Storing Data in Control Flow

Russ Cox
July 11, 2023

research.swtch.com/pcdata

A decision that arises over and over when designing concurrent programs is whether to represent program state in control flow or as data. This post is about what that decision means and how to approach it. Done well, taking program state stored in data and storing it instead in control flow can make programs much clearer and more maintainable than they otherwise would be.

Before saying much more, it's important to note that concurrency is not parallelism.:

- Concurrency is about *how you write programs*, about being able to compose independently executing control flows, whether you call them processes or threads or goroutines, so that your program can be *dealing with* lots of things at once without turning into a giant mess.
- On the other hand, parallelism is about *how you execute programs*, allowing multiple computations to run simultaneously, so that your program can be *doing* lots of things at once efficiently.

Concurrency lends itself naturally to parallel execution, but the focus in this post is about how to use concurrency to write cleaner programs, not faster ones.

The difference between concurrent programs and non-concurrent programs is that concurrent programs can be written as if they are executing multiple independent control flows at the same time. The name for the smaller control flows varies by language: thread, task, process, fiber, coroutine, goroutine, and so on. No matter the name, the fundamental point for this post is that writing a program in terms of multiple independently executing control flows allows you to store program state in the execution state of one or more of those control flows, specifically in the program counter (which line is executing in that piece) and on the stack. Control flow state can always be maintained as explicit data instead, but then the explicit data form is essentially simulating the control flow. Most of the time, using the control flow features built into a programming language is easier to understand, reason about, and maintain than simulating them in data structures.

The rest of this post illustrates the rather abstract claims I've been making about storing data in control flow by walking through some concrete examples. They happen to be written in Go, but the ideas apply to any language that supports writing concurrent programs, including essentially every modern language.

A Step-by-Step Example

Here is a seemingly trivial problem that demonstrates what it means to store program state in control flow. Suppose we are reading characters from a file and want to scan over a C-style double-quoted string. In this case, we have a non-parallel program. There is no opportunity for parallelism here, but as we will see, concurrency can still play a useful part.

If we don't worry about checking the exact escape sequences in the string, it suffices to match the regular expression `"([^"\\]|\\.)*"`, which matches a double quote, then a sequence of zero or more characters, and then another double quote. Between the quotes, a character is anything that's not a quote or backslash, or else a backslash followed by anything (including a quote or backslash).

Every regular expression can be compiled into finite automaton or state machine, so we might use a tool to turn that specification into this Go code:

```
state := 0
for {
    c := read()
    switch state {
    case 0:
        if c != '"' {
            return false
        }
        state = 1
    case 1:
        if c == '"' {
            return true
        }
        if c == '\\' {
            state = 2
        } else {
            state = 1
        }
    case 2:
        state = 1
    }
}
```

The code has a single variable named `state` that represents the state of the automaton. The for loop reads a character and updates the state, over and over, until it finds either the end of the string or a syntax error. This is the kind of code that a program would write and that only a program could love. It's difficult for people to read, and it will be difficult for people to maintain.

The main reason this program is so opaque is that its program state is stored as data, specifically in the variable named `state`. When it's possible to store state in code instead, that often leads to a clearer program. To see this, let's transform the program, one small step at a time, into an equivalent but much more understandable version.

We can start by duplicating the `read` calls into each case of the switch:

```
state := 0
for {
    c := read()
    switch state {
    case 0:

        if c != ’"’ {
            return false
        }
        state = 1
    case 1:

        if c == ’"’ {
            return true
        }
        if c == ’\’ {
            state = 2
        } else {
            state = 1
        }
    case 2:

        state = 1
    }
}
```

```
state := 0
for {

    switch state {
    case 0:
        c := read()
        if c != ’"’ {
            return false
        }
        state = 1
    case 1:
        c := read()
        if c == ’"’ {
            return true
        }
       if c == ’\’ {
            state = 2
        } else {
            state = 1
        }
    case 2:
        c := read()
        state = 1
    }
}
```

(In this and all the displays that follow, the old program is on the left, the new program is on the right, and lines that haven’t changed are printed in gray text.)

Now, instead of writing to `state` and then immediately going around the for loop again to look up what to do in that state, we can use code labels and goto statements:

```
state := 0                      state0:
for {
    switch state {
    case 0:
        c := read()                 c := read()
        if c != ’"’ {               if c != ’"’ {
            return false                return false
        }                           }
        state = 1                   goto state1
    case 1:                     state1:
        c := read()                 c := read()
        if c == ’"’ {               if c == ’"’ {
            return true                 return true
        }                           }
        if c == ’\’ {              if c == ’\’ {
            state = 2                   goto state2
        } else {                    } else {
            state = 1                   goto state1
        }                           }
    case 2:                     state2:
        c := read()                 read()
        state = 1                   goto state1
    }
}
```

Then we can simplify the program further. The `goto state1` right before the `state1` label is a no-op and can be deleted. And we can see that there’s only one way to get to state2, so we might as well replace the `goto state2` with the actual code from state2:

```
state0:                         state0:
    c := read()                     c := read()
    if c != ’"’ {                   if c != ’"’ {
        return false                    return false
    }                               }
    goto state1
state1:                         state1:
    c := read()                     c := read()
    if c == ’"’ {                   if c == ’"’ {
        return true                     return true
    }                               }
    if c == ’\’ {                  if c == ’\’ {
        goto state2
    } else {
        goto state1
    }
state2:
    read()                              read()
    goto state1                         goto state1
                                    } else {
                                        goto state1
                                    }
```

Then we can factor the “goto state1” out of both branches of the if statement.

```
state0:                         state0:
    c := read()                     c := read()
    if c != ’"’ {                   if c != ’"’ {
        return false                    return false
    }                               }

state1:                         state1:
    c := read()                     c := read()
    if c == ’"’ {                   if c == ’"’ {
        return true                     return true
    }                               }
    if c == ’\’ {                  if c == ’\’ {
        read()                          read()
        goto state1                 }
    } else {                        goto state1
        goto state1
    }
```

Then we can drop the unused `state0` label and replace the `state1` loop with an actual loop. Now we have something that looks like a real program:

```
state0:
    c := read()                 c := read()
    if c != ’"’ {               if c != ’"’ {
        return false                return false
    }                           }

state1:                         for {
    c := read()                     c := read()
    if c == ’"’ {                   if c == ’"’ {
        return true                     return true
    }                               }
    if c == ’\’ {                  if c == ’\’ {
        read()                          read()
    }                               }
    goto state1                 }
```

We can simplify a little further, eliminating some unnecessary variables, and we can make the check for the final quote (`c == ""`) be the loop terminator.

```
c := read()                     if read() != ’"’ {
if c != ’"’ {
    return false                    return false
}                               }

for {                           var c byte
    c := read()                 for c != ’"’ {
    if c == ’"’ {                   c = read()
        return true
    }
    if c == ’\’ {                  if c == ’\’ {
        read()                          read()
    }                               }
}                               }
                                return true
```

The final version is:

```
func parseQuoted(read func() byte) bool {
    if read() != '"' {
        return false
    }
    var c byte
    for c != '"' {
        c = read()
        if c == '\\' {
            read()
        }
    }
    return true
}
```

Earlier I explained the regular expression by saying it “matches a double quote, then a sequence of zero or more characters, and then another double quote. Between the quotes, a character is anything that’s not a quote or backslash, or else a backslash followed by anything.” It’s easy to see that this program does exactly that.

Hand-written programs can have opportunities to use control flow too. For example, here is a version that a person might have written by hand:

```
if read() != '"' {
    return false
}
inEscape := false
for {
    c := read()
    if inEscape {
        inEscape = false
        continue
    }
    if c == '"' {
        return true
    }
    if c == '\\' {
        inEscape = true
    }
}
```

The same kinds of small steps can be used to convert the boolean variable `inEscape` from data to control flow, ending at the same cleaned up version.

Either way, the `state` variable in the original is now implicitly represented by the program counter, meaning which part of the program is executing. The comments in this version indicate the implicit value of the original's `state` (or `inEscape`) variables:

```
func parseQuoted(read func() byte) bool {
    // state == 0
    if read() != '"' {
        return false
    }

    var c byte
    for c != '"' {
        // state == 1 (inEscape = false)
        c = read()
        if c == '\\' {
            // state == 2 (inEscape = true)
            read()
        }
    }
    return true
}
```

The original program was, in essence, *simulating* this control flow using the explicit `state` variable as a program counter, tracking which line was executing. If a program can be converted to store explicit state in control flow instead, then that explicit state was merely an awkward simulation of the control flow.

More Threads for More State

Before widespread support for concurrency, that kind of awkward simulation was often necessary, because a different part of the program wanted to use the control flow instead.

For example, suppose the text being parsed is the result of decoding base64 input, in which sequences of four 6-bit characters (drawn from a 64-character alphabet) decode to three 8-bit bytes. The core of that decoder looks like:

```
for {
    c1, c2, c3, c4 := read(), read(), read(), read()
    b1, b2, b3 := decode(c1, c2, c3, c4)
    write(b1)
    write(b2)
    write(b3)
}
```

If we want those `write` calls to feed into the parser from the previous section, we need a parser that can be called with one byte at a time, not one that demands a `read` callback. This decode loop cannot be presented as a `read` callback because it obtains 3 input bytes at a time and uses its control flow to track which ones have been written. Because the decoder is storing its own state in its control flow, `parseQuoted` cannot.

In a non-concurrent program, this base64 decoder and `parseQuoted` would be at an impasse: one would have to give up its use of control flow state and fall back to some kind of simulated version instead.

To rewrite parseQuoted, we have to reintroduce the state variable, which we can encapsulate in a struct with a Write method:

```
type parser struct {
    state int
}

func (p *parser) Init() {
    p.state = 0
}

func (p *parser) Write(c byte) Status {
    switch p.state {
    case 0:
        if c != '"' {
            return BadInput
        }
        p.state = 1
    case 1:
        if c == '"' {
            return Success
        }
        if c == '\\' {
            p.state = 2
        } else {
            p.state = 1
        }
    case 2:
        p.state = 1
    }
    return NeedMoreInput
}
```

The Init method initializes the state, and then each Write loads the state, takes actions based on the state and the input byte, and then saves the state back to the struct.

For parseQuoted, the state machine is simple enough that this may be completely fine. But maybe the state machine is much more complex, or maybe the algorithm is best expressed recursively. In those cases, being passed an input sequence by the caller one byte at a time means making all that state explicit in a data structure simulating the original control flow.

Concurrency eliminates the contention between different parts of the program over which gets to store state in control flow, because now there can be multiple control flows.

Suppose we already have the parseQuoted function, and it's big and complicated and tested and correct, and we don't want to change it. We can avoid editing that code at all by writing this wrapper:

```
type parser struct {
    c      chan byte
    status chan Status
}

func (p *parser) Init() {
    p.c = make(chan byte)
    p.status = make(chan Status)
    go p.run()
    <-p.status // always NeedMoreInput
}

func (p *parser) run() {
    if !parseQuoted(p.read) {
        p.status <- BadSyntax
    } else {
        p.status <- Success
    }
}

func (p *parser) read() byte {
    p.status <- NeedMoreInput
    return <-p.c
}

func (p *parser) Write(c byte) Status {
    p.c <- c
    return <-p.status
}
```

Note the use of parseQuoted, completely unmodified, in the run method. Now the base64 decoder can use p.Write and keep its program counter and local variables.

The new goroutine that Init creates runs the p.run method, which invokes the original parseQuoted function with an appropriate implementation of read. Before starting p.run, Init allocates two channels for communicating between the p.run method, runing in its own goroutine, and whatever goroutine calls p.Write (such as the base64 decoder's goroutine). The channel p.c carries bytes from Write to read, and the channel p.status carries status updates back. Each time parseQuoted calls read, p.read sends NeedMoreInput on p.status and waits for an input byte on p.c. Each time p.Write is called, it does the opposite: it sends the input byte c on p.c and then waits for and returns an updated status from p.status. These two calls take turns, back and forth, one executing and one waiting at any given moment.

To get this cycle going, the Init method does the initial receive from p.status, which will correspond to the first read in parseQuoted. The actual status for that first update is guaranteed to be NeedMoreInput and is discarded. To end the cycle, we assume that when Write returns BadSyntax or Success, the caller knows not to call Write again. If the caller incorrectly kept calling Write, the send on p.c would block forever, since parseQuoted is done. We would of course make that more robust in a production implementation.

By creating a new control flow (a new goroutine), we were able to keep the code-state-based implementation of `parseQuoted` as well as our code-state-based base64 decoder. We avoided having to understand the internals of either implementation. In this example, both are trivial enough that rewriting one would not have been a big deal, but in a larger program, it could be a huge win to be able to write this kind of adapter instead of having to make changes to existing code. As we'll discuss later, the conversion is not entirely free – we need to make sure the extra control flow gets cleaned up, and we need to think about the cost of the context switches – but it may well still be a net win.

Store Stacks on the Stack

The base64 decoder's control flow state included not just the program counter but also two local variables. Those would have to be pulled out into a struct if the decoder had to be changed not to use control flow state. Programs can use an arbitrary number of local variables by using their call stack. For example, suppose we have a simple binary tree data structure:

```
type Tree[V any] struct {
    left  *Tree[V]
    right *Tree[V]
    value V
}
```

If you can't use control flow state, then to implement iteration over this tree, you have to introduce an explicit "iterator":

```
type Iter[V any] struct {
    stk []*Tree[V]
}

func (t *Tree[V]) NewIter() *Iter[V] {
    it := new(Iter[V])
    for ; t != nil; t = t.left {
        it.stk = append(it.stk, t)
    }
    return it
}

func (it *Iter[V]) Next() (v V, ok bool) {
    if len(it.stk) == 0 {
        return v, false
    }
    t := it.stk[len(it.stk)-1]
    v = t.value
    it.stk = it.stk[:len(it.stk)-1]
    for t = t.right; t != nil; t = t.left {
        it.stk = append(it.stk, t)
    }
    return v, true
}
```

On the other hand, if you can use control flow state, confident that other parts of the program that need their own state can run in other control flows, then you can implement iteration without an explicit iterator, as a method that calls a yield function for each value:

```
func (t *Tree[V]) All(f func(v V)) {
    if t != nil {
        t.left.All(f)
        f(t.value)
        t.right.All(f)
    }
}
```

The `All` method is obviously correct. The correctness of the `Iter` version is much less obvious. The simplest explanation is that `Iter` is simulating `All`. The `NewIter` method's loop that sets up `stk` is simulating the recursion in `t.All(f)` down successive `t.left` branches. `Next` pops and saves the `t` at the top of the stack and then simulates the recursion in `t.right.All(f)` down successive `t.left` branches, setting up for the next `Next`. Finally it returns the value from the top-of-stack `t`, simulating `f(value)`.

We could write code like `NewIter` and argue its correctness by explaining that it simulates a simple function like `All`. I'd rather write `All` and stop there.

Comparing Binary Trees

One might argue that `NewIter` is better than `All`, because it does not use any control flow state, so it can be used in contexts that already use their control flows to hold other information. For example, what if we want to traverse two binary trees at the same time, checking that they hold the same values even if their internal structure differs. With `NewIter`, this is straighforward:

```
func SameValues[V any](t1, t2 *Tree[V]) bool {
    it1 := t1.NewIter()
    it2 := t2.NewIter()
    for {
        v1, ok1 := it1.Next()
        v2, ok2 := it2.Next()
        if v1 != v2 || ok1 != ok2 {
            return false
        }
        if !ok1 && !ok2 {
            return true
        }
    }
}
```

This program cannot be written as easily using `All`, the argument goes, because `SameValues` wants to use its own control flow (advancing two lists in lockstep) that cannot be replaced by `All`'s control flow (recursion over the tree). But this is a false dichotomy, the same one we saw with `parseQuoted` and the base64 decoder. If two different functions have different demands on control flow state, they can run in different control flows.

In our case, we can write this instead:

```
func SameValues[V any](t1, t2 *Tree[V]) bool {
    c1 := make(chan V)
    c2 := make(chan V)
    go gopher(c1, t1.All)
    go gopher(c2, t2.All)
    for {
        v1, ok1 := <-c1
        v2, ok2 := <-c2
        if v1 != v2 || ok1 != ok2 {
            return false
        }
        if !ok1 && !ok2 {
            return true
        }
    }
}

func gopher[V any](c chan<- V, all func(func(V))) {
    all(func(v V) { c <- v })
    close(c)
}
```

The function `gopher` uses `all` to walk a tree, announcing each value into a channel. After the walk, it closes the channel.

`SameValues` starts two concurrent gophers, each of which walks one tree and announces the values into one channel. Then `SameValues` does exactly the same loop as before to compare the two value streams.

Note that `gopher` is not specific to binary trees in any way: it applies to *any* iteration function. That is, the general idea of starting a goroutine to run the `All` method works for converting any code-state-based iteration into an incremental iterator. My next post, "Coroutines for Go," expands on this idea.

Limitations

This approach of storing data in control flow is not a panacea. Here are a few caveats:

- If the state needs to evolve in ways that don't naturally map to control flow, then it's usually best to leave the state as data. For example, the state maintained by a node in a distributed system is usually not best represented in control flow, because timeouts, errors, and other unexpected events tend to require adjusting the state in unpredictable ways.
- If the state needs to be serialized for operations like snapshots, or sending over a network, that's usually easier with data than code.
- When you do need to create multiple control flows to hold different control flow state, the helper control flows need to be shut down. When `SameValues` returns false, it leaves the two concurrent `gophers` blocked waiting to send their next values. Instead, it should unblock them. That requires communication in the other direction to tell `gopher` to stop early. "Coroutines for Go" shows that.
- In the multiple thread case, the switching costs can be significant. On my laptop, a C thread switch takes a few microseconds. A channel operation and goroutine switch is an order of magnitude cheaper: a cou-

ple hundred nanoseconds. An optimized coroutine system can reduce the cost to tens of nanoseconds or less.

In general, storing data in control flow is a valuable tool for writing clean, simple, maintainable programs. Like all tools, it works very well for some jobs and not as well for others.

Counterpoint: John McCarthy's GOPHER

The idea of using concurrency to align a pair of binary trees is over 50 years old. It first appeared in Charles Prenner's "The control structure facilities of ECL" (*ACM SIGPLAN Notices*, Volume 6, Issue 12, December 1971; see pages 106–109). In that presentation, titled "Tree Walks Using Coroutines", the problem was to take two binary trees A and B with the same number of nodes and copy the value sequence from A into B despite the two having different internal structure. They present a straightforward coroutine-based variant.

Brian Smith and Carl Hewitt introduced the problem of simply comparing two Lisp-style cons trees (in which internal nodes carry no values) in their draft of "A Plasma Primer" (March 1975; see pages 61-62). For that problem, which they named "samefringe", they used continuation-based actors to run a pair of "fringe" actors (credited to Howie Shrobe) over the two trees and report nodes back to a comparison loop.

Gerald Sussman and Guy Steele presented the samefringe problem again, in "Scheme: An Interpreter for Extended Lambda Calculus" (December 1975; see pages 8–9), with roughly equivalent code (crediting Smith, Hewitt, and Shrobe for inspiration). They refer to it as a "classic problem difficult to solve in most programming languages".

In August 1976, *ACM SIGART Bulletin* published Patrick Greussay's "An Iterative Lisp Solution to the Samefringe Problem", This prompted a response letter by Tim Finin and Paul Rutler in the November 1976 issue (see pages 4–5) pointing out that Greussay's solution runs in quadratic time and memory but also remarking that "the SAMEFRINGE problem has been notoriously overused as a justification for coroutines." That discussion prompted a response letter by John McCarthy in the February 1977 issue (see page 4).

In his response, titled "Another samefringe", McCarthy gives the following LISP solution:

```
(DE SAMEFRINGE (X Y)
       (OR (EQ X Y)
           (AND (NOT (ATOM X))
                (NOT (ATOM Y))
                (SAME (GOPHER X) (GOPHER Y)))))

(DE SAME (X Y)
       (AND (EQ (CAR X) (CAR Y))
            (SAMEFRINGE (CDR X) (CDR Y))))

(DE GOPHER (U)
       (COND ((ATOM (CAR U)) U)
             (T (GOPHER (CONS (CAAR U)
                              (CONS (CDAR U) (CDR U)))))))
```

He then explains:

> *gopher* digs up the first atom in an S-expression, piling up the *cdr* parts (with its hind legs) so that indexing through the atoms can be resumed. Because of shared structure, the number of new cells in use in each ar-

> gument at any time (apart from those occupied by the original expression and assuming iterative execution) is the number of *cars* required to go from the top to the current atom – usually a small fraction of the size of the S-expression.

In modern terms, McCarthy's `GOPHER` loops applying right tree rotations until the leftmost node is at the top of the tree. `SAMEFRINGE` applies `GOPHER` to the two trees, compares the tops, and then loops to consider the remainders.

After presenting a second, more elaborate solution, McCarthy remarks:

> I think all this shows that *samefringe* is not an example of the need for co-routines, and a new "simplest example" should be found. There is no merit in merely moving information from data structure to control structure, and it makes some kinds of modification harder.

I disagree with "no merit". We can view McCarthy's `GOPHER`-ized trees as an encoding of the same stack that `NewIter` maintains but in tree form. The correctness follows for the same reasons: it is simulating a simple recursive traversal. This `GOPHER` is clever, but it only works on trees. If you're not John McCarthy, it's easier to write the recursive traversal and then rely on the general, concurrency-based `gopher` we saw earlier to do the rest.

My experience is that when it is possible, moving information from data structure to control structure usually makes programs clearer, easier to understand, and easier to maintain. I hope you find similar results.