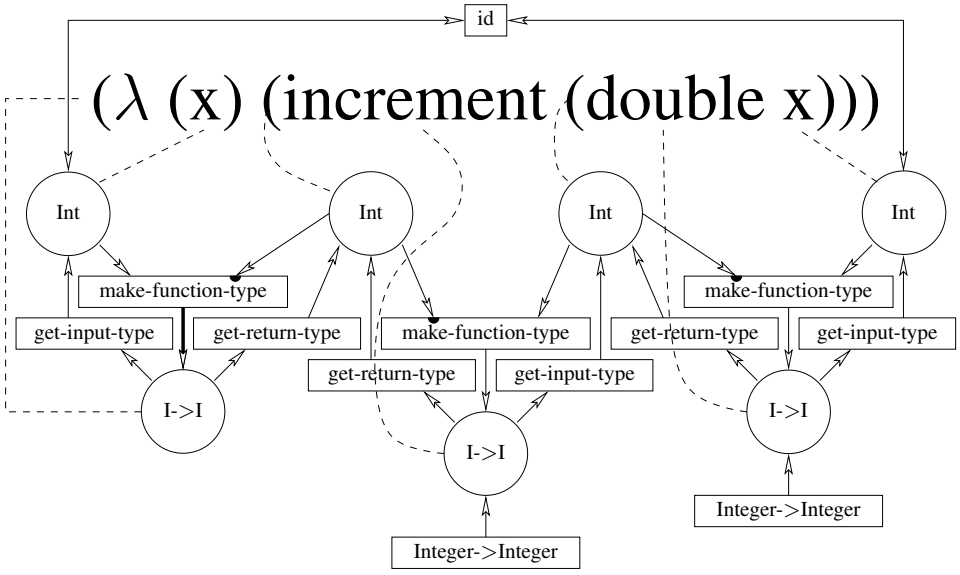


Figure 5-18: Sample type inference 4: The type of the overall term can be deduced from its definition.

a black-box constraint solver. However, as can be seen from this example, the equations are very local, and therefore sparse. Using sparse equations to define a propagator network like this helps emphasize their sparseness, and clarify the way information flows through them during the process of solution.[20]

One possible benefit of seeing this flow of information is that it is natural to think about layering, say, provenance information (like we did in Section 4.1) onto the objects the network passes around. Such provenance decorations can then be used to inspect the final solutions to the equations, and indicate, for example, which places in a program led to the type deductions the system made. In particular, such a mechanism may be helpful for generating good error messages for type inference systems.

That type inference is itself multidirectional we have already seen. But intriguingly, it can lend multidirectionality to an otherwise unidirectional program. Specifically, if which method of a generic function is invoked can depend on the inferred type needed for its output, then, in effect, information can flow "backwards" through the function-return channel. Typeclasses in Haskell [Hudak et al., 1999], for instance, have this character. As a particular example, the signature of the `return` function specified in the `Monad` class includes the particular monad only in its return type: such code as `return 1` has to rely on the inferred type information about its desired result to decide whether to produce `[1]` for the `List` monad, `Just 1` for the `Maybe` monad, or something else for some other monad. If one wrote `(return 1)` in Scheme, however, one would always get the same result, or one would not be able to claim `return` as a pure function.[21]

Propagation networks are inherently multidirectional—indeed, multidirectionality is the point—so they can seamlessly integrate this kind of producer-consumer communication without needing to rely on an external mechanism such as type inference. And of course, the only reason for type inference to be external to a normal program is its multidirectionality. A program running on a propagation

---

[20] Of course, not all systems of even sparse equations can reliably be solved by propagation alone. Sometimes the systems really have larger-scale dependencies, which show up as loops in the network structure. These larger scale equations can be discovered by propagating a symbolic variable around the network, and seeing where it meets itself, as we have done with "Boolean variables", in the form of premises, in Section 4.4.

Sending variables around the loops of a network requires a reliable way to generate unique variables, which is to say a means of giving things identities. That is a nontrivial problem in distributed systems, but at least everything else about this strategy is completely local. Furthermore, the identities don't have to be completely global: they need only be unique within the travel distance of the objects they identify — two variables that will never meet need not have distinguishable identities.

Having to solve systems of dense equations is inescapable in general, but propagation can drastically reduce the number of equations that must be solved simultaneously. In particular, propagation over a dynamically growable network can be used to solve the relevant portion of a theoretically infinite set of simultaneous equations, which might perhaps be impossible to solve by first writing all the equations out and then solving them all as a simultaneous system.

[21] One could actually pull this stunt off in Scheme too, if one modified the Scheme to track some type information about the value a continuation expects, and then allowed a generic function mechanism to read that type information during dispatch. But no Scheme implementation I know of does this.

infrastructure could incrementally, dynamically infer more information about the types flowing through it as relevant inputs became available, perhaps refining a compiler-computed type analysis at runtime.[22]

Type inference is also only one of the things compilers do that can look like propagation. Flow analysis [Shivers, 1991] comes to mind as also having the character of being able to gather information about what can happen to an object, and which objects might flow to a place, from multiple sources within the program being analyzed. Polyvariant flow analysis, in particular, needs to be able to dynamically add new cells (contexts, in their terminology) as it goes, because the number of different places a function might flow to, and therefore the number of different cells needed for information about what might flow into its input, is only discovered as the analysis progresses.

So, in perfect analogy with the situation for types, propagation might be able to offer both an elegant substrate for implementing compiler stages such as flow analysis, and the opportunity to embed such stages more readily into running programs. For instance, once a closure has been constructed by a running program, the bindings it closes over are bound to known values, which may not have been known at the time the closure was originally compiled. Perhaps, if that closure is to be called many times, it may be worth the program's time to refine any compiler analyzes that have been done on it, bearing the now known bindings in the closure's environment in mind.

At the beginning of this chapter I promised to show you the expressive power of propagation. We have considered seven examples, each of which has interested computer scientists over the years, and we found that propagation is a suitable substrate on which to express those ideas. Having therefore concluded our survey of the expressive power of propagation, we are ready to move on to building a complete programming system out of it.

[22]Type-safe `eval`? Interleaving type checking and macro expansion? Anyone have any other ideas?

Chapter 6

Towards a Programming Language

CHAPTERS prior to this one have given us a chance to consider propagation as a model of computation. We built a basic propagation system in Chapter 3, illustrated its flexibility by implementing dependency tracing, multiple worldviews, and good backtracking in Chapter 4, and discussed how it unifies and generalizes related models of computation in Chapter 5. It is now time to explore several hitherto unaddressed issues that arise in trying to build a complete programming system (programming language or virtual machine, if you will) based on the propagation model.

6.1 Conditionals Just Work

Every programming system has some notion of a conditional, and propagation is no exception. The basic conditional propagation construct is the switch, drawn in Figure 6-1. This propagator either passes its input to its output, if the signal on the control is `true`, or not.

The switch propagator differs from the familiar `if` expression because `switch` propagates conditionally, whereas `if` evaluates conditionally. In particular, in a world of always-on machines, the switch propagator has no effect on the propagators that compute its input, whereas `if` does determine which of its branches is evaluated. This will have consequences for how we arrange recursive propagator networks in Section 6.2.

The basic switch propagator for the all-or-nothing information type is easy

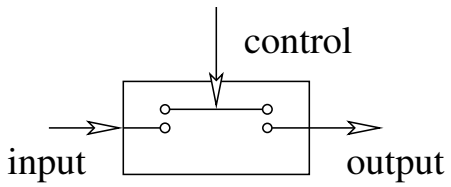


Figure 6-1: Conventional diagram of a `switch`

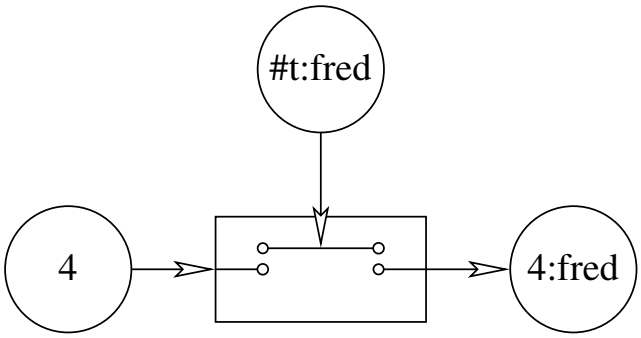


Figure 6-2: A `switch` should pass on the dependencies of its control. If we have to believe `fred` to know that the control of the `switch` is `#t`, then we also have to believe `fred` to know that the output of the `switch` is 4, even though the input may be 4 unconditionally.

enough to implement:

```
(define switch
  (function->propagator-constructor
   (lambda (control input)
     (if (nothing? control)
         nothing
         (if control input nothing)))))
```

or even

```
(define switch
  (function->propagator-constructor
   (handling-nothings
    (lambda (control input)
      (if control input nothing)))))
```

Note how the `switch` does not propagate its input if the control is unknown (i.e., `nothing`). This is a consequence of interpreting `nothing` as an information state: if a switch knows nothing about its control, it in particular knows nothing about the circumstances under which its control might be true; therefore it can say nothing about its output.

The reasoning about what `switch` must do if its control is `nothing` generalizes to other forms of partial information. For instance, if the control of a switch is a justified `#t`, then the `switch`'s output should depend on the justification of the control (as well as of the input); for example as in Figure 6-2. Fortunately, the generic machinery constructed in Appendix A.5.1 does the right thing:[1]

[1]This is not an accident; in fact, getting `switch` and `car` (Section 6.3) right was the main motivation for writing that code that way. Appendix A.5 describes how it was done and why it is interesting, but suffice it to say here that a single uniform treatment of partial information types does what we want for `+`, `switch` and `car`.

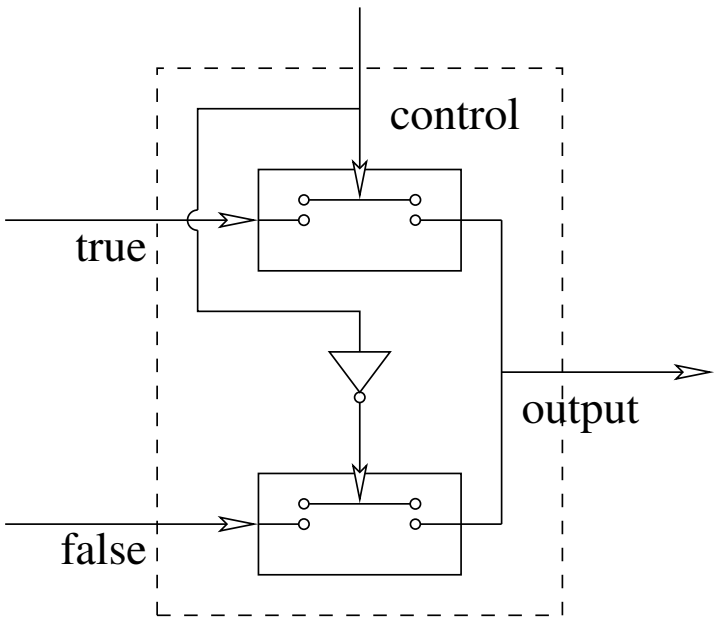


Figure 6-3: A `conditional` made out of `switches`

```
(define switch
  (function->propagator-constructor
   (nary-unpacking
    (lambda (control input)
      (if control input nothing)))))

(define input (make-cell))
(define control (make-cell))
(define output (make-cell))
(switch control input output)

(add-content input 4)
(add-content control (supported #t '(fred)))
(content output)
#(supported 4 (fred))
```

A more direct analogue of the usual `if` would have two inputs, corresponding to the two arms of the `if`. We do not need such an object as a primitive propagator, because it is easy enough to assemble out of `switches` (and a logical inverter):

```
(define (conditional control if-true if-false output)
  (let ((not-control (make-cell)))
    (inverter control not-control)
    (switch control if-true output)
    (switch not-control if-false output)))
```

This wiring diagram looks like Figure 6-3. This compound is common enough to have a picture of its own, in Figure 6-4.

Among propagators, a conditional with two inputs and one output is no more special than one with one input and two outputs, that routes that input to one or the

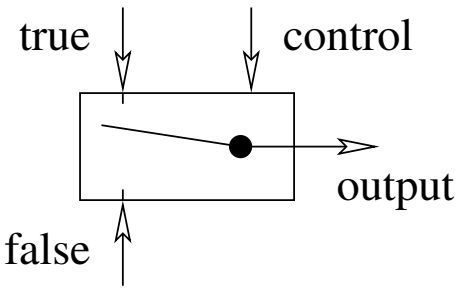


Figure 6-4: Conventional diagram of a `conditional`

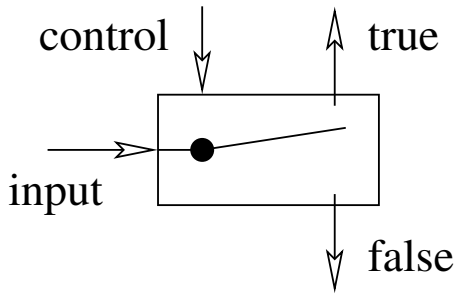


Figure 6-5: Conventional diagram of a `conditional-writer`

other output depending on the control. Such a machine, drawn like Figure 6-5, is also easy to build out of switches. Indeed, the code is dual, and the wiring diagram, in Figure 6-6, is symmetric.

```
(define (conditional-writer control input if-true if-false)
  (let ((not-control (make-cell)))
    (inverter control not-control)
    (switch control input if-true)
    (switch not-control input if-false)))
```

The beautiful symmetry of this conditional with the other, together with the fact that `switch` is a special case of each, forms strong evidence that `switch` is the right primitive propagator for conditional propagation.

To one steeped in the ways of Scheme, where `if` is a magical creature that must be treated with the greatest care, it was quite a surprise that `switch` was a better primitive than the two-armed `conditional`, and that its implementation was just a function (even one that was ok to wrap in `nary-unpacking`!)

One of the reasons this implementation of `conditional` works is that when the `output` cell merges the value that comes in from the selected branch with the `nothing` that comes in from the unselected branch, the value always subsumes the `nothing`. So while the `switch` propagators conditionally turn one or the other incoming value into `nothing`, the job of actually converting the two answers from the two branches into one is done by the merge that happens in the `output` cell. In

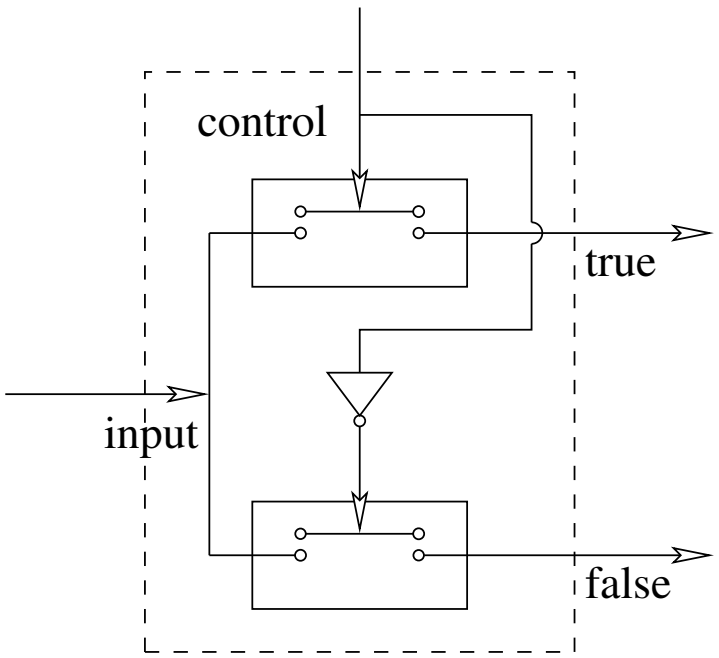


Figure 6-6: A `conditional-writer` made out of `switches`. Note the symmetry with the `conditional` in Figure 6-3.

effect, one of the things that was a little magic about `if` in expression languages comes out of there already being an `if` inside every cell's merge.

You may be wondering by now how in the world we are going to do recursion if all our propagators, including `switch`, aggressively run their computations whenever they can. The glib answer is: by denying them inputs, so they can't. The careful description awaits in Section 6.2, but the effect is that we will make the abstraction mechanism itself responsible for deciding whether to actually expand a recursive call, instead of relying on a magical property of a surrounding expression. Like relying on an `if` buried inside the cell's merge, this relies on an `if` buried inside the abstraction mechanism; the magic that used to be associated with `if` has moved.

6.2 There are Many Possible Means of Abstraction

Every language must have primitives, means of combination, and means of abstraction. The primitives in the propagator network are cells and primitive propagators. The means of combination is wiring them together into larger propagator networks. What are the means of abstraction?

The job of an abstraction mechanism is to package up a propagator network and give it a name, so that it may be treated as a primitive to build larger combinations that aren't interested in its internal structure. How can this job be done?

6.2.1 Compound blueprints

One means of abstraction is inherited directly from the host Scheme. The primitive propagators are represented as Scheme procedures that act as blueprints: call it on a collection of neighbor cells, and it will attach a copy of the appropriate propagator to those cells, and schedule it. One can therefore abstract the blueprints: write Scheme procedures that construct entire subnetworks rather than just individual primitives. For example, we built sum constraints out of primitive adders and subtractors thus:

```
(define (sum a b total)
  (adder a b total)                    ; a + b -> total
  (subtractor total a b)               ; total - a -> b
  (subtractor total b a))              ; total - b -> a
```

From the perspective of the network, this acts like a macro system because the compound blueprints are expanded when the network is constructed, and in particular before the scheduler runs.

One advantage of abstraction by compound blueprints is that since it is inherited from the host Scheme, no work needs to be done to implement this abstraction mechanism. Perhaps more importantly for the author of a document, no work needs to be done to introduce it to the reader—in fact, I have been using it freely in this dissertation so far without, I hope, generating any confusion.

Abstraction by compound blueprints also enjoys all the advantages of a macro system. In particular, it introduces no runtime overhead, because all such abstractions are gone by the time the network is started. Moreover, if there is any computation that can be done in a compound blueprint rather than in the network itself, it is of course done before the network's runtime as well.

Being a macro system is a sword with two edges, though. Since compound blueprints are fully expanded before the network is run, they have no access to the data that the network actually computes. As a consequence, this abstraction mechanism does not suffice to implement, for example, recursion: That requires a notionally infinite propagator network, whose structure is elaborated only as the computation being done actually needs it, but compound blueprints have no way to know what will be actually needed and what will not.

6.2.2 Delayed blueprints

If one wants to implement recursion with some abstraction mechanism like compound blueprints, but is stymied by the fact that they will fully expand before the network is run, the natural thing to do is to delay them. In other words, define a propagator that, when run, will expand some compound blueprint. That compound blueprint can then presumably refer to more such propagators that delay more compound blueprints, but those will not be expanded until the computation needs them. One way to do this is as follows:

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

This code presumes that the blueprint (here `to-build`) is already closed over the appropriate cells, and can therefore be called with no arguments. It also presumes that an abstraction will not do anything interesting if all of its neighbor cells contain `nothing`; we will consider how wise a presumption that is in Section 6.2.4.

The delayed blueprint approach makes the shape of the network reflect the shape of the execution of a program: every time an abstraction is invoked, a copy of its network structure is constructed at the invocation site. Since the invocation site is constructed afresh every time the abstraction containing *it* is invoked, these constructed copies are not reused for later invocations (but the blueprint that generated them is).

One advantage of this strategy is that the contents of individual cells don't need to know that they are participating in an abstraction: cells can continue to contain whatever style of partial information they would contain if the network in question were built by hand. Another advantage is that propagators that build propagators are almost certainly a very useful concept anyway,[2] so perhaps doing abstraction this way does not require any additional machinery.

One consequence of this strategy is that, since the `to-build` procedure gets called during the execution of the network, it can examine the contents of its neighbor cells when deciding what propagator network to build. This creates a potentially thorny problem, because if the interior of an abstraction can depend upon information that is only partial, then it seems that one must have a notion of partial information about the resulting network structure. Indeed, the structure of the network (and the contents of the scheduler's queue) is a separate repository of state from the cells. The main thesis of this dissertation is that we derive great advantage from making the information stored in cells conceptually partial, and from giving the user control of the knowledge representation used. It then stands to reason that making the information contained in the network's connections and in the scheduler's queue partial as well would likewise yield advantage. Propagators that build propagators based on information stored in cells are a direct way that information stored in these two domains interacts, bringing the issue of partialness of network connections to a head. Unfortunately, we have no good story about partial information about network structure. On the other hand, this problem can be avoided by imposing on `to-build` procedures the discipline that they not examine the contents

[2]Though we have not yet found any other use for them...

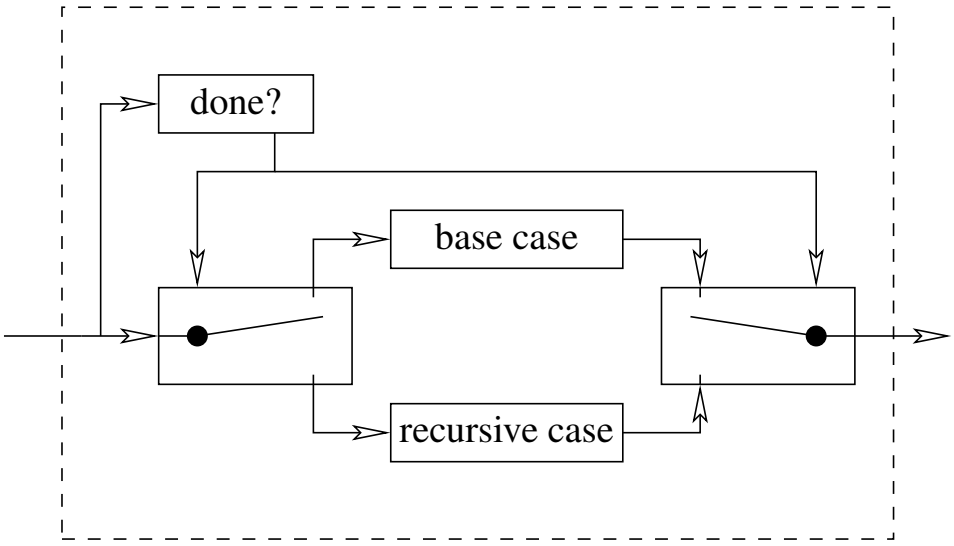


Figure 6-7: A template for simple recursive functions. If the `done?` test passes, the recursive call gets no input and remains unexpanded.

of the cells they are linking to.

A disadvantage of abstraction by delayed blueprints is that reclaiming the storage used to compute the return values of functions becomes an exercise in deducing what portions of a network have done everything they were going to do, and manipulating the network structure to remove them. This is harder with propagators than with expressions because even if a compound has produced something in some "output", one must wait until all the propagators internal to it have quiesced to know that it will not produce any more. Quiescence is scary because it is an emergent property of an entire (piece of) network, rather than something easily observable in any one place. Further, to actually completely remove an expanded compound, one needs to know that one will never wish to submit refinements to it (or if one does, that one is willing to call it again from scratch). Trying to get all this right is daunting, especially if it involves directly mutating network structure.

Recursion with delayed blueprints

The delayed blueprints strategy for abstraction suffices to build recursive networks. The technique might be called **input starvation**. Consider Figure 6-7. We install a switch to control the input to the recursive call. `Compound-propagator` is written so that if all the inputs to an abstraction are `nothing`, we do not recur into the abstraction, but merely assume that it does nothing. We will examine this choice more carefully in Section 6.2.4, but for now let us work out a concrete example of doing recursion like this, and verify that it works.

For our example recursive network, let us consider the problem of computing square roots by Heron's method [Heath, 1921]. The idea is that given a number x whose square root we want to find, we can improve any guess g of that square root

by the formula $g' = (g + x/g)/2$. We can start by building a little network that abstracts that formula; for purposes of illustration, we use `compound-propagator` to make it a delayed blueprint, even though, since it is not recursive, it could have been a non-delayed blueprint (which would have been a bit like making it a macro instead of a procedure in a normal programming language).

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

We can test this out, and notice that the abstraction works as desired;

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

(heron-step x guess better-guess)

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

and even produces a decent answer.

To complete our square root network, we will use three more abstractions. The interesting one repeatedly uses the `heron-step` subnetwork to make a guess better and better until an appropriate test is satisfied.

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

This is drawn in Figure 6-8; as you can see, it uses switches to prevent the recursive call if the `good-enuf?` test is satisfied; as such it is an instance of the simple pattern shown in Figure 6-7. This works because a `compound-propagator` only builds its

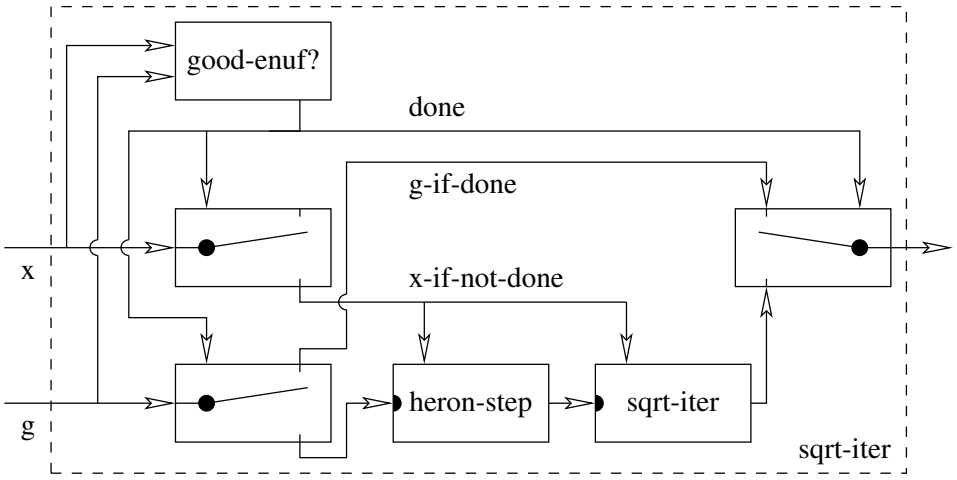


Figure 6-8: The recursive `sqrt-iter` network

body when presented with non-`nothing` values in the cells declared as its inputs, whereas `switches` always write `nothings` if their control inputs are false. This makes an interesting contrast with recursion in expression languages, where `if` notionally lives on the outputs of expressions, and controls whether those expressions are evaluated by suppressing interest in those outputs.

To actually try this out, we need two more abstractions: the `sqrt-network` that supplies an initial guess (in this case 1.0)

```
(define (sqrt-network x answer)
  (compound-propagator x
    (lambda ()
      (let ((one (make-cell)))
        ((constant 1.0) one)
        (sqrt-iter x one answer)))))
```

and the `good-enuf?` test

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

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

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

(sqrt-network x answer)

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

6.2.3 Virtual copies

A third way to do abstraction in a propagator network is in effect to use delayed blueprints, but to create "virtual" rather than "physical" copies of the network the blueprint describes.[3] This can be accomplished by introducing a data structure into cells that maps a token representing the identity of a virtual copy to the value (actually partial knowledge) in that cell in that virtual copy. Abstraction by virtual copies makes the shape of the propagator network reflect the shape of the source code of a program rather than the shape of its execution: the propagators actually constructed correspond to bits of code, whereas the shape of the execution is tracked by the collection of virtual copies that exist: for any particular token, the set of mappings of that token in all cells corresponds to one frame of a Scheme lexical environment. In order to build closures, I expect to need a notion of "parent token" for the tokens.

In order to actually implement virtual copies, it is also necessary to insert a translation object at each call site that will create new virtual copies as they are needed, attach the appropriate token to "procedure arguments" when transmitting them into the abstraction, and detach it when transmitting "return values" out of it. In particular, this translation object must maintain a mapping between the virtual copies of the "caller" and those of the "callee", so that each "call" can take inputs from and produce outputs to the correct virtual copy. Such a mapping is a generalization of the "previous frame" concept in a stack.

One disadvantage of abstraction by virtual copies is that it requires additional machinery not present in the base propagator system, namely the token-value mappings and the translation objects. This is not so bad, however, because the propagator infrastructure is flexible enough that this machinery can be added in user space: the token-value mappings are yet another kind of partial information, and the translators are yet another kind of propagator. No fundamental design changes are required.

One advantage of virtual copies is that removing functions that have finished might be easier. Of course, one must still detect this condition, with all the attendant problems of internal quiescence and external promises of non-refinement, but if one has, there is no need to explicitly manipulate the link structure of the propagator network because the intermediate virtual copy that is known to be finished can just

[3] In that sense, this strategy effects the separation of invocations in a "logical" space rather than the "physical" space of the network structure.

be unlinked from the caller-callee mapping.[45] This corresponds to removing a stack frame from the stack after it returns, or garbage-collecting an environment that was used for an evaluation that returned (and wasn't captured by a closure).

The virtual copy data structure is a special case of a truth maintenance system, with a particular pattern of premises. Specifically the premises are of the form "I am operating in virtual copy X." No two such premises are presumably ever believed at once, and they presumably have predictable patterns of times when they will never be believed again. Both of these assumptions may suggest more efficient implementations of virtual copies than with general-purpose TMSes, but in principle these two structures are very similar.

Since abstraction by virtual copies requires more machinery than the other abstraction techniques, some questions about virtual copies remain open:

First, the preceding discussion of virtual copies ignores the famous "funarg problem". In thinking of the token to value maps as just flat maps, one may be shooting oneself in the foot with respect to implementing closures. These mappings may need to be somewhat more elaborate environment structures. Also, what do "free variables" mean in this context?

Second, the `binary-amb` propagator interacts strangely with virtual copies. The problem is that each `binary-amb` has an identity that is visible in the true and false premises that it creates. As a consequence, each instance of a blueprint that calls for a `binary-amb` had better contain its own `binary-amb`, but since the virtual copy mechanism described so far only copies the contents of the cells, everything breaks. Especially since `binary-amb` has no input cells it can examine to determine that it is in a virtual copy (or, for that matter, which virtual copy it's in), some additional mechanism seems necessary to make the two interoperate. What should that mechanism be?

6.2.4 Opening abstractions

An abstraction is a named piece of network that is manipulated by reference to its name alone. In particular, recursion works because the name of the recursive call suffices until we need to invoke it, and it is not necessary to explicitly represent the fact that that recursive call may have a recursive call of its own, and that deeper one may have one deeper yet, and so forth. In order to execute an abstraction, however, it is necessary to dereference its name, and construct, in whatever way the abstraction mechanism specifies, a representation of that use of that abstraction. In previous sections we were concerned with how that might be done; here let us turn

[4]Presumably, if the token keying a virtual copy does not appear as the target of any caller mapping, the garbage collector can reclaim it. Presumably, the garbage collector can also reclaim all the entries in all the copy-value tables that are keyed by a token that has become inaccessible.

[5]There may be forms of partial information that inherently break returns by insisting on tracking the whole history of a computation. In this case, the "finished" frames never become irrelevant, perhaps because they can be revisited. That problem is separate from the ability to implement abstractions at all.

our attention to questions of when to do it.

Recall the `compound-propagator` procedure from Section 6.2.2:

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

Here the abstraction is opened (by calling the `to-build` procedure) as soon as any of its inputs is not `nothing`[6] (and the act of opening is memoized to avoid building the innards repeatedly).

This strategy is the most eager possible opening of abstractions. The only way to be more eager is to open all abstractions unconditionally, but then recursive networks would be impossible because there would be no way to keep an unneeded abstraction (like the recursive call one doesn't make) closed. This strategy is similar in a sense to normal order evaluation: in normal order an abstraction is opened as soon as its output is demanded, which demand is always the first thing to show up at the abstraction boundary. Here, the abstraction is opened as soon as anything arrives at the boundary.

One might wonder what applicative order may be like. Consider the entire argument list as a single entity: The eager strategy opens the abstraction as soon as anything is known about any component of it. Applicative order, however, corresponds to waiting until *everything* is known about the argument list before opening the abstraction. But in network land, that means waiting until *everything* is known about *all* the arguments.

That strategy is much too draconian for general partial information types. If one is using a truth maintenance system, for example, one may know that some answer is 5 according to Brian, but one may never receive final confirmation that Brian is to be believed unconditionally. Waiting for the rest of the network to quiesce, even if plausible, is also no guarantee that no more information about an input is forthcoming, because if the network has a cycle, some consequence of the abstraction itself may eventually refine one of its inputs.

There is one situation where an analogue of applicative order for networks is

[6]Inputless compound propagators are strange. As written, this will never open any of them. The obvious modification to this that always opens inputless compounds will open them even if their outputs are not needed; this can cause trouble should one build a recursive inputless compound propagator. The essence of the problem is that if one has no inputs, knowing nothing about them is the same as knowing everything about them. Regular programming languages avoid this problem, because computations are not attempted until they are needed; in effect, computations have an extra input which indicates whether the computation is desired or not, and consequently there are no truly inputless computations.

workable, and that is when the "partial" information structures are actually complete. If one has a five in a cell, and that five means "This cell has a five and that's final", that can effectively be treated as full information. In a region of network where all cells reliably carry such information, an applicative-order abstraction-opening policy is possible. This case is important because it is how networks mimic ordinary programming, and applicative order may prove useful in such regions for the same reasons that it proves useful in ordinary programming languages. On the other hand, in network regions with richer partial information structure, the question of normal versus applicative order stops being a dichotomy, and a very wide variety of abstraction opening policies become possible.

The decision of how much knowledge to insist on before opening an abstraction can be left to user space. If the network's general policy is to open abstractions as soon as anything is known about their argument lists, taken as whole entities, one can always put a filtering propagator on the front of any particular abstraction that will wait until its input contains enough knowledge to satisfy the filter that the abstraction is worth opening, and only then pass that knowledge along; so that even though things may accumulate before it is ready, the abstraction behind the filter will see only `nothing` until the filter is satisfied. (In general, such filters can be placed on a per-call-site basis, but of course one can abstract the combination of a particular such filter on top of a particular abstraction, and thus mimic the discipline of putting the filters on the abstractions themselves.) This corresponds quite directly to strictness annotations on procedures (or call sites!) in an otherwise lazy language, except that rather than a binary choice of "strict" or "not strict" in each argument, one actually has access to a continuum of abilities to insist on various amounts of knowledge about various arguments (and even their totality taken as one). One might therefore imagine an extension of the idea of strictness analysis to autogenerate such filters for abstractions that may benefit from them. It is also intriguing that such filters seem very similar to the deciders suggested in Section 7.3 for general side effects. Perhaps choosing to dedicate the resources to the representation of an abstraction invocation is sufficiently observable and irreversible to constitute a side effect of sorts?

6.3 What Partial Information to Keep about Compound Data?

Compound data structures like Lisp's `cons` cells are key to the utility of a programming system. It turns out compound data differs enough from primitive data (like numbers[7]) that its treatment in a propagation context is worth examining as such.

Consider a pair, as made by `cons`. Suppose we are using a TMS, like in Section 4.2, for our partial information (over primitives), and are trying to cons up two

[7]It is ironic that numbers are the standard example of a primitive data type, as one might argue that numbers are perhaps the richest and most complex domain that programming languages must routinely deal with. Perhaps calling them primitive is just a way to hide that richness and complexity behind a wall called "primitive arithmetic" and avoid having to think about it.

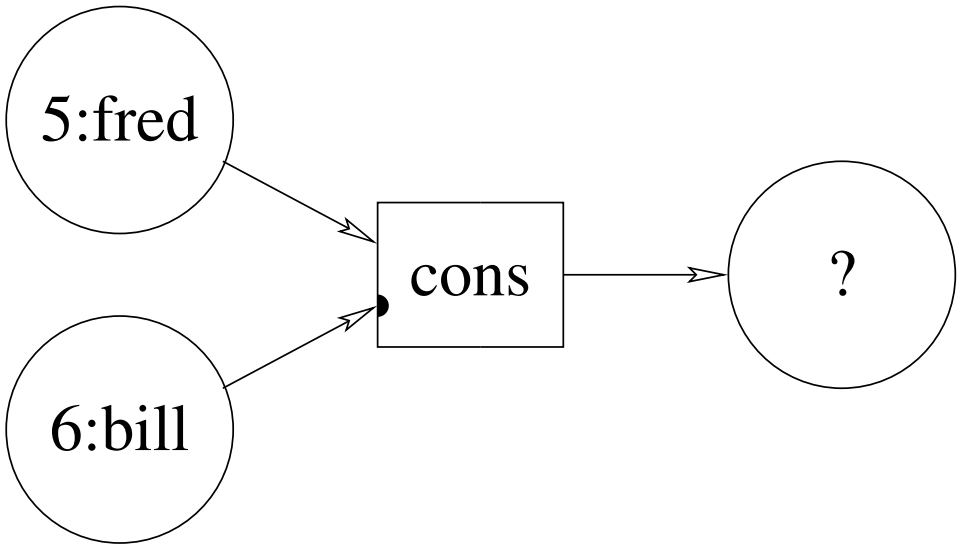


Figure 6-9: Consing two TMSes. If the first argument to `cons` depends on `fred`, and the second argument depends on `bill`, what structure should the `cons` emit into its output?

objects: `(cons x y)`. So `x` is a cell, and suppose it contains a TMS that says "`5` if you believe Fred" (notated `5:fred`); `y` is a cell, and suppose it contains a TMS that says "`6` if you believe Bill" (notated `6:bill`); and `cons` is a propagator that needs to produce something to put into the cell attached to the output of that expression. See Figure 6-9. What should it put there?

The naive thing to do is to treat `cons` the same way we treat `+` (if you'll recall the treatment of `+` from Section 3.4),

```
(define generic-cons (make-generic-operator 2 'cons cons))
(define conser
  (function->propagator-constructor (nary-unpacking generic-cons)))
```

that is to unwrap the input TMSes, run the primitive `cons`, and wrap the result in an appropriate output TMS. This produces a TMS that says "`(5 . 6)` if you believe Fred and Bill:"

```
(define x (make-cell))
(add-content x (make-tms (supported 5 '(fred))))
(define y (make-cell))
(add-content y (make-tms (supported 6 '(bill))))
(define out (make-cell))
(conser x y out)
(content out)
#(tms (#(supported (5 . 6) (bill fred))))
```

While this is certainly consistent, it is dissatisfying, because when we then take the `car` of the resulting object in the same style,

```
(define generic-car (make-generic-operator 1 'car car))
(define carer
  (function->propagator-constructor (nary-unpacking generic-car)))
```

we get "5 if you believe Fred and Bill:"

```
(define x-again (make-cell))
(carer out x-again)
(content x-again)
#(tms (#(supported 5 (bill fred))))
```

We have violated the identity that `(car (cons x y))` should be `x` regardless of `y`. The violation is conservative, in that we have not forgotten any dependencies we should have remembered, but we have manufactured dependencies that weren't originally present. In effect, the consing has forgotten which of its dependencies came from the first part and which from the second, and so gave us all of them.

This is a serious defect in the dependency tracking system. Data aggregates are so prevalent that being unable to separate the dependencies of the individual members of the aggregate would limit the dependency tracker's usefulness for anything big enough that one can no longer assign a separate variable to each individual datum.

One might argue that this is a defect only in the dependency tracking system, and therefore does not illuminate the design of the propagation infrastructure. I would disagree with such a one, on the grounds that the dependency tracking system is a very compelling example of a partial information type, and illustrates a problem that other partial information types may also encounter. I would suggest that it is worth considering how to design the basic `cons`, `car`, and `cdr` propagators so as to make it convenient to process partial information types as elaborate as dependency tracking, especially since there turn out to be different options, some of which do require support from the propagation infrastructure at a very low level. So let us consider the general approaches to this problem, using dependency tracking as our motivating example.

The essence of both proposed solutions is to keep track of the information we have about each element of the pair separately; the difference being how to manage the separation. For dependencies, that amounts to keeping separate truth maintenance systems for each element of the pair. Before we go into that, though, it is worth noting that our information about the pair itself may also be partial, entirely separately from any partialness to our information about its parts. The easiest way to imagine that happening is to think of a pair we received through an `if` whose predicate we didn't fully know. Suppose for example that we have a predicate `foo?` that in our case will produce a TMS that says "`#t` if you believe George and `#f` if you believe Carol". Then

```
(if (foo?)
    (cons 1 2)
    (cons 3 4))
```

should produce a TMS that says "`(1 . 2)` if you believe George and `(3 . 4)` if you believe Carol," which is a partial information structure that describes which pair we have in our hands. The "which pair" question is entirely separate from information about the contents of those pairs. In this case, we had complete information about the contents of the pairs, but it could have been partial too; and then we would need three partial information data structures, one for each of the `car` and the `cdr`, and one more for the pair itself.

6.3.1 Recursive partial information

This dissertation offers two strategies for separating the partial information about the parts of a data structure from each other and from the information about the data structure itself. The first is to ask the data structure to contain partial information structures instead of direct values. The `cons` propagator for that turns out to be really simple:

```
(define conser (function->propagator-constructor cons))
```

Then if we are always certain about the pairs themselves, extraction need, in principle, only pull out the partial information about the appropriate piece:

```
(define carer (function->propagator-constructor car))
```

If, however, the information about the pair itself (the "which pair" question, from above) happened to be partial, then the extractor needs to pull the pair out of that structure and let the result inherit the uncertainties about the pair itself appropriately. It turns out that the generic operation machinery built in Appendix A.5.1 handles that case correctly:[8]

```
(define carer (function->propagator-constructor (nary-unpacking car)))
```

A simple mechanism for merging pairs that show up in cells suggests itself: just recursively merge the `car`s and the `cdr`s and `recons` them.

---

[8]This is not an accident; in fact, getting `switch` (Section 6.1) and `car` right was the main motivation for writing that code that way. Appendix A.5 describes how it was done and why it is interesting, but suffice it to say here that a single uniform treatment of partial information types does what we want for `+`, `switch` and `car`, even though we did something different for `cons`.

```
(define (pair-merge pair1 pair2)
  (let ((car-answer (merge (car pair1) (car pair2)))
        (cdr-answer (merge (cdr pair1) (cdr pair2))))
    (cond ((and (eq? (car pair1) car-answer)
                (eq? (cdr pair1) cdr-answer))
           pair1)
          ((and (eq? (car pair2) car-answer)
                (eq? (cdr pair2) cdr-answer))
           pair2)
          (else
           (cons car-answer cdr-answer)))))

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

So let's put it all together and see Spot run:

```
(define x (make-cell))
(define y (make-cell))
(define pair (make-cell))
(conser x y pair)

(content pair)
( #(*the-nothing*) . #(*the-nothing*) )
```

Note how `conser` aggressively manufactures pairs with empty contents—the fact that there is supposed to be a pair there may be interesting to some client in itself, regardless of what should be in it.

This machine also does the right thing about uncertainty about the pairs themselves:

```
(define control (make-cell))
(define switched-pair (make-cell))
(switch control pair switched-pair)

(add-content control (make-tms (supported #t '(joe))))
(content switched-pair)
#(tms (#(supported ( #(*the-nothing*) . #(*the-nothing*) ) (joe))))
```

Since the `switch` is only on if we believe `joe`, whether there is a pair in the `switch`'s output is also dependent on `joe`. So that's an instance of a pair whose identity is not known for sure, and which consequently resides inside a TMS.

```
(define x-again (make-cell))
(carer switched-pair x-again)

(content x-again)
#(*the-nothing*)
```

But if we try to take the `car` of this pair, we get `nothing`, because there's nothing in the `car` position of the pair yet. This situation looks like Figure 6-10. Now let's see what happens when we add something to the `x` that's getting `cons`ed up:

```
(add-content x (make-tms (supported 4 '(harry))))
```

Our addition gets put into the pair in the `pair` cell,

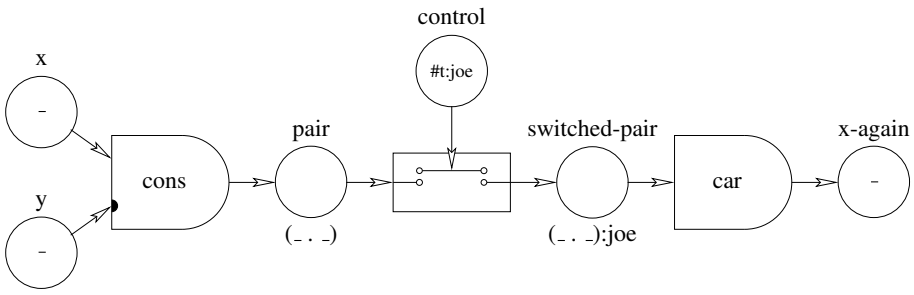


Figure 6-10: A propagator network with a `cons` and a `switch`

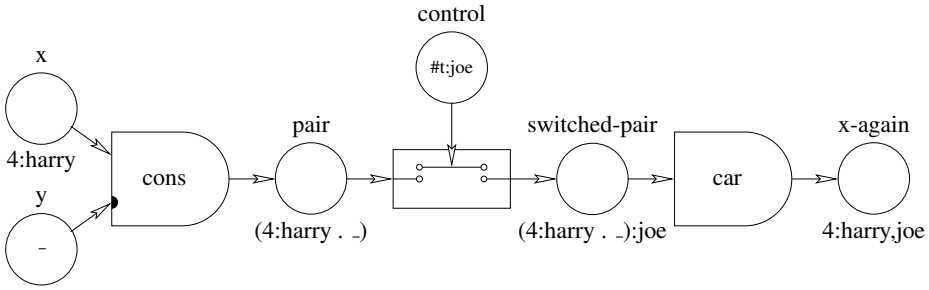


Figure 6-11: A propagator network with a `cons`, a `switch`, and some data

```
(content pair)
( #(tms (#(supported 4 (harry)))) . #(*the-nothing*) )
```

whence it travels to the conditional pair in the `switched-pair` cell,

```
(content switched-pair)
#(tms (#(supported ( #(tms (#(supported 4 (harry)))) . #(*the-nothing*) )
                   (joe))))
```

so that our `carer` propagator can extract it, appropriately tracking the fact that it now depends on `joe` as well as `harry`.

```
(content x-again)
#(tms (#(supported 4 (harry joe))))
```

This situation looks like Figure 6-11. Note that we managed to get a useful result out of this computation even though we never supplied anything in the `y` cell, that was supposed to go into the `cdr` position of our pair. This behavior looks like a lazy language with a non-strict primitive `cons`. As remarked in Section 6.2.4, this is in keeping with our general philosophy of aggressively running propagators whenever possible.[9]

[9]On the subject of Section 6.2.4, it argues that one needs to suppose *some* situation where one can rely on an abstraction not doing anything interesting, in order to be able to avoid an infinite regress of expansions of recursive abstractions. Section 6.2.4 further argues that the most sensible such rule

This strategy recovers the nice feature that the inverse of a `cons` is an appropriate pair of a `car` and `cdr`, and the inverse of a lone `car` or `cdr` is a `cons` with an empty cell for the missing argument.

While the code presented sufficed for the example in the text, the recursive partial information strategy is not yet fully worked out. If one runs the given code as written, one will discover an unfortunate bug, which is that a TMS inside the `car` of a pair that is itself inside of a TMS is not properly aware of the dependencies it inherits from its container. In particular, if the interior TMS detects a contradiction, as when merging `(4:fred . _):george` with `(3:bill . _):joe`, it will signal the contradiction without proper accounting for its context (that is, it will signal that Fred and Bill contradict each other, without noticing that all four of Fred, George, Bill, and Joe are actually needed to cause a contradiction).

Another, perhaps related, problem is that the code as written is not precise enough about tracking the dependencies of the fact that some cell in fact contains a pair. For example, when merging `(4:fred . _):george` with `(_ . 3:bill)`, one should observe that the result is a pair unconditionally; that its `cdr` is 3 only depends on Bill; and that its `car` is 4 depends on Fred and George. In other words, the right merge is `(4:fred,george . 3:bill)`. The code presented produces `(4:fred . 3:bill):george` instead. This is at least conservative, but still disappointing. On the other hand, doing better may involve large amounts of research on the structure of the `cons`, `car`, `cdr` algebra, and teaching the dependency tracker to be aware of it.[10]

6.3.2 Carrying cells

The second strategy is to let pairs contain network cells directly. Then the `cons` propagator operates not on the contents of its input cells, but on the input cells themselves. It can package them up in the pair data structure and let the result flow along the network; then when it comes time to extract, the `car` and `cdr` propagators can extract the appropriate cell from the pair, and then further extract its content. The simplest incarnation of this in code is indeed simple:

---

is to assume that an abstraction will do nothing useful if all its inputs are `nothing`. However, we have here a propagator, namely `cons`, that *does* do something potentially useful, even though all of its inputs are `nothing` (and it actually has inputs, unlike a `constant`). That means that wrapping `cons` into an abstraction such as suggested in Section 6.2.4 would alter its behavior. Oops. Seeking a good solution to this problem is an avenue open to future research.

[10]The difficulty of doing algebra is perhaps the fundamental reason why `+` is different, in our minds, from `cons`. Indeed, why are we content to let arithmetic operations on numbers just flatly merge their dependencies, but we single out compound data like `cons` and force just them to carefully track the dependencies of what came in? Why do we demand that `(car (cons x y))` not carry the dependencies of `y`, but allow `(+ x (* -1 x))` to carry the dependencies of `x`? Maybe the reason is just that algebra over the arithmetic of numbers is so much harder than algebra over simple containers like `cons` that we accept a gross approximation in one case and balk at that same approximation in the other.

```
(define (conser a-cell d-cell output)
  (propagator ()                          ; That's right, no inputs.
    (lambda ()
      (add-content output
        (cons a-cell d-cell)))))

(define (carer cons-cell output)
  (propagator (list cons-cell)
    (lambda ()
      (add-content output
        (content (car (content cons-cell)))))))
```

This strategy has the advantage that if some partial information about some object contained inside a pair gets refined, the update need not propagate through an entire long chain of cells containing pairs, but can go straight through to the cell receiving the appropriate component. On the other hand, this has the weird effect of having cells contain data structures that contain pointers to other cells, and that causes some problems. Philosophically, there is the effect that the structure of the network is now even more dynamic, and that's scary. More practically, the `carer` given above actually needs to register either itself or some other propagator to receive updates from the transported cell, perhaps thus:

```
(define (carer cons-cell output)
  (propagator (list cons-cell)
    (lambda ()
      (identity (car (content cons-cell)) output))))

(define (identity input output)
  (propagator (list input)
    (lambda ()
      (add-content output (content input)))))
```

This addition of the `identity` propagators should be memoized, so that we do not end up with a large number of the same propagators floating around. Further, if the incoming pair is only known partially, the extraction of the transported cell will need to take care to actually collect a pair to operate on; and the `identity` propagator will need to be somehow conditional on the partial information about said incoming pair, and should of course merge that partial information with the result it produces. For example, in the case of TMSes,

```
(define (carer cons-cell output)
  (propagator (list cons-cell)
    (lambda ()
      (let* ((best-pair (tms-query (content cons-cell)))
             (transported-cell (car (v&s-value best-pair))))
        ((conditional-identity (v&s-support best-pair))
         transported-cell output)))))
```

```
(define ((conditional-identity support) input output)
  (propagator (list input)
    (lambda ()
      (if (all-premises-in? support)
          (add-content output
            (attach-support (tms-query (content input)) support))))))

(define (attach-support v&s more-support)
  (supported
   (v&s-value v&s)
   (lset-union eq? (v&s-support v&s) more-support)))
```

Note that the `conser` propagator need not be changed, since it deals with cells and not their contents; nonetheless, doing this uniformly across partial information types remains an outstanding challenge. For example, the astute reader may have noticed that the code above lacks several necessary checks for the possibility of things being `nothing`.

Another challenge also remains outstanding with this "carrying cells" strategy. It is not at all clear how to merge two pairs, because doing that seems to require "merging" the cells they carry. Perhaps it would be appropriate to keep all those cells, and interlink them with (conditional) identity propagators? I also expect that all the problems attendant on merging pairs represented with the recursive partial information strategy of Section 6.3.1 will resurface in one or another form here.

6.3.3 Other considerations

Compound data structures also present an entirely different problem: moving them around. Consider a `cons` propagator tasked with making a pair out of the contents of its two input cells. What if those cells themselves contain large compound data structures? What if the output of the `cons` is an input to another `cons`? Copying big compound data structures from cell to cell could prove very expensive!

On modern hardware architectures, this breaks down into two cases: Either the cells in question live in the same piece of memory (e.g., multiple cells in a network simulated by one processor, or on a shared-memory multiprocessor), or they live in different pieces of memory (e.g., cells in a distributed network that are homed on different computers).

The first case is of course easier to deal with than the second: Functional programming style solves the problem completely. That is to say, forbid in-place mutation of data structures (no `set-car!`)[11]. Then you never have to copy data structures, because a reference will always be as good as a copy if all anyone can do is read it. You also don't need to worry about detecting mutations deep in the guts of a data structure that a cell is holding to notify the propagators reading that cell that they may need to rerun.

[11]This restriction is far less severe than it may seem. There is already a great deal of research, e.g., [Driscoll et al., 1989], about **persistent data structures**, that is data structures that are updated by producing new versions with the desired changes rather than by mutation in place.

The second case, large data structures on distributed networks, is of course harder. I also have not dealt with it, because my prototype does live on just one computer, so all cells do share a memory, and copying pointers to immutable structures works perfectly. Nonetheless, I can foresee two basic approaches: one is to hope (or arrange) that large data structures will not have to be copied across machines very often, and the other is to play some game with cross-machine pointers (perhaps copying (read: transmitting across a network) structure lazily as such pointers are dereferenced). Both would be greatly simplified by again disallowing mutation; the latter approach would also complicate garbage collection.

6.4 Scheduling can be Smarter

Most of the discussions in this dissertation have assumed that computation is essentially free—per Section 2.1, every propagator is a notionally independent machine that tries its computation all the time, presumably without costing anyone anything; per Section 6.2.2, more of these propagators can be created at will; and per the rest of the dissertation they operate on arbitrarily complex partial information structures, and ask the cells to merge arbitrarily complex additions of information. This is all good and well for the study of the general properties of such propagation systems, and for the consideration of their expressive power in the abstract, which of course is the main concern in the present work, but the assumption of limitless computation will surely break down in practice. Let us therefore take a moment to think on what the propagation infrastructure might be able to do to use resources reasonably.

Indeed, the prototype presented in this dissertation is subject to resource limitations of exactly the kind that it gleefully ignores. Since all my propagators are simulated on (a single core of) a single computer, they compete with each other for CPU time—when one propagator in one part of the network is running, others are not, even though perhaps they could. What, then, can be done to make the best use of that limited CPU time? How well can useless or redundant work be avoided?

Efficient resource allocation to propagators is an important issue open to future work. Just one observation draws attention to itself: Our infrastructure is aware of the partialness of the information it deals with, and it is clear that some partial information structures contain strictly more information than others. Perhaps there may be a way to measure the amount of information a particular partial structure contains. If in addition it proves possible to predict the dependence between the new information that has become available in a propagator's inputs since its last run and the information it will produce for its outputs if it is run again, it may then be good to schedule propagators by the general philosophy "run the most informative first."

Let me illustrate the idea with an example. Suppose we wanted to solve the single-source shortest-paths problem [Cormen et al., 2001] for some graph, by using a propagator network. We could make a network with one cell for each node,

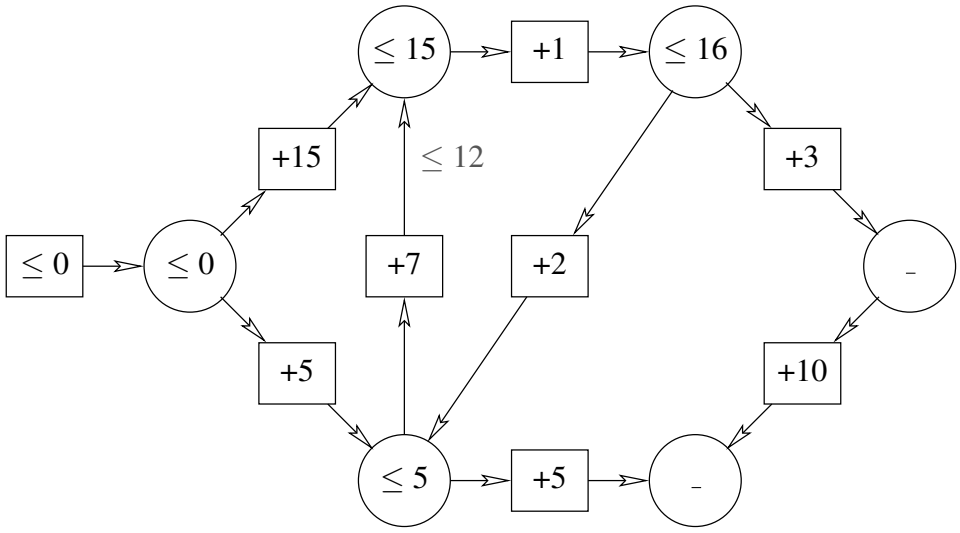


Figure 6-12: A network computing the shortest paths in a graph. It will converge to the right answer eventually, but depending on the order in which it invokes the propagators, it may invoke them many more times than needed.

one propagator for each directed edge, and a partial information structure that stored an upper bound on how far each node was from the source. Each propagator's job would be to inform the head of its edge that it was no further from the source than the distance of the tail of the edge plus the length of the edge. Such a network in operation might look like Figure 6-12.

If we ran this shortest-paths network with the scheduler in Appendix A.2, it would converge to the right answer eventually, but there's no telling how many times it would invoke each edge-propagator, deducing all manner of small improvements to the possible paths. We could, however, say that every upper bound is as informative as it is small, and that every propagator's output will be informative in proportion to the informativeness of its input. Then the propagators whose tail cells have the lowest upper bounds for paths from the source are also the most informative propagators to run, and the priority queue of propagators by informativeness will behave exactly like the standard priority queue of nodes to expand in the usual shortest paths algorithm. We will have replicated the performance of a hand-coded algorithm, up to constant factors, in a framework expressive enough to deduce the control loop we wanted for us.

In addition to the elegance of the previous example, I deem this direction of research encouraging because the scheduling policy outlined in Section 3.1 and used in this dissertation is actually a special case of the rule that uninformative propagators should be run last. Specifically it can be interpreted as running informative propagators first using the following simpleminded information metric: any propagator whose inputs have changed since its last run is predicted to have informativeness 1, any whose have not 0, and the rest of the world (i.e., returning

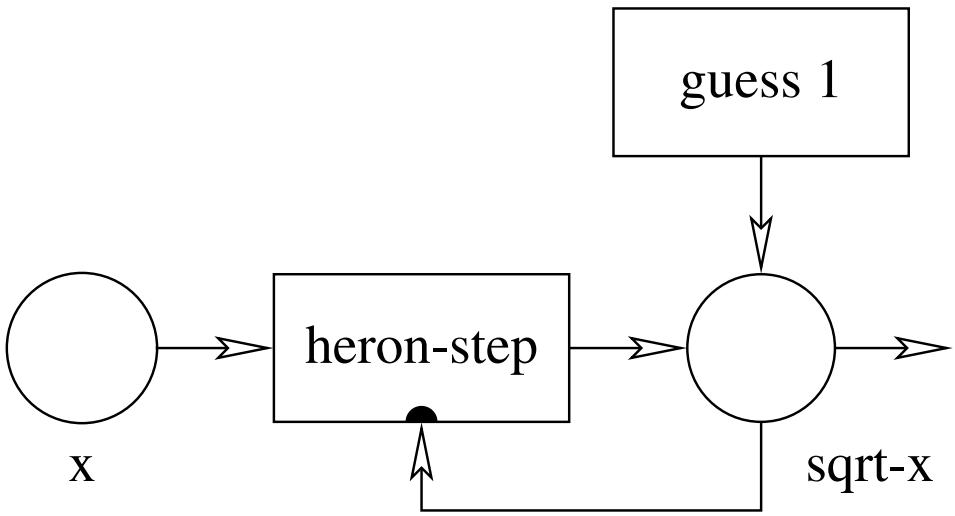


Figure 6-13: Square roots by feedback. In principle, this could refine its guess forever, but it is possible to predict that further refinements are less and less useful.

from the simulation loop) 1/2. Thus all quiescent propagators implicitly sort after the "end the simulation" pseudopropagator, and don't need to be kept on the queue at all; and all non-quiescent propagators sort before it in some haphazard order, represented by the scheduler's single (non-priority-)queue.

On the other hand, guessing the actual informativeness of running a propagator based on the amount of information accumulated in its inputs is of course uncomputable in general. The deciders suggested in Section 7.3 promise to be particularly practically troubling, because by their very nature, their translation from their inputs to their outputs can either suppress or greatly amplify changes in the inputs' information content. Simple propagators acting on simple information spaces can be predicted, however; there is much research in the constraint satisfaction field on that particular topic, as well as on the topic of scheduling propagators in general [Schulte and Stuckey, 2008, Tack, 2009].

As another example of the possible uses for scheduling propagators by informativeness, consider computing estimated square roots with a feedback loop[12] as in Figure 6-13 (as opposed to the throttled recursion of Figure 6-8 on page 118). If the `sqrt-x` cell holds an appropriate partial information structure, the feedback loop can continue refining the guess indefinitely (or until it hits the limits of precision, if the arithmetic is inexact). On the other hand, it is easy enough to predict that after a point successive refinements will be of limited use, as they are only making a small error smaller. So at some point, whatever propagators are reading that square root answer will gain higher priority than those continuing to refine the guess, and will run. Then whatever propagators are reading *those* answers will gain higher

[12]Analog circuits are often arranged to compute things with feedback loops. Perhaps this idea can recover some of the benefits of analog computation, but without paying the cost of noise intolerance.

priority than the propagators continuing to refine the guess, and *they* will run—the computation will proceed, in spite of containing a possibly infinite loop;[13] and the question of whether an answer is good enough can be left for the system itself to determine by its informativeness measures rather than having to be hardcoded into the square root routine.

This idea opens avenues of thought about weighing the possible *benefit* of new information propagators might produce. One can envision mechanisms for the user to indicate their interest in some or another item of information, and transitively the items from which it is computed. One can perhaps also envision estimates of the *cost* of running some propagators, and weighing that cost against the estimated benefit of the information they may produce. This is highly reminiscent of various work in the constraint satisfaction field (e.g., [Tack, 2009]); perhaps some of those techniques can be adapted to general-purpose computation.

6.5 Propagation Needs Better Garbage Collection

Garbage collection is an important technical issue that the present work has largely swept under the rug. The prototype propagation system presented in this dissertation gives no help to the underlying Scheme garbage collector. In fact, since propagators hold on to their cells and cells hold on to their propagators, both strongly, entire networks, including all the information they have deduced, remain in memory as long as the host Scheme holds on to any portion of them.

In the abstract, retaining entire networks in memory as units is not all that wrong—after all, who knows what portion of the network may receive a new input or deduce something interesting, and where in the network that deduction may propagate? Especially if the network has cycles? On the other hand, if we are to use networks to mirror the executions of long-running programs, we must, as a practical matter, be able to find sections of network to reclaim.

The right way to do that is an open problem. It relates to the mechanisms of abstraction the network uses, because those mechanisms will have much to say about the shape of the network, and perhaps even about which portions of the network are in active use. It also relates to the questions of information measurement brought up in Section 6.4, in that reclaiming the memory associated with a piece of network structure is perhaps tantamount to saying that none of those propagators will be run soon enough to be worth remembering (especially if the abstraction blueprint from which they were built is still around somewhere). Likewise, questions of decision-making discussed in Section 7.3 are perhaps relevant, because choosing to reclaim some memory is perhaps an irrevocable and observable (side-)effect on the resource use of the system.

[13] This therefore amounts to a notion of fairness of resource allocation. If we presume, or engineer, that propagators running in a limited area of the network will in time exhaust their relative informativeness, then they are automatically prevented from hogging the system's resources indefinitely.

In addition to questions of garbage collecting the network structure itself, a propagation substrate brings up questions of garbage collecting appropriate pieces of the partial information structures being propagated. For example, consider a truth maintenance system in a cell that knows "This cell is 5 if you believe Fred and Bill," and subsequently discovers "This cell is 5 if you believe Fred." The second of these facts logically subsumes the first, so the first can have no further effect on the computation. Therefore, the first can be forgotten. In a sense, this is a form of garbage collection, which depends on the invariants of the truth maintenance system.

Some such garbage collection can be done locally during the updates of individual structures. For example, the TMSes in the prototype in this dissertation do release the storage associated with logically subsumed facts within a single TMS inside a single cell. Other such garbage collection may perhaps involve more global processes. For example, a `binary-amb` propagator that is responsible for choosing between A and $\neg A$ should of course be informed if A and B are discovered to form a nogood, so that it can refrain from choosing A if B is currently believed. However, there is no a priori reason for it to be informed if B alone is discovered to form a nogood—after all, what does B have to do with A?—except that in this particular case the `binary-amb` *should* be briefly informed of this, because B alone being no good subsumes the A and B nogood, and so means that the A and B nogood should be garbage collected, for which purpose that `binary-amb` needs to release it (and then it can also release the B nogood, because it now presumably has no further impact on the A chooser's activities). The right way to arrange such semantic garbage collection remains an interesting question.

6.6 Side Effects Always Cause Trouble

No programming system is complete without the ability to emit side effects. To be sure, many a programming system is useful without that ability, but we have come to expect it from those we accept as "full programming languages." How, then, should a propagation system do side effects?

The first stage of the answer is obvious: produce primitive propagators that execute actions on the world. For example, we could write

```
(define displayer (function->propagator-constructor display))
```

to make a propagator that, every time it runs, displays the contents of its input cell and writes a token (the return value of `display`) into its output cell.

Unfortunately, what to do next is not as obvious. The `displayer` described above, used naively, could run at chaotic times, and could expose the details of partial information structures—we need something more. For instance, if we wanted to compute and display an answer to some question, we might find, at some point during our computation, that the answer is twenty-three if we believe Bob and thirty-five if we believe Fred. If our `displayer` happened to be attached to that

cell directly, it might then display a representation of that truth maintenance system; and then when we later discovered that both Bob and Fred were actually wrong and the incontrovertible answer was forty-two, the displayer might run again, and display the forty-two. Perhaps this behavior is not what we want.

The right solution to this problem is an exciting topic for future research. There is a glimmer of hope in the observation that timings are not always unpredictable in propagator networks: If some region of the network is acyclic, and admits only `nothing` or complete, non-refinable values as partial information states, then that portion of the network will behave basically like a normal program, and can therefore be arranged to execute any side-effecting propagators it contains at predictable points. The technical question of communication between such regions and other regions where more elaborate partial information structures reign remains open; I will only mention the matter again to reflect on the deep meaning of that question in Section 7.3.

6.7 Input is not Trivial Either

Symmetrically to emitting side effects, no programming system is complete without being able to observe events in the world and incorporate them in one or another way into a program's behavior. How, then, should a propagation system accept input?

The prototype presented in this dissertation cops out. Input is given to a network by the user calling `add-content` on some cell from the Scheme read-eval-print loop while the network is not running. Doing this schedules the neighbors of that cell to be run when next the scheduler is started, and that allows them to incorporate that input into their computation.

One particular feature of this input scheme is that, in a uniprocess Scheme system, the user cannot provide input while the rest of the network is running (short of interrupting the run). In effect, the "outside world," in its capacity as a source of input, looks to the network like just one big propagator, which it always runs only when everything else has quiesced.

Clearly, treating the rest of the world as a propagator that a particular network can choose to "run" when it pleases is untenable as such, but in a concurrent system that might not be unreasonable. If we say that the scheduler is always running, and will accept input as it comes, we may have a reasonable high-level story for input (though the meaning of communicating the network's quiescence to the user becomes an interesting question).

How to really do a good job of input to propagator networks remains an open question. I suspect that functional reactive systems, discussed in Section 5.5, will prove useful, both as a source of inspiration for the architecture, and as a source of concrete suggestions for particular abstractions (like a user's keystrokes, or the current time).

6.8 What do we Need for Self-Reliance?

The prototype presented in this dissertation is very much embedded in its host Scheme. What would it take to build a complete, self-hosting system out of it?

As a technical question, this question is the following: what is a sufficient set of primitives for the propagation mechanism that can be implemented in a host system (like Scheme) so that all further extensions of the propagation system can be made in the system itself?

That is a good question. The only answer I have at the moment is an enumeration of the places where the host Scheme is used for extending this prototype.

First, the language for making compound blueprints (Section 6.2.1) is Scheme. This is not entirely unreasonable: some textual language is necessary for constructing wiring diagrams, so why not let it be Scheme? Or, for that matter, any other language we might care to invent. This seems like a matter of choosing surface syntax, or of compiling Scheme descriptions of wiring diagrams to actual propagators that will execute those diagrams. It will of course be appropriate to add a suitable programmer interface for manipulating wiring diagrams as data, and the ability to attach them to the network afterward—the moral equivalent of the traditional Lisp `eval`.

Second, more worrisomely, delayed blueprints (see Section 6.2.2) are currently also written in Scheme. This is unfortunate—the host language should not intrude unduly upon so fundamental an operation of the guest language as creating abstractions. On the other hand, there is only one object that introduces the transition from networks to Scheme for this purpose; this problem may therefore be solvable with a dedicated piece of syntax.

Third, arguably even worse, new partial information types are also defined in Scheme. This breaks down into three pieces: defining how a new type of partial information merges (methods on the generic function `merge`), defining how existing propagators pass that information along (e.g., methods on the generic functions `generic-unpack` and `generic-flatten` from Appendix A.5), and defining new propagators that do interesting things with that type of partial information (the moral equivalent of `binary-amb`). The only answer I have to this is that since propagation can do anything that Scheme can do, it must be possible to write the various methods in network diagrams. I have not, however, explored this question in detail, because the prototype is slow enough as it is that trying to run a self-hosting version of it threatened to be an exercise in extended waiting. I expect that one interesting question will arise: if merging the existing content of cell A with some new content is to be done as a propagation (sub-)process, then it would probably be wise to run that (sub-)process to completion before permitting the network directly operating on cell A to continue. This suggests either some sort of recursive scheduling loop, or some sort of scheduler queue structure. Both of these are rather fearsomely global-looking.

Chapter 7

Philosophical Insights

A SHIFT such as from evaluation to propagation is transformative. You have followed me, gentle reader, through 137 pages of discussions, examples, implementations, technicalities, consequences and open problems attendant upon that transformation; sit back now and reflect with me, amid figurative pipe smoke, upon the deepest problems of computation, and the new way they can be seen after one's eyes have been transformed.

7.1 On Concurrency

The "concurrency problem" is a bogeyman of the field of computer science that has reached the point of being used to frighten children. The problem is usually stated equivalently to "How do we make computer languages that effectively describe concurrent systems?", where "effectively" is taken to mean "without tripping over our own coattails". This problem statement contains a hidden assumption. Indeed, the concurrency itself is not difficult in the least—the problem comes from trying to maintain the illusion that the events of a concurrent system occur in a particular chronological order.

We are used to thinking, both in our everyday lives and in our computer programs, of things as happening in a sequence. Some A occurs before B, which occurs before C, etc. Time of occurrence imposes a total order on events. When two events happen, each of which is not strictly before the other, we are used to saying that they occurred "at the same time", and composing them into a compound simultaneous event consisting of both of them. A concurrent system, by definition, is one where time becomes a partial order: Perhaps A occurs before B and before C, and perhaps B and C both occur before D, but perhaps neither of B or C occurs definitely before the other, and perhaps there is no consistent way to squish them into a compound simultaneous event. This seems weird and difficult to think about, especially when programming computers.

In point of fact, concurrency is the natural state of affairs, and synchronicity is what's difficult. The physical world in which we live is perfectly concurrent: every little patch of universe evolves on its own, according to local rules, and physical

effects travel from one patch to another at a finite speed (which we even know: 299,792,458 meters per second). As a matter of physics, time in the universe is partially ordered: A occurs before B if and only if A has any chance of influencing B, to wit if and only if light emitted at A can reach the location of B before B occurs. Over sufficiently large spaces or sufficiently small times this entails pairs of events neither of which occurred before the other, because light hasn't the time to reach in either direction. Moreover, there is no good way to call these events simultaneous, because C may occur near B so that B is clearly earlier than C, but so that still neither A nor C is earlier than the other; and also because an observer traveling towards the location of A may see A before seeing B, whereas one traveling towards B may see B before seeing A.

Our experience of time appears as a linear stream of events only because our memory imposes an order on them, which is the order in which events are remembered to have occurred. A human society, however, is in principle a concurrent system, because pairs of events can occur which different people will learn about in different orders, so the order in which people learned of something cannot unambiguously determine which event happened first. For example, a father may observe the birth (event A) of what he thinks is his first child. But he may not know that a one-night stand he had nine months ago resulted in a pregnancy. That other baby's mother, participating in that other baby's birth (event B) may know nothing of event A, and may presume that her child is the man's first. When each later learns about the other, it may perhaps matter which baby is the man's first-born, but the participants' native ordering will not determine an ordering for the events, because the wife will have learned of A before B, but the mistress will have learned of B before A.

The illusion of synchronous time is sustainable for our society, however, because the patch of universe we occupy synchronizes far faster than the temporal resolution of "event" our society expects. We can in principle use clocks and recorded times to disambiguate which of two events happened first after the fact, if we so choose. It may perhaps come to light that baby B was born at 3:14pm September 15th in Paris, while baby A at 1:41pm September 15th in New York, putting B first. This is a big enough difference in time and a small enough difference in space that the necessary relativistic distortions effected by movement upon clocks and those who read them is insignificant, and so our chronology infrastructure lets us arbitrate the order of these two events. Indeed, we are so used to the principle that, with good enough clocks and records, we could arbitrate the order of any two events, that we think of time as linear, and of all events as ordered, in principle, by time of occurrence.

As with the universe, so with electrical circuits (whether etched onto a piece of silicon or otherwise). A large electrical circuit, observed at a fine timescale, is fundamentally concurrent: electromagnetic waves, whose behavior we observe as currents and voltages, travel through it at a finite speed. Components like digital logic gates have definite propagation delays (the time it takes after inputs are forcibly

changed before the output matches them properly), and components in different parts of the circuit will compute concurrently. A great deal of engineering effort is expended to build from this substrate computational devices that provide the illusion of synchronous time. A ("monocore", if you will) computer chip contains a clock that emits a periodic electrical signal with reasonably regular, relatively sharp voltage changes. This signal is distributed throughout the chip, at considerable difficulty and expense, so as to rise and fall at a predictable time in every portion of the device. That signal is then used to mark a semantically synchronous passage of time, because "events" in such a computer are restricted to being observed on voltage changes in that clock signal. In fact, the proximate limiting factor to the speed of a monocore is that these voltage changes need to be spaced far enough apart that the waves in the computation circuits have time to travel hither and yon enough to ensure that these observed events will be consistent.

The monocore model has become untenable. A monolithic computer with a single global clock (and modest power consumption) can go only so fast. The obvious thing to do is to make some variation on multicore computers, which consist of many complete locally-synchronous computer blocks. Unfortunately, the standard rendition of this idea is fatally flawed because it provides for a large memory that the cores all share. This large memory implicitly imposes a linear order on the reads and writes that it experiences. Since synchronizing concurrent things well is hard, the implicit synchronization done by such a memory is terrible. As a consequence of the pervasive, terrible synchronization, it appears that all concurrent programming is hard.

The propagation infrastructure presented in this dissertation offers an alternative view. Since propagators by design make only local observations and have only local effects, the structure of a propagator network is analogous to space, and the speed at which information propagates through it is analogous to the speed of light. Propagator A in one part of a network may compute something, and propagator B in another may compute something else. If the consequences of neither computation reach the other before it is made, then in a sense it doesn't make sense to ask which happened "first", because some cells will have learned of the consequences of A before B, and others of B before A. If, however, the `merge` function that determines the laws of physics in this space commutes with respect to A and B, then it doesn't matter, because as long as a cell is exposed to the consequences of both events, it will end up with the same information regardless of the order in which it learns of them. In this way, the idea of partial information lets us build a propagation infrastructure that does not force the notion of synchronous time.

Various synchronizations are still possible in this infrastructure. To take an extreme example, one could imagine a network with a single central cell for holding the current time, with all propagators observing that cell. We could require that each propagator seize a lock on that cell before doing anything, and increment the time in the cell when performing an action, and annotate its results with the time thus read. Such a draconian strategy would produce a perfectly synchronous network,

where every pair of events can be ordered according to which was first. The big win of using networks as a way to think is that such a monstrous synchronization is not necessary. It is up to the user of the network to add appropriate synchronization mechanisms to whatever portions of the computation the user wants synchronized. Propagators don't magically solve the hard problem of synchronizing concurrent systems, they merely move it where it belongs—away from the "concurrent" and to the "synchronizing".

7.2 On Time and Space

At the bottom of modern computation lie the highly parallel laws of electromagnetism, and the solid state physics of devices, that describe the behavior of electrical circuits.[1] An electrical circuit viewed as a computational device, however, is a fixed machine for performing a fixed computation, which is very expensive to rearrange to perform a different computation. An electronic computer is a fixed electrical circuit that performs a universal computation, that is, it implements a layer of indirection which allows it to mimic any computation that is described to it.

At the same time, a standard computer chip imposes an illusion of sequentiality on the parallel laws of electrical circuits: we understand the meaning of machine instructions as sequences of things for the computer to do, with a very powerful notion of the flow of time. Time is nature's way of keeping everything from happening at once; space is nature's way of keeping everything from happening in the same place.[2] A working computation needs at least one of the distinctions, so since a computer chip simulates a single ideal machine that is not differentiated in space, but is conceptually one thing with one series of instructions and one memory, it is inextricably dependent on time to separate its happenings. So in this world, space is a point, and time is a totally ordered line.

Pure linear sequences, of course, are insufficient for describing the computations we want to describe, so we add means of naming portions of such sequences and referring to them, thus letting the computations dance among the linear instructions as we wish. To the extent that they share access to memories, however, such named and recombined sequences still rely on time to separate events. This practice survives today under the retronym "imperative programming".

Virtual memory introduces a notion of space. The memories of separate programs running on the same computer are kept separate from each other by an operating system. Each program, therefore, has its own flow of time, and the computer devotes itself to advancing each from time to time.

Lexical scope introduces a finer-grained notion of space. The local names of

[1] The digital abstraction transforms high-resolution but noisy computations, which we call analog, into noiseless but low-resolution computations, which we call digital. Noiselessness allows arbitrarily large compositions of computations.

[2] with apologies to Woody Allen, Albert Einstein, and John Archibald Wheeler, to whom variations of this quote are variously attributed.

one piece of code are automatically separated from those of another, and from different invocations of the same one.[3] Each lexical scope is therefore a sort of point in space; and their interconnections give space a topology.

More differentiated space permits a lesser reliance on time. The fewer nonlocal effects a procedure has, the less are the temporal constraints on when it may be invoked. Functional programming is the discipline of having no nonlocal effects except for returning a value, and therefore being no slave to time except that the consumer of a value must wait until after its producer.

A great mental shift comes when thinking of the fundamental structure of computations as trees rather than lines. Instead of sequences like "Add 3 to 4; then multiply that by 7; add 1 to 5; compute the sine of this; produce the difference of those two", we can write "(the difference of (the product of (the sum of 3 and 4) and 7) and (the sine of (the sum of 1 and 5)))". The nesting structure of the parentheses produces a tree; a node is a computation combining the results of some subcomputations, each of which may have subcomputations of its own. Trees have a great advantage over lines in that they clearly expose the fact that different subexpressions of an expression can be computed independently of each other and in any order (if they have no interacting side effects)—relative to trees, lines introduce spurious constraints by arbitrarily placing the computation of one partial result before that of another. Tree-shaped computing systems can of course be made out of line-shaped ones with some additional mechanism, such as a recursive interpreter of the tree structure of a program. Conceptually, the space of such a system is topologically a tree, and time is inherited from the underlying linear interpreter, and therefore constitutes some traversal of the tree that is space.

The use of names to refer to reusable partial results of a tree computation turns it into a directed graph computation. The sequential time of the underlying computer constitutes a topological sort of the computation, forcing the graph to be acyclic. If we wish the computer to trace some real phenomenon that has feedback, therefore, we must explicitly arrange iterations around the feedback loop, either stretching them out through space with recursive calls or sequencing them in time with mutation.

The next step in the path that goes from lines through trees to acyclic graphs takes us to arbitrary graphs.[4] The propagator networks of this dissertation directly describe (directed) graph-shaped computations. The simulation thereof on a sequential computer must queue up pending computations and go around loops repeatedly, but these issues of control flow can be made implicit in that simulation.

---

[3]Recursion is the reuse of (a conceptually different incarnation of) a piece of code as a subroutine within that piece of code itself. As such, it requires additional mechanism.

[4]Graphs are not necessarily the end: A graph specifies merely "these two interact; these two do not". The real world, and therefore the models we may wish to build, have interactions of varying strengths, the weaker of which perhaps need only be considered if the stronger do not dictate the answer we are looking for. There are also computations that may benefit from continuous notions of time and space. Such extensions of the shape of computation are beyond the scope of this dissertation.

Space then has any graph topology[5] we may desire, and time is either conceptually avoided by the partial information mechanisms presented herein, or a haphazard consequence of the accidents of order in our simulation.[6]

Thus we are come full-circle: from asynchronous devices with local communication; through computers that forcibly synchronize them; to simulations of asynchronous devices with local communication. On the way we have gained the great advantage of being able simply to describe the computation we want rather than having to build a physical device to perform it, but we have gotten confused about the meaning of time because of a detour into synchronicity.

7.3 On Side Effects

Side effects are the great bane of elegant models of computation. Everything becomes so much easier if one merely bans side effects! Suddenly functional programming just works, logic programming just works, all manner of purely declarative methods can leave the trouble of optimizing execution to a sufficiently clever implementation, and countless angels dance on the heads of whatever pins we wish.

Unfortunately, side effects always intrude. There are, of course, many important programs whose only side effect is to type the final answer out on the screen, but there are many other important programs that do other things to the world as well. So for a programming system to be considered complete, it must be able to encode programs that produce side effects, and if that system is to obey some elegant model, then the model must either accommodate side effects to begin with, or be made to accommodate them by some machination. In practice, elegant models have an unfortunate tendency toward nonaccommodation of side effects, and machinations have a nasty tendency toward kludgery.

The essential reason why side effects tend to mess up models of computation is that side effects inescapably introduce time. An observer can watch the side effects a program emits, and they are inherently sequenced in time. We want to be able to build programs whose observable actions are what we want them to be; for that we need the observable actions of our programs to be predictable from the programs themselves.

Elegance, on the other hand, is usually gained by separating the flow of time in the world from the progress of a computation. Such a separation is a source of great power. Once the nature of the computation no longer inextricably entails the flow

---

[5] In retrospect, it is perhaps reasonable that the prototype you have been reading about is, the laments in Section 6.8 notwithstanding, deeply enmeshed with the Scheme in which it is implemented. After all, even though the overall computation may be a graph, that graph has computations at vertices. Insofar as those vertices are points, within each one time must flow linearly; so it is not unreasonable that `merge`, for example, which happens only at the vertices of the graph, is written completely in Scheme.

[6] Engineering useful intermediates is an interesting question; some thoughts on that follow in Section 7.3. The infrastructure is flexible enough that such intermediates can be engineered in user-space.

of time in the computer performing it, one gains great freedom. One can choose not to compute things until and unless they are needed (lazy evaluation); one can let oneself try computations many times with different guesses for inputs (`amb`); one can reason about the computation one is about to do, and take it apart and put it back together to do it faster (fancy compiler optimizations); one can even specify the computation at a much higher level as a statement of the problem that needs to be solved, and let the system work out the details of how to solve it (various declarative programming systems).

There we have the fundamental tension: between giving our computers the freedom to operate as they will on their internal structures, where we are not concerned with the details of progress but only with its result; and constraining our computers to act the way we wish externally, where we do in fact care that the dialog "Launch missile? OK / Cancel" show up (and be answered in the affirmative!) *before* the missile is actually launched, even though the computer may have thought very carefully about how to target the missile, and where it stood to land and upon whose hapless head it stood to detonate, in advance of asking that question.

The same tension exists in the architecture of human minds. Indeed, people too can both think and act; and we neither know nor care about the detailed order in which thoughts occur in the heads of others (unless they are prone to speaking their thoughts aloud) but we do very much both know and care about the actions others perform, and often the order in which those actions are carried out. The reason I bring up human minds is that we have a word for the transition humans make between thought and action: "decision". Since we have a word for it, perhaps it is an important thing in its own right.

I would then suggest that the kludgery or absence of side effects in elegant models lies in an inadequate separation between thought and action. Abandoning all respect for the flow of time is characteristic of thinking. Acting coherently requires attention to time. But the standard paradigms provide no acknowledgement that thought and action are different, no barrier between the two, no machinery for transforming one into the other, i.e., for making decisions. So the result is either that the primitives for action amount to always doing everything whenever one thinks about doing it, or that the story about thinking offers no primitives for action at all. In the former case, complete freedom of the order of thoughts would entail complete chaos in the order of actions, so thought is constrained so that actions become predictable; in the latter case, the disparate components of a system can each only think or only act, with painful extralinguistic bridges between the two (which fail in many ways, for instance being unable to monitor the progress of the thoughts to decide when to stop them and act).

The propagator paradigm presented here can change the nature of this debate. Propagation in itself accommodates both thought and action: when a network has many fanins and cycles, and operates on interesting partial information structures, propagators can run in a chaotic order, and be run many times, and let the merging of the partial information ensure that they produce a coherent result at the end; this

is thought. On the other hand, when a network is acyclic and the "partial" information structure is the all-or-nothing structure, then propagation mimics conventional evaluation, and the order of events is perfectly predictable. This is action. It is then natural, in this view of the world, to think of building bridges between these two modes. That is decision. The deciders are propagators that will accept partial inputs, that can perhaps be refined with the passage of time, and will take upon themselves the responsibility of picking a moment, making a choice, and committing to an all-or-nothing answer.

There are many ways a decider could be built; rightly, because they reflect different ways of making decisions. If the partial information a decider is observing is measurable as to completeness, for example by the decreasing length of an interval in an interval arithmetic system, then one could make a decider that waits until the partial information is complete enough, and then decides. In some cases, like probability distributions, it may even be the case that some information that is technically still incomplete (lacking, say, a detailed analysis of the outcomes of an unlikely case) is actually enough to make the decision optimally. In other cases, it may be necessary to make heuristic guesses about when enough thought is enough. If the acting portion of a system has to be reactive, it may make sense to build a decider that has an all-or-nothing input from the acting system that means "decide now", and then it can read whatever partial information has been deduced so far and make the best decision it can from that. If, on the other hand, the machine has great liberty to think as it will, it may be possible to build a decider that waits until all of the relevant thinking propagators have quiesced, and then makes a decision knowing that it knows everything that the system ever stands to deduce on the topic about which it is deciding.

Presuming that decisions will be made by appropriate deciders, propagators for the actual side effects are easy enough to define. A box with a side effect can be one that demands all-or-nothing inputs and does nothing until they are all completely specified. Then, once that condition obtains, it performs its side effect (exactly once, because its inputs can never again change without erroring out in a contradiction once they have been specified completely) and, if we want, writes an output that indicates its effect has been performed (which we can perhaps use as a condition for the next action, to reliably sequence them). Such a box need never see the uncertainty and partialness that is useful for solving problems because the deciders can insulate it; likewise collections of such boxes need not suffer from the chaotic flow of time in problem solving, for once a decision has been made the timing of its consequences is predictable.[7]

[7]This story suffices if propagators really are independent asynchronous machines that are always trying to do their thing. If, on the other hand, groups of them are being simulated in a single process, the design just shown suffices only to require that certain side effects happen before others; whereas we may also want to predict the time that elapses between them. For that we need to depend on a certain speed of propagation in the "acting" portions of the system regardless of how much work the "thinking" portions of the system are doing, which is to say the simulator must be able to provide fairness guarantees.

Appendix A

Details

I'm meticulous. I like to dot every i and cross every t. The main text had glossed over a number of details in the interest of space and clarity, so those dots and crossbars are collected here. A particular objective of this Appendix is to ensure that the results in the dissertation are identically reproducible on MIT/GNU Scheme [Hanson et al., 2005], and therefore that everything necessary is documented, either in the dissertation itself, in the MIT/GNU Scheme documentation, or in wider Scheme language materials [Kelsey et al., 1998, Shivers, 1999]. Most of the content here is on that level of detail. Perhaps the main redeeming feature of the Appendix is Section A.5, which shows how the generic propagators are arranged, and discusses why they are arranged that way.

A.1 The Generic Operations System

The generic operations system used in this dissertation is lifted almost verbatim from the generic operations mechanism of the Mechanics [Sussman et al., 2001] Scheme system, with just a couple minor tweaks. This particular piece of machinery is really incidental to the main point of this dissertation—the only essential thing is to have a sufficiently flexible generic dispatch mechanism—but I am including it here for completeness of disclosure, and to satisfy the curiosity of pedantic people like me who want access to *all* the details.

This system is a predicate dispatch system along the lines of [Ernst et al., 1998]. For simplicity's sake, and in contrast with [Ernst et al., 1998], this system does not compute any notion of predicate specificity; if more than one method is applicable, the last one added to the function will be applied. I reproduce the source code file in full below.

```
;;;;           Most General Generic-Operator Dispatch

;;; Generic-operator dispatch is implemented here by a discrimination
;;; list, where the arguments passed to the operator are examined by
;;; predicates that are supplied at the point of attachment of a
;;; handler (by ASSIGN-OPERATION).
```

```
;;; To be the correct branch all arguments must be accepted by
;;; the branch predicates, so this makes it necessary to
;;; backtrack to find another branch where the first argument
;;; is accepted if the second argument is rejected.  Here
;;; backtracking is implemented by OR.

(define (make-generic-operator arity #!optional name default-operation)
  (guarantee-exact-positive-integer arity 'make-generic-operator)
  (if (not (fix:fixnum? arity))
      (error:bad-range-argument arity 'make-generic-operator))
  (if (not (default-object? name))
      (guarantee-symbol name 'make-generic-operator))
  (if (not (default-object? default-operation))
      (guarantee-procedure-of-arity
       default-operation arity 'make-generic-operator))
  (let ((record (make-operator-record arity)))
    (define operator
      (case arity
        ((1)
         (lambda (arg)
           ((or (find-branch (operator-record-tree record) arg win-handler)
                default-operation)
            arg)))
        ((2)
         (lambda (arg1 arg2)
           ((or (find-branch (operator-record-tree record) arg1
                             (lambda (branch)
                               (find-branch branch arg2 win-handler)))
                default-operation)
            arg1
            arg2)))
        (else
         (lambda arguments
           (if (not (fix:= (length arguments) arity))
               (error:wrong-number-of-arguments operator arity arguments))
           (apply (or (let loop ((tree (operator-record-tree record))
                                 (args arguments))
                        (find-branch tree (car args)
                                     (if (pair? (cdr args))
                                         (lambda (branch)
                                           (loop branch (cdr args)))
                                         win-handler)))
                      default-operation)
                  arguments)))))
    (define (find-branch tree arg win)
      (let loop ((tree tree))
        (and (pair? tree)
             (or (and ((caar tree) arg) (win (cdar tree)))
                 (loop (cdr tree))))))
    (define (win-handler handler) handler)
    (set! default-operation
      (if (default-object? default-operation)
          (lambda arguments (no-way-known operator arguments))
          default-operation))
    (set-operator-record! operator record)
    ;; For backwards compatibility with previous implementation:
    (if (not (default-object? name))
        (set-operator-record! name record))
    operator))
```

```
(define *generic-operator-table*
  (make-eq-hash-table))

(define (get-operator-record operator)
  (hash-table/get *generic-operator-table* operator #f))

(define (set-operator-record! operator record)
  (hash-table/put! *generic-operator-table* operator record))

(define (make-operator-record arity) (cons arity '()))
(define (operator-record-arity record) (car record))
(define (operator-record-tree record) (cdr record))
(define (set-operator-record-tree! record tree) (set-cdr! record tree))

(define (operator-arity operator)
  (let ((record (get-operator-record operator)))
    (if record
        (operator-record-arity record)
        (error "Not an operator:" operator))))

(define (assign-operation operator handler . argument-predicates)
  (let ((record
         (let ((record (get-operator-record operator))
               (arity (length argument-predicates)))
           (if record
               (begin
                 (if (not (fix:= arity (operator-record-arity record)))
                     (error "Incorrect operator arity:" operator))
                 record)
               (let ((record (make-operator-record arity)))
                 (hash-table/put! *generic-operator-table* operator record)
                 record)))))
    (set-operator-record-tree! record
                               (bind-in-tree argument-predicates
                                             handler
                                             (operator-record-tree record))))
  operator)

(define defhandler assign-operation)

(define (bind-in-tree keys handler tree)
  (let loop ((keys keys) (tree tree))
    (let ((p.v (assq (car keys) tree)))
      (if (pair? (cdr keys))
          (if p.v
              (begin
                (set-cdr! p.v
                          (loop (cdr keys) (cdr p.v)))
                tree)
              (cons (cons (car keys)
                          (loop (cdr keys) '()))
                    tree))
          (if p.v
              (begin
                (warn "Replacing a handler:" (cdr p.v) handler)
                (set-cdr! p.v handler)
                tree)
              (cons (cons (car keys) handler)
                    tree))))))
```

```
(define (no-way-known operator arguments)
  (error "Generic operator inapplicable:" operator arguments))

;;; Utility
(define (any? x)
  #t)
```

A.2 The Scheduler

The job scheduling system supporting this dissertation is deliberately not very sophisticated. The main purpose of the dissertation is to demonstrate the expressive power of programming with notionally independent propagators, and this scheduler is just a device for playing with that paradigm in a uniprocess computing environment. As discussed in Section 2.2 and the "Interlude on simulation" in Section 3.1, it is, in fact, especially important that the scheduler not make too many promises, because a helpful promise that supports one application is a shackle that restrains another.

Before dumping the actual scheduler code, this is the place to come clean about a small cheat in the text. For the sake of clarity of exposition, the examples in the dissertation omit references to the scheduler; but in order for those examples to actually work in a standard Scheme system, the scheduler must be called. For instance, attaching a constant to a cell and reading its value would be written in the text as

```
(define foo (make-cell))
((constant 42) foo)
(content foo)
;; Expect 42
```

but inputting literally that into a standard Scheme wouldn't work, at worst because the scheduler's internal state would still be uninitialized, or at best because the `constant` propagator would not have had a chance to run before the user read the `content` of `foo`. At a minimum, the scheduler must be initialized before beginning the interaction, and invoked before requesting any answers, so:

```
(initialize-scheduler)
(define foo (make-cell))
((constant 42) foo)
(run)
(content foo)
42
```

But, it is acceptable to overshoot, calling `(run)` after every toplevel form:

```
(initialize-scheduler)

(define foo (make-cell))
(run)

((constant 42) foo)
(run)

(content foo)
42
```

This leads to no adverse consequences, so a production propagation system could offer user interaction via a read-eval-run-print loop, inserting the appropriate extra step into the standard Scheme interaction. With a bit of poetic licence, therefore, one can say that the main text presumes the existence of such an interaction mechanism, even though I have not implemented it.

Now, without further ado, the scheduler implementation, in all its detail:

```
;;;; Basic scheduling system

;;; This scheduler maintains a list of jobs that need to be run.  Each
;;; job is a thunk.  Jobs are run serially and are not preempted.
;;; When a job exits (normally) it is forgotten and the next job is
;;; run.  The jobs are permitted to schedule additional jobs,
;;; including rescheduling themselves.  Jobs are presumed idempotent,
;;; and specifically it is assumed acceptable not to count how many
;;; times a given job (by eq?-ness) was scheduled, but merely that it
;;; was scheduled.  When the scheduler runs out of jobs, it returns
;;; the symbol 'done to its caller.

;;; The scheduler supplies an escape mechanism: running the procedure
;;; abort-process, with a value, will terminate the entire job run,
;;; and return the supplied value to the scheduler's caller.
;;; Subsequent calls to the scheduler without first scheduling more
;;; jobs will also return that same value.  If abort-process is called
;;; outside the dynamic extent of a run, it deschedules any jobs that
;;; might be scheduled and saves the value for future reference as
;;; above.

;;; This scheduler is meant as a low-level support for the propagator
;;; network in this dissertation.  In that use case, the jobs would be
;;; propagators that the network knows need to be run.  Any cells in
;;; the network are invisible to the scheduler, but presumably help
;;; the network schedule more propagators to run (namely those that
;;; may be interested in the cell's goings on).

;;; The public interface is
;;;   (initialize-scheduler)      clear all scheduler state
;;;   (alert-propagators jobs)    schedule a list (or set) of jobs
;;;   (alert-all-propagators!)    reschedule all jobs ever scheduled
;;;   (run)                       run scheduled jobs until done
;;;   (abort-process x)           terminate the run returning x
```

```
(define *alerted-propagators*)
(define *alerted-propagator-list*)
(define *abort-process*)
(define *last-value-of-run*)
(define *propagators-ever-alerted*)
(define *propagators-ever-alerted-list*)

(define (initialize-scheduler)
  (clear-alerted-propagators!)
  (set! *abort-process* #f)
  (set! *last-value-of-run* 'done)
  (set! *propagators-ever-alerted* (make-eq-hash-table))
  (set! *propagators-ever-alerted-list*
        (list '*propagators-ever-alerted-list*))
  'ok)

(define (any-propagators-alerted?)
  (< 0 (hash-table/count *alerted-propagators*)))

(define (clear-alerted-propagators!)
  (set! *alerted-propagators* (make-strong-eq-hash-table))
  (set! *alerted-propagator-list* (list '*alerted-propagator-list*)))

;; Turning this off makes the order in which propagators are run vary
;; chaotically.  That is not supposed to cause trouble in principle,
;; but a reproducible run order can be valuable for debugging the
;; infrastructure.  The chaotic variation also causes variations in the
;; *number-of-calls-to-fail* when doing dependency directed backtracking.
(define *reproducible-run-order* #t)

(define (order-preserving-insert thing table lst)
  (hash-table/lookup
   table
   thing
   (lambda (value) 'ok)
   (lambda ()
     (hash-table/put! table thing #t)
     (push! lst thing))))

(define (push! headed-list thing)
  (set-cdr! headed-list (cons thing (cdr headed-list))))

(define (ordered-key-list table lst)
  (if *reproducible-run-order*
      (list-copy (cdr lst))
      (hash-table/key-list table)))

(define (alert-propagators propagators)
  (for-each
   (lambda (propagator)
     (if (not (procedure? propagator))
         (error "Alerting a non-procedure" propagator))
     (order-preserving-insert
      propagator *propagators-ever-alerted* *propagators-ever-alerted-list*)
     (order-preserving-insert
      propagator *alerted-propagators* *alerted-propagator-list*))
   (listify propagators))
  #f)
(define alert-propagator alert-propagators)
```

```
(define (alert-all-propagators!)
  (alert-propagators
   (ordered-key-list *propagators-ever-alerted*
                     *propagators-ever-alerted-list*)))

(define (the-alerted-propagators)
  (ordered-key-list *alerted-propagators*
                    *alerted-propagator-list*))

(define (with-process-abortion thunk)
  (call-with-current-continuation
   (lambda (k)
     (fluid-let ((*abort-process* k))
       (thunk)))))

(define termination-trace #f)

(define (abort-process value)
  (if termination-trace
      (ppc `(calling abort-process with ,value and ,*abort-process*)))
  (if *abort-process*
      ;; if the propagator is running
      (begin (clear-alerted-propagators!)
             (*abort-process* value))
      ;; if the user is setting up state
      (begin (clear-alerted-propagators!)
             (set! *last-value-of-run* value))))

(define (run-alerted)
  (let ((temp (the-alerted-propagators)))
    (clear-alerted-propagators!)
    (for-each (lambda (propagator)
                (propagator))
              temp))
  (if (any-propagators-alerted?)
      (run-alerted)
      'done))

(define (run)
  (if (any-propagators-alerted?)
      (set! *last-value-of-run* (with-process-abortion run-alerted)))
  *last-value-of-run*)
```

A.3 Primitives for Section 3.1

```
(define adder (function->propagator-constructor (handling-nothings +)))
(define subtractor (function->propagator-constructor (handling-nothings -)))
(define multiplier (function->propagator-constructor (handling-nothings *)))
(define divider (function->propagator-constructor (handling-nothings /)))

(define absolute-value
  (function->propagator-constructor (handling-nothings abs)))
(define squarer
  (function->propagator-constructor (handling-nothings square)))
(define sqrter
  (function->propagator-constructor (handling-nothings sqrt)))

(define =? (function->propagator-constructor (handling-nothings =)))
(define <? (function->propagator-constructor (handling-nothings <)))
(define >? (function->propagator-constructor (handling-nothings >)))
(define <=? (function->propagator-constructor (handling-nothings <=)))
(define >=? (function->propagator-constructor (handling-nothings >=)))

(define inverter
  (function->propagator-constructor (handling-nothings not)))
(define conjoiner
  (function->propagator-constructor (handling-nothings boolean/and)))
(define disjoiner
  (function->propagator-constructor (handling-nothings boolean/or)))
```

A.4 Data Structure Definitions

Our data structures are simple tagged vectors. Mercifully, we do not use vectors elsewhere in the system, so no confusion is possible. Also mercifully, the `define-structure` macro from MIT/GNU Scheme [Hanson et al., 2005] automates the construction and abstraction of such data structures.

Intervals:

```
(define-structure
 (interval
  (type vector) (named 'interval) (print-procedure #f) (safe-accessors #t))
 low high)

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

Supported values:

```
(define-structure
 (v&s (named 'supported) (type vector)
      (constructor supported) (print-procedure #f)
      (safe-accessors #t))
 value support)
```

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

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

Lists of supported values for truth maintenance:

```
(define-structure
  (tms (type vector) (named 'tms)
       (constructor %make-tms) (print-procedure #f)
       (safe-accessors #t))
  values)

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

Synthetic premises for hypothetical reasoning

```
(define-structure
  (hypothetical (type vector) (named 'hypothetical)
                (print-procedure #f) (safe-accessors #t)))
```

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

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

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

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

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

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

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

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

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

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

We also provide a procedure for resetting the states of all premises. This is

mostly useful for being able to use symbols as premises in multiple different examples without worrying about spilling information.

```
(define (reset-premise-info!)
  (set! *premise-outness* (make-eq-hash-table))
  (set! *premise-nogoods* (make-eq-hash-table)))
```

Finally, we need the counter that tracks our fun failure statistics:

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

A.5 Generic Primitives

I have not solved the problem of producing a nice, uniform generic mechanism for handling partial information types. The main desiderata for such a mechanism are that it be additive, namely that new partial information types can be added to it without disturbing its handling of the old; that it permit interoperation between partial information types that can reasonably interoperate; and that new partial information types can be additively adjoined to interoperating groups. I have achieved that to my satisfaction for the partial information types used in this dissertation, but I am not convinced that my solution is general. I do not consider this a great difficulty, however—managing diverse partial information types is a separate issue from the value of propagation and partial information in general.

The sections that follow handle all the partial information types used in this dissertation with two slightly competing mechanisms. In brief, the first mechanism is to define a generic operation for each primitive, and add appropriate methods to it for each partial information type. The second mechanism is to wrap those generic operations in a common generic wrapper, and handle suitable partial information types by adding methods to the wrapper, independently of what the underlying primitive operations are. These methods are presented and used in the following sections; I reflect on their relative merits and shortcomings in a retrospective Appendix A.5.5.

A.5.1 Definitions

We start by defining the primitives we will use as generic functions, so that we can give them arbitrary behavior on our various partial information structures.

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

We also define general generic operators in which to wrap these operations so we can handle some partial information types (namely `nothing`, `v&s`, and `tms`) uniformly irrespective of the particular underlying operation. These operators are loosely inspired by Haskell's Monad typeclass [Hudak et al., 1999, Piponi, 2006].

```
(define (generic-bind thing function)
  (generic-flatten (generic-unpack thing function)))

(define generic-unpack
  (make-generic-operator 2 'unpack
    (lambda (object function)
      (function object))))

(define generic-flatten
  (make-generic-operator 1 'flatten (lambda (object) object)))
```

We will additionally use a wrapper that handles n-ary Scheme procedures by binding each argument in turn.

```
(define (nary-unpacking function)
  (lambda args
    (let loop ((args args)
               (function function))
      (if (null? args)
          (function)
          (generic-bind
           (car args)
           (lambda (arg)
             (loop (cdr args)
                   (lambda remaining-args
                     (apply function (cons arg remaining-args))))))))))
```

Both of those bits of generic machinery being available, we let our propagators be machines that will first try to process their arguments with `nary-unpacking`, and bottom out to the generic operations underneath if there is no uniform handler defined for the partial information structure they are exposed to.

```
(define adder
  (function->propagator-constructor (nary-unpacking generic-+)))
(define subtractor
  (function->propagator-constructor (nary-unpacking generic--)))
(define multiplier
  (function->propagator-constructor (nary-unpacking generic-*)))
(define divider
  (function->propagator-constructor (nary-unpacking generic-/)))

(define absolute-value
  (function->propagator-constructor (nary-unpacking generic-abs)))
(define squarer
  (function->propagator-constructor (nary-unpacking generic-square)))
(define sqrter
  (function->propagator-constructor (nary-unpacking generic-sqrt)))

(define =? (function->propagator-constructor (nary-unpacking generic-=)))
(define <? (function->propagator-constructor (nary-unpacking generic-<)))
(define >? (function->propagator-constructor (nary-unpacking generic->)))
(define <=? (function->propagator-constructor (nary-unpacking generic-<=)))
(define >=? (function->propagator-constructor (nary-unpacking generic->=)))

(define inverter
  (function->propagator-constructor (nary-unpacking generic-not)))
(define conjoiner
  (function->propagator-constructor (nary-unpacking generic-and)))
(define disjoiner
  (function->propagator-constructor (nary-unpacking generic-or)))
```

Now we are ready for our first partial information type. Adding support for `nothing` (which is analogous to Haskell's `Maybe` monad) is just a matter of adding the right methods to our uniform generic facility:

```
(defhandler generic-unpack
  (lambda (object function) nothing)
  nothing? any?)

;;; This handler is redundant but harmless
(defhandler generic-flatten
  (lambda (thing) nothing)
  nothing?)
```

A.5.2 Intervals

I will use interval arithmetic as an example of implementing support for a partial information type using the underlying generic functions themselves, rather than the uniform `nary-unpacking` mechanism. It is possible that intervals fit into the uniform mechanism, but I don't want to think about how to do it, and the example is valuable. So I attach interval arithmetic handlers to the appropriate generic operations directly—having both mechanisms around offers maximum flexibility.

```
(defhandler generic-* mul-interval interval? interval?)
(defhandler generic-/ div-interval interval? interval?)
(defhandler generic-square square-interval interval?)
(defhandler generic-sqrt sqrt-interval interval?)
```

We also define machinery for coercing arguments of compatible types to intervals when appropriate.

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

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

(defhandler generic-* (coercing ->interval mul-interval) number? interval?)
(defhandler generic-* (coercing ->interval mul-interval) interval? number?)
(defhandler generic-/ (coercing ->interval div-interval) number? interval?)
(defhandler generic-/ (coercing ->interval div-interval) interval? number?)
```

A.5.3 Supported values

Supported values are the first place where we can show off the advantages of the uniform mechanism from Appendix A.5.1. Justifications are a form of information that is amenable to a uniform treatment,[1] and we can specify that treatment by attaching appropriate handlers to `generic-unpack` and `generic-flatten`.

`Generic-unpack` is textbook:

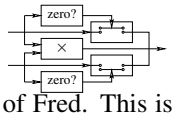
```
(defhandler generic-unpack
  (lambda (v&s function)
    (supported
     (generic-bind (v&s-value v&s) function)
     (v&s-support v&s)))
  v&s? any?)
```

If the incoming value is supported, strip off the supports, operate on the underlying value, and add the supports back on again afterward.

`Generic-flatten` turns out to be much more interesting, because we don't know ahead of time the exact shape of the object we want to flatten. Specifically, how to flatten a supported value depends on what is actually supported: if something primitive like a number is supported, then the flattening is the supported value itself. If the supported object inside is `nothing` then we would like the flattened result to be `nothing`, without the support.[2] But of greatest interest, if the supported value is itself a supported value, we want the result to be a single supported value

---

[1] If we are willing to forgo the refinement that `(* 0 x)` is `0` irrespective of the dependencies of `x`. It is not clear how to implement that refinement in the `nary-unpacking` system directly, but it can be implemented in the propagator system with the diagram on the right; or by an analogous composition of generic operations in Scheme.



[2] On the logic that if Fred tells you `nothing`, you still know `nothing`, regardless of Fred. This is a consistent choice, but alternatives are possible; one may, for instance, care to record that Fred has considered the matter some and still produced `nothing`. Explorations of such alternatives are deferred to future work.

that is supported by the union of the supports.

We take advantage of the flexibility of predicate dispatch to encode all three of those possibilities as separate handlers for `generic-flatten`. Besides being aesthetically more pleasing than a big `cond`, this will let us seamlessly add support for TMSes in Appendix A.5.4.

```
(defhandler generic-flatten
  (lambda (v&s) v&s)
  v&s?)

(defhandler generic-flatten
  (lambda (v&s) nothing)
  (lambda (thing) (and (v&s? thing) (nothing? (v&s-value thing)))))

(defhandler generic-flatten
  (lambda (v&s)
    (generic-flatten
     (supported
      (v&s-value (v&s-value v&s))
      (merge-supports v&s (v&s-value v&s)))))
  (lambda (thing) (and (v&s? thing) (v&s? (v&s-value thing)))))
```

Merging supported values

In order to merge supported values with other things properly, we need a notion of what's appropriate to coerce into a supported value. The notion used here is simpleminded, but at least somewhat extensible.

```
(define *flat-types-list* '())

(define (flat? thing)
  (apply boolean/or (map (lambda (type) (type thing)) *flat-types-list*)))

(define (specify-flat type)
  (if (memq type *flat-types-list*)
      'ok
      (set! *flat-types-list* (cons type *flat-types-list*))))

(specify-flat symbol?)
(specify-flat number?)
(specify-flat boolean?)
(specify-flat interval?)

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

(defhandler merge (coercing ->v&s v&s-merge) v&s? flat?)
(defhandler merge (coercing ->v&s v&s-merge) flat? v&s?)
```

A.5.4 Truth maintenance systems

Adjusting the primitives for truth maintenance also turns out to be a uniform transformation. `Generic-unpack` is again textbook:

```
(defhandler generic-unpack
  (lambda (tms function)
    (let ((relevant-information (tms-query tms)))
      (make-tms (list (generic-bind relevant-information function)))))
  tms? any?)
```

If the input is a TMS, query it, operate on that, and pack the result up into a (one-item) TMS. The only choice made here is that we are following the philosophy set out in the main text of operating on the strongest possible consequences of what we currently believe (as found by `tms-query`) rather than on all variations of everything we might believe.

`Generic-flatten` is again more interesting. If the input is a TMS, we recursively flatten all its contents, coerce the results into TMS value lists, and make a new TMS out of that.[3]

```
(defhandler generic-flatten
  (lambda (tms)
    (let ((candidates
           (append-map tms-values
                       (map ->tms
                            (map generic-flatten (tms-values tms))))))
      (if (null? candidates)
          nothing
          (make-tms candidates))))
  tms?)
```

But we also need to extend the code for flattening a `v&s`, because we want to do something interesting if we find a `v&s` with a TMS inside (namely incorporate the support of that `v&s` into the consequences of that TMS).

```
(defhandler generic-flatten
  (lambda (v&s)
    (generic-flatten
     (make-tms
      (generic-flatten
       (supported (tms-query (v&s-value v&s)) (v&s-support v&s))))))
  (lambda (thing) (and (v&s? thing) (tms? (v&s-value thing)))))
```

This handler also uses `tms-query` to consider only the currently believed consequences of the interior TMS it is operating on.

Merging truth maintenance systems

Finally, to merge TMSes in cells, we need a mechanism to coerce things to TMSes, and methods on `merge` that use it. These methods use a layer of indirection because we redefine `tms-merge` in the text (to handle contradictions).

---

[3] If you are wondering why this operates on the whole contents of the TMS rather than querying it like `generic-unpack` does so pointedly above, read about the troubles of the `nary-unpacking` mechanism in Appendix A.5.5.

```
(define (->tms thing)
  (cond ((tms? thing) thing)
        ((nothing? thing) (make-tms '()))
        (else (make-tms (list (->v&s thing))))))

(define (the-tms-handler thing1 thing2)
  (tms-merge thing1 thing2))

(defhandler merge the-tms-handler tms? tms?)
(defhandler merge (coercing ->tms the-tms-handler) tms? v&s?)
(defhandler merge (coercing ->tms the-tms-handler) v&s? tms?)
(defhandler merge (coercing ->tms the-tms-handler) tms? flat?)
(defhandler merge (coercing ->tms the-tms-handler) flat? tms?)
```

A.5.5 Discussion

The uniform generic machinery in Appendix A.5.1, namely the `nary-unpacking` function and its supporting `generic-unpack` and `generic-flatten`, may seem overly complex at first. Indeed, for functions like `+`, only the moral equivalent of `generic-unpack` is necessary, and `generic-flatten` is just superfluous complexity. Why, then, is it necessary to add it?

The reason is epitomized by `switch` and `car`. The `switch` propagator is nontrivial essentially because its base function can return a partial information state even when given fully known arguments. What I mean is that if `+` is given numbers, it is guaranteed to return a number; the code to lift `+` to accept TMSes over numbers and produce TMSes over numbers is consequently simpler than the full `nary-unpacking` caboodle. `Switch`, in contrast, can return `nothing` even if its inputs are both perfectly normal fully known values (in fact, exactly when its control argument is `#f`). It therefore does not suffice to implement a `switch` on TMSes by simply unpacking two TMSes, operating on the internal flat values, and packing the TMSes back up again.

For `switch` this could perhaps have been kludged, but `car` (at least as requested by the "Recursive Partial Information" strategy in Section 6.3.1) is even worse. Consider trying to take the `car` of a TMS over pairs of TMSes over numbers. You start by unpacking the top-level TMS to get a raw pair. Then you call the base `car` on it, but the answer, instead of being a raw value, is another TMS! So you have to flatten. Indeed, `switch`'s behavior can also be seen as something that requires flattening.

The same requirements also apply to `apply`; that is, if one is `apply`ing a supported procedure, the value returned by that procedure must depend upon the justification of the procedure. But if the inputs to the procedure also carried justifications, or something else that needed justifying happened during the procedure's execution, then the procedure may produce a justified answer, and the justification of the procedure itself must then be flattened onto the justification of the result it returns. The similarity of `if` with `apply` is not surprising given the Church encoding of booleans as procedures that ignore one or the other of their arguments; the solution is the same.

Haskell's storied `Monad` typeclass [Hudak et al., 1999, Piponi, 2006] also contains ideas of unpacking and flattening, albeit sliced a bit differently. As such it constitutes evidence that unpacking and flattening are valuable kinds of plumbing. Therefore, I bit the monadic bullet and wrote code that does both unpacking and flattening generically (as opposed to just generic unpacking, which would have sufficed for functions like +).

Unfortunately, `nary-unpacking` does not quite satisfy the demands I want to place on generic mechanisms. It's good enough for the present purpose, but getting it right in general is worth further thought. In my experience so far, it falls down in four ways:

1. Interval arithmetic was far easier to do with specific generic operations than with the general mechanism;

2. the general mechanism treats the arguments of symmetric binary functions asymmetrically;

3. the means by which `generic-unpack` talks to `generic-flatten` can violate the invariants of the partial information structures I am using; and

4. `nary-unpacking` seems completely inapplicable to the `merge` generic procedure.

Intervals

I am not too concerned about `nary-unpacking`'s apparent inability to handle interval arithmetic. Unlike the other partial information types that show up in this dissertation, intervals have a strong restriction on what they "contain", namely that the upper and lower bounds are supposed to be real numbers. Different basic operations also handle real intervals differently: addition operates on the two pairs of corresponding bounds, but multiplication, in principle, needs to play with signs, and division, in the fully general case of intervals that may span zero, does something completely weird. So it may not matter that `nary-unpacking` doesn't do intervals.

Asymmetry

For an example of the asymmetry problem, consider what happens if one tries to add a TMS to `nothing`. If the TMS is the first argument seen by `nary-unpacking`, it will dutifully unpack that TMS, save the justification of the result, and make its recursive call. The recursive call will then notice that the second argument was `nothing` and return `nothing` immediately, which will then get packed up into a TMS with the saved justification, and only flattened into `nothing` at the end. In contrast, the specific generic function mechanism admits a handler that looks for `nothing` anywhere in the input and returns `nothing` immediately. On the other hand, the specific mechanism offers no good way to attach that handler to all the primitive generic operations at once, without looping over a list of them.

Perhaps this asymmetry problem can be solved for binary functions (which are of course the majority of interesting symmetric functions) by defining an appropriate `generic-binary-unpacking`. Such a wrapper could have the appropriate symmetric handlers, but it would need to replicate the work of `generic-flatten` internally to merge, for example, two supports from the two arguments to, say, `+`.

Invariants

For an example of the violated invariants problem, consider again taking the `car` of a pair inside a TMS. You start with unpacking the TMS and saving the justification of the pair. Then you need to flatten that justification onto whatever you get out of the pair with `car`, so what you really need to pass to `generic-flatten` is the justification and the `car`, separately. The code as I have written it, however, just blindly wraps that `car` up in a record structure that looks like a TMS and carries the needed justification; with no respect for whether or not that `car` violates the intended invariants of the TMS data type. Since this is only a communication channel between `generic-unpack` and `generic-flatten`, that doesn't cause too much trouble, but the methods on `generic-flatten` need to be aware that they will be asked to handle data structures whose invariants may be wrong. That's why the `generic-flatten` method on TMSes defined in Appendix A.5.4 looks so weird.

Perhaps this invariant violation problem can be solved by a better interface between `generic-unpack` and `generic-flatten`; or perhaps there is a deep problem here, and Haskell made `bind` the basic primitive for a reason. The separation of `generic-unpack` and `generic-flatten` has served my purpose well, on the other hand, because it allowed seamless interoperation between the `v&s` and `tms` partial information types, and additive adjoining of the latter to a system that already contained the former. On the other hand, perhaps that same additivity can be achieved with monad transformers [Liang et al., 1995] as well.

Merge

Inapplicability to `merge` may well not be a problem. Thinking this out carefully is open to future work; but I suspect that the reason why `unpack` and `flatten` don't seem to be useful for `merge` is that `merge` is not their client but their peer—not merely another function, but a *new kind of plumbing*. If so, it is perhaps a kind of plumbing that is only applicable to propagator networks, because only they expect to experience multiple sources writing to the same place and needing to be merged.

A.6 Miscellaneous Utilities

I told you on page 147 that I was meticulous! There's really nothing at all interesting to see here. But if you really want to know *every* detail of how the code in this dissertation works, this section is the only thing left between the text and a bare MIT/GNU Scheme [Hanson et al., 2005].

```
(define (for-each-distinct-pair proc lst)
  (if (not (null? lst))
      (let loop ((first (car lst)) (rest (cdr lst)))
        (for-each (lambda (other-element)
                    (proc first other-element))
                  rest)
        (if (not (null? rest))
            (loop (car rest) (cdr rest))))))

(define (sort-by lst compute-key)
  (map cdr
       (sort (map (lambda (thing)
                    (cons (compute-key thing) thing))
                  lst)
             (lambda (pair1 pair2)
               (< (car pair1) (car pair2))))))

(define (listify object)
  (cond ((null? object) object)
        ((pair? object) object)
        (else (list object))))

(define (identity x) x)

(define (ignore-first x y) y)

(define (default-equal? x y)
  (if (and (number? x) (number? y))
      (close-enuf? x y 1e-10)
      (equal? x y)))

(define (close-enuf? h1 h2 #!optional tolerance scale)
  (if (default-object? tolerance)
      (set! tolerance *machine-epsilon*))
  (if (default-object? scale)
      (set! scale 1.0))
  (<= (magnitude (- h1 h2))
      (* tolerance
         (+ (* 0.5
               (+ (magnitude h1) (magnitude h2)))
            scale))))

(define *machine-epsilon*
  (let loop ((e 1.0))
     (if (= 1.0 (+ e 1.0))
         (* 2 e)
         (loop (/ e 2)))))
```

Bibliography

[Abdelmeged et al., 2007] Ahmed Abdelmeged, Christine Hang, Daniel Rinehart, and Karl Lieberherr (2007). Superresolution and P-Optimality in Boolean MAX-CSP Solvers. *Transition*.

[Abelson et al., 1996] Harold Abelson, Gerald Jay Sussman, and Julie Sussman (1984, 1996). *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA.

[Allen et al., 2005] E. Allen, D. Chase, V. Luchangco, J.W. Maessen, S. Ryu, G.L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, J. Hallett, et al. (2005). *The Fortress Language Specification*. Sun Microsystems.

[Andersen, 1837] Hans Christian Andersen (1837). The Emperor's New Clothes. In *Tales for Children*.

[Apt, 2003] Krzysztof R. Apt (2003). *Principles of Constraint Programming*. Cambridge University Press, Cambridge, UK.

[Apt et al., 1999] Krzysztof R. Apt, Victor W. Marek, Miroslaw Truszczynski, and David S. Warren, editors (1999). *The Logic Programming Paradigm: a 25-Year Perspective*. Springer, Berlin; New York.

[Athanassakis, 1983] Apostolos N. Athanassakis (1983). *Hesiod: Theogony, Works and Days and The Shield of Heracles*. Johns Hopkins University Press, Baltimore and London. p.90.

[Bacchus et al., 2003] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi (2003). Value Elimination: Bayesian Inference via Backtracking Search. In *Uncertainty in Artificial Intelligence*, pages 20–28.

[Bonawitz, 2008] Keith A. Bonawitz (2008). Composable Probabilistic Inference with Blaise. CSAIL Tech Report MIT-CSAIL-TR-2008-044, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA.

[Borning, 1979] Alan H. Borning (1979). *ThingLab–a Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, Stanford, CA, USA.

[Borning, 1981] Alan H. Borning (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387.

[Bricklin and Frankston, 1999] Dan Bricklin and Bob Frankston (1999). VisiCalc: Information from its Creators, Dan Bricklin and Bob Frankston. `http://bricklin.com/visicalc.htm`.

[Calandra, 1961] Alexander Calandra (1961). *The Teaching of Elementary Science and Mathematics*. Washington University Press, St. Louis.

[Calandra, 1968] Alexander Calandra (1968). Angels on a Pin. *Saturday Review*.

[Cooper, 2008] Gregory H. Cooper (2008). *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University.

[Cooper and Krishnamurthi, 2004] Gregory H. Cooper and Shriram Krishnamurthi (2004). FrTime: Functional Reactive Programming in PLT Scheme. Computer science technical report CS-03-20, Brown University.

[Cooper and Krishnamurthi, 2006] Gregory H. Cooper and Shriram Krishnamurthi (2006). Embedding Dynamic Dataflow in a Call-by-Value Language. *Lecture Notes in Computer Science*, 3924:294.

[Cormen et al., 2001] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein (2001). *Introduction to Algorithms*. MIT Press, Cambridge, MA.

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

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

[Dinesman, 1968] Howard P. Dinesman (1968). *Superior Mathematical Puzzles, with Detailed Solutions*. Simon and Schuster, New York, NY.

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

[Doyle, 1890] Sir Arthur Conan Doyle (1890). *The Sign of the Four*. Lippincott's Monthly Magazine. Available online at `http://www.gutenberg.org/files/2097/2097-h/2097-h.htm`.

[Driscoll et al., 1989] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan (1989). Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1).

[Elliott and Hudak, 1997] Conal Elliott and Paul Hudak (1997). Functional Reactive Animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. Association for Computing Machinery, New York, NY.

[Erman et al., 1980] L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy (1980). The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys (CSUR)*, 12(2):213–253.

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

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

[Forbus and de Kleer, 1993] Kenneth D. Forbus and Johan de Kleer (1993). *Building Problem Solvers*. MIT Press, Cambridge, MA.

[Friedman and Wand, 2007] Daniel P. Friedman and Mitchell Wand (2007). *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 3rd edition.

[Goodman et al., 2008] N.D. Goodman, V.K. Mansinghka, D. Roy, K. Bonawitz, and J.B. Tenenbaum (2008). Church: a Language for Generative Models. In *Uncertainty in Artificial Intelligence*.

[Gosling et al., 2005] J. Gosling, B. Joy, G. Steele, and G. Bracha (2005). *The Java (TM) Language Specification*. Addison-Wesley Professional.

[Gupta et al., 1996] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat (1996). Models for Concurrent Constraint Programming. *Lecture Notes in Computer Science*, pages 66–83.

[Hand, 2009] Linda Hand (2009). Have you found all three easter eggs in this document? In *Look Carefully*, page 7. MIT Press, Cambridge, MA.

[Hanson, 2007] Chris Hanson (2007). Personal communication.

[Hanson et al., 2005] Chris Hanson et al. (2005). *MIT/GNU Scheme Reference Manual*. Massachusetts Institute of Technology, Cambridge, MA. `http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/index.html`. The code presented in this dissertation was run specifically on MIT/GNU Scheme Release 7.7.90.+, Snapshot 20080401, under a GNU/Linux operating system.

[Hayes-Roth, 1985] Frederick Hayes-Roth (1985). Rule-Based Systems. *Communications of the ACM*, 28(9):921–932.

[Heath, 1921] Thomas Little Heath (1921). *A History of Greek Mathematics*. Clarendon Press, Oxford.

[Hewitt, 1969] Carl E. Hewitt (1969). Planner: A Language for Proving Theorems in Robots. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 295–301.

[Hudak et al., 1999] Paul Hudak, John Peterson, and Joseph H. Fasel (1999). *A Gentle Introduction to Haskell 98*. Online tutorial.

[Jaynes, 2003] Edwin T. Jaynes (2003). *Probability Theory: The Logic of Science*. Cambridge University Press, Cambridge, UK.

[Johnston et al., 2004] W.M. Johnston, J.R.P. Hanna, and R.J. Millar (2004). Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34.

[Jones, 2002] Simon L. Peyton Jones (2002). Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. *Engineering theories of software construction*.

[Kelsey et al., 1998] Richard Kelsey, William D. Clinger, Jonathan Rees, et al. (1998). Revised^5 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 33(9):26–76.

[Kiczales et al., 1999] Gregor Kiczales, Daniel G. Bobrow, and Jim des Rivières (1999). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA.

[Konopasek and Jayaraman, 1984] Milos Konopasek and Sundaresan Jayaraman (1984). *The TK! Solver Book: a Guide to Problem-Solving in Science, Engineering, Business, and Education*. Osborne/McGraw-Hill.

[Liang et al., 1995] S. Liang, P. Hudak, and M. Jones (1995). Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 333–343. Association for Computing Machinery, New York, NY.

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

[Lloyd, 1987] J.W. Lloyd (1987). *Foundations of Logic Programming*. Springer-Verlag, New York, NY.

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

[McAllester, 1990] David Allen McAllester (1990). Truth Maintenance. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, volume 2, pages 1109–1116. AAAI Press.

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

[Mikkelson and Mikkelson, 2007] Barbara Mikkelson and David Mikkelson (2007). The Barometer Problem. `http://www.snopes.com/college/exam/barometer.asp`.

[Milch et al., 2005] B. Milch, B. Marthi, S. Russell, D. Sontag, D.L. Ong, and A. Kolobov (2005). BLOG: Probabilistic Models with Unknown Objects. In *Proceedings of the Nineteenth Joint Conference on Artificial Intelligence*.

[Müller, 2001] Tobias Müller (2001). *Constraint Propagation in Mozart*. PhD thesis, Universität des Saarlandes, Saarbrücken.

[Nilsson et al., 2002] H. Nilsson, A. Courtney, and J. Peterson (2002). Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64. Association for Computing Machinery, New York, NY.

[Nilsson and Małuszyński, 1995] Ulf Nilsson and Jan Małuszyński (1995). *Logic, Programming and Prolog*. Wiley, second edition.

[Norvig, 2004] Peter Norvig (2004). *Paradigms of Artificial Intelligence Programming: Case Studies in Common LISP*. Morgan Kaufmann.

[Ockham, 1340] William of Ockham (ca. 1340). Ockham's razor.

[Park et al., 2005] S. Park, F. Pfenning, and S. Thrun (2005). A Probabilistic Language Based upon Sampling Functions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 171–182. Association for Computing Machinery, New York, NY.

[Pfeffer, 2001] Avi Pfeffer (2001). IBAL: A Probabilistic Rational Programming Language. In *International Joint Conferences on Artificial Intelligence*, pages 733–740.

[Pfeffer, 2007] Avi Pfeffer (2007). The Design and Implementation of IBAL: A General-Purpose Probabilistic Language. In *An Introduction to Statistical Relational Learning*, pages 399–432. MIT Press, Cambridge, MA.

[Piponi, 2006] Dan Piponi (2006). You Could Have Invented Monads! (And Maybe You Already Have.). Weblog post. `http://sigfpe.blogspot.com/2006/08/you-could-have-invented-monads-and.html`.

[Puget, 1998] J.F. Puget (1998). A Fast Algorithm for the Bound Consistency of Alldiff Constraints. In *Proceedings of the National Conference on Artificial Intelligence*, pages 359–366. John Wiley & Sons Ltd.

[Radul, 2007] Alexey Radul (2007). Report on the Probabilistic Language Scheme. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 2–10. Association for Computing Machinery, New York, NY. `http://hdl.handle.net/1721.1/39831`.

[Radul and Sussman, 2009a] Alexey Radul and Gerald Jay Sussman (2009a). The (Abridged) Art of the Propagator. In *Proceedings of the 2009 International Lisp Conference*, pages 41–56, Cambridge, MA. Association of Lisp Users, Sterling, Virginia, USA.

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

[Ramsey and Pfeffer, 2002] Norman Ramsey and Avi Pfeffer (2002). Stochastic Lambda Calculus and Monads of Probability Distributions. *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 154–165.

[Royce, 1970] Winston W. Royce (1970). Managing the Development of Large Software Systems. In *Proceedings, IEEE WESCON*, pages 1–9.

[Russell and Norvig, 1995] Stuart J. Russell and Peter Norvig (1995). *Artificial Intelligence: a Modern Approach*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

[Schulte and Stuckey, 2008] C. Schulte and P.J. Stuckey (2008). Efficient Constraint Propagation Engines. *Transactions on Programming Languages and Systems*, 31(1).

[Shivers, 1991] Olin Shivers (1991). *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA. CMU-CS-91-145.

[Shivers, 1999] Olin Shivers (1999). SRFI 1: List Library. Scheme Requests for Implementation. `http://srfi.schemers.org/srfi-1/srfi-1.html`.

[Siskind and McAllester, 1993] Jeffrey Mark Siskind and David Allen McAllester (1993). Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp. Technical report IRCS-93-03, University of Pennsylvania Institute for Research in Cognitive Science.

[Sitaram, 2004] Dorai Sitaram (2004). *Teach Yourself Scheme in Fixnum Days*. Online at `http://www.ccs.neu.edu/home/dorai/ty-scheme/ty-scheme.html`.

[Smolka, 1995] Gert Smolka (1995). The Oz Programming Model. *Lecture Notes in Computer Science*, 1000:324–343.

[Stallman and Sussman, 1977] Richard Matthew Stallman and Gerald Jay Sussman (1977). Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence*, 9:135–196.

[Steele Jr., 1980] Guy L. Steele Jr. (1980). The Definition and Implementation of a Computer Programming Language Based on Constraints. AI Memo 595, MIT Artificial Intelligence Laboratory, Cambridge, MA.

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

[Strachey, 1967] Christopher Strachey (1967). Fundamental Concepts in Programming Languages. Lecture notes for International Summer School in Computer Programming.

[Strachey, 2000] Christopher Strachey (2000). Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1):11–49.

[Stroustrup et al., 1991] Bjarne Stroustrup et al. (1991). *The C++ Programming Language*. Addison-Wesley Reading, MA.

[Sulzmann and Stuckey, 2007] M. Sulzmann and P.J. Stuckey (2007). HM (X) Type Inference is CLP (X) Solving. *Journal of Functional Programming*, 18(02):251–283.

[Sussman et al., 2001] Gerald Jay Sussman, Jack Wisdom, and M.E. Mayer (2001). *Structure and Interpretation of Classical Mechanics*. MIT Press, Cambridge, MA.

[Sutherland, 1963] Ivan E. Sutherland (1963). *SketchPad: A Man-Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA.

[Tack, 2009] Guido Tack (2009). *Constraint Propagation - Models, Techniques, Implementation*. PhD thesis, Saarland University, Germany.

[Van Hentenryck, 1989] Pascal Van Hentenryck (1989). *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA.

[Van Roy, 2005] Peter Van Roy (2005). *Multiparadigm Programming in Mozart/Oz*. Springer.

[Van Roy and Haridi, 2004] Peter Van Roy and Seif Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA.

[Waltz, 1972] David L. Waltz (1972). *Generating Semantic Description from Drawings of Scenes with Shadows*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA.

[Waltz, 1975] David L. Waltz (1975). Understanding Line Drawings of Scenes with Shadows. *The Psychology of Computer Vision*, pages 19–91.

[Wan and Hudak, 2000] Z. Wan and Paul Hudak (2000). Functional Reactive Programming from First Principles. *ACM SIGPLAN Notices*, 35(5):242–252.

[Zabih, 1987] Ramin Zabih (1987). Dependency-Directed Backtracking in Non-Deterministic Scheme. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

[Zabih, 1998] Ramin Zabih (1998). Dependency-Directed Backtracking in Non-Deterministic Scheme. AI Memo 956, MIT Artificial Intelligence Laboratory, Cambridge, MA.

[Zabih et al., 1987] Ramin Zabih, David Allen McAllester, and David Chapman (1987). Non-Deterministic Lisp with Dependency-Directed Backtracking. In *Proceedings of AAAI 87*, pages 59–64.