Sequential control

Thus far, the code we’ve written has had very unusual semantics. Rather than specifying a specific action (i.e. a specific set of voltages or speeds to send to the motors), our code has specified policies – mappings from states of the world (or sensor readings thereof) to actions. We’ve specified these policies as parallel circuits rather than as sequences of imperative commands and we’ve built compound policies by combining simpler circuits in parallel, perhaps with some sort of arbitration mechanism. This is in marked contrast to standard sequential programming languages in which the fundamental units are actions or “imperatives,” such as assigning a value to a variable or calling a procedure, and the fundamental ways of combining them involve sequencing constructions such as \texttt{begin} ... \texttt{end} pairs, loops, etc.

The advantage of parallel programming is that you don’t have to worry about what the program is “paying attention to.” In a sequential program, the imperative $x = y + z$ states that $x$ should be \textit{temporarily} made equal to $y + z$. The semantics of imperative programs guarantee only that $x$ holds the sum of $y$ and $z$ only immediately after the execution of the command. All bets are off once any of those variables change. To maintain the relationship, you need to ensure that the program explicitly recomputes $x$ any time $y$ or $z$ are changed. Missing a case in which $x$ needs to be recomputed could lead to a collision or other failure. On the other hand, the circuit specification \texttt{(define-signal $x$ (+ $y$ $z$))} states that $x$ will \textit{always} be equal to $y + z$, even when $y$ and $z$ change, eliminating the need for explicit updates.

Nevertheless, there are still times when you really want to be able to write a robot program that says “do this, then do that.” For example, you may want to program the robot to go down the corridor, take the first left, then go to the end of the hall. Most robot control programs involve a mixture of this kind of sequential control with some kind of behavior-like control.

Discrete actions

Sequential controllers have a radically different view of the world. It’s not just a matter of whether the behaviors run in parallel or serially. Sequential control requires that the very nature of behaviors be changed. Thus far, we’ve always taken behaviors to be control policies that know when to activate themselves; but in sequential control, a separate component of the system – the sequential controller – turns behaviors on and off. Moreover, the behaviors we’ve written thus far are \textit{never done}, only \textit{inactive}; but a sequential controller can’t activate a behavior until the previous behavior in the sequence has finished – whatever finished might mean.

So we need to introduce a new kind of object into the system that can (1) be remotely activated and deactivated and (2) be asked whether it’s done. We’ll call these objects \texttt{actions}. We can make an action out of a behavior simply by:

- Replacing its activation level with a \texttt{register} (i.e. a memory location) that other components can turn on and off.
- Allowing the other components to read the register to see if it’s done yet.
- Providing it with a \texttt{termination condition} that specifies when it should switch itself off.

We’ll call these action-behaviors \texttt{primitive actions} or just “actions” for short. They’re actions that compute motor vectors suitable for controlling the robot. Sequential controllers are themselves a kind of
action, in this case, a **compound action** that does its work by turning other actions (primitive or compound) on and off.

In GRL, you can create primitive actions with the `define-action` macro. In its simplest form, you just give a motor vector and a termination condition:

```
(define-action (name)
    (terminate-when boolean-signal)
    motor-vector-signal)
```

This creates a new behavior called `name` that you can use as you would any other behavior – you can pass it to `drive-base`, compose it with other behaviors in a `behavior-or`, etc.

**Plans**

We will call the compound actions implemented by sequential controllers **plans**. You can make a plan using the `define-plan` macro by specifying a name and a body for the plan. The body can pretty much look like ordinary sequential Scheme code, except that the “procedures” you call have to be actions (i.e. primitive actions or other plans):

```
(define-plan (run-program)
    (follow-corridor-to-first-left)
    (turn-left)
    (follow-corridor-to-end))
```

This will produce a sequential controller that activates `follow-corridor-to-first-left`, then waits until it self-terminates, then activates `turn-left`, etc. It can be called itself in another plan, in which case that other plan will wait until `run-program` has self-terminated, i.e. until its gotten all the way through `follow-corridor-to-end`. Of course, you need some way to get the first plan (the equivalent of the C/C++ `main()` routine) started. You can do this with the `initially-running?` option:

---

1 The word “plan” has a long history in AI and cognitive science. The term was introduced in the 50s by Miller, Galanter, and Pribram, to mean a kind of controller (sequential or otherwise) that tried to achieve some definite end using some definite strategy analogous to a computer program. Plans were presented as an alternative to radical behaviorism’s model of behavior as being controlled purely by stimulus-response rules learned through reinforcement; they provided a more structured, and more “cognitive,” model of behavior, without having to resort to the seemingly occult notions of psychic energy, consciousness, etc. found, for example, in Freud’s theories. One obvious question was where the plans came from – were they innate, learned, or “deduced” by some kind of general reasoning system.

When AI came on the scene in the 60s, with its emphasis on computational models of deductive reasoning, it was natural for researchers to examine the problem of constructing plans deductively from goals using automated reasoning techniques. This became known as the **planning problem**. Most planning research has focused on deducing sequential controllers, although there is also a large literature on deducing policies for Markov decision processes, on computing “universal plans”, which are more or less like policies, and on intermediate points between sequences and policies. Nevertheless, the term “plan” is most often used to mean “prespecified action sequence.” Although the term “sequential controller” is more precise, we will follow the common usage since `define-sequential-controller` is needlessly verbose.
Of course, some behaviors, such as following corridors, just don’t fit nicely into the self-terminating action model. In order to fit corridor following into our ontology, we’ve had to break it up into the separate actions of `follow-corridor-to-first-left`, `follow-corridor-to-end`, etc. This is ugly and a pain. Fortunately, you can separate the acts of starting and stopping an action from waiting for a condition. By saying:

```scheme
(define-plan (run-program)
  (start (follow-corridor))
  (wait-until left-turn?)
  (stop follow-corridor)
  (turn-left)
  (start (follow-corridor))
  (wait-until end-of-corridor?)
  (stop follow-corridor))
```

we can collapse the different flavors of corridor following into a single action.

**Passing arguments**

At this point, you’re probably confused as to why `run-program` has parentheses around it and why `follow-corridor` sometimes has parentheses and sometimes doesn’t. The answer is that you can pass arguments to an action. For example, instead of having `turn-left` and `turn-right`, you could have a single `turn` action that takes the number of degrees as an argument:

```scheme
(define-action (turn (degrees 0))
  (terminate-when ...) ...)
```

```scheme
(define-plan (run-program)
  (start (follow-corridor))
  (wait-until left-turn?)
  (stop follow-corridor)
  (turn 90)
  (start (follow-corridor))
  (wait-until end-of-corridor?)
  (stop follow-corridor))
```

This says that `turn` is a primitive action that takes an argument, `degrees`, whose initial value is 0. The initial value is there partly because you can make actions that are initially running or that self-activate themselves (like normal behaviors) and partly so that the compiler knows what data type the argument is.

**Assignment statements and control structures**

You may be asking why it is that we don’t pass the termination condition (i.e. `left-turn?` versus `end-of-corridor?`) to `follow-corridor` as an argument. The answer is that arguments act like normal Scheme or C variables — they hold their values until they’re changed with `set!`; the call to and action sets its arguments once. The action can then modify those arguments like normal variables if it wants to. This
is useful in some ways, but it does mean that a plan can’t pass a time-varying signal as an argument to an action.\textsuperscript{2}

In general, you can write fairly normal-looking Scheme code for the body of a plan. For example, you can say:

\begin{verbatim}
(define-plan (do-laps)
  (let ((lap-count 0))
    (while #t
      (set! lap-count (+ lap-count 1))
      (format #t "Starting lap ~D~%" lap-count)
      (follow-corridor-to-left-turn)
      (turn-left)
      (follow-corridor-to-left-turn)
      (turn-left)
      (follow-corridor-to-left-turn)
      (turn-left)
      (follow-corridor-to-left-turn)
      (turn-left)))

\end{verbatim}

which would do counter-clockwise laps around a building, each time printing out the lap number on the screen.\textsuperscript{3}

One thing you can’t do in plans is recursive calls. It’s straightforward to do in principle, but the GRL compiler doesn’t support it for reasons that will become clear later.

\textsuperscript{2} We may modify the language to allow this some day, but for the moment, you can’t do it.

\textsuperscript{3} \texttt{Format} is the equivalent of \texttt{printf}; \texttt{~D} is like \texttt{%d} in C, and \texttt{~%} is like \texttt{\n}. The \texttt{#t} argument means that it should print to the screen.