• Nondeterministic evaluator

To load the nondeterministic metacircular evaluator, say

(load "~cs61a/lib/ambeval.scm")

Solution spaces, streams, and backtracking

Many problems are of the form “Find all A such that B” or “find an A such that B.” For example: Find an even integer that is not the sum of two primes; find a set of integers $a, b, c$, and $n$ such that $a^n + b^n = c^n$ and $n > 2$. (These problems might not be about numbers: Find all the states in the United States whose first and last letters are the same.)

In each case, the set $A$ (even integers, sets of four integers, or states) is called the solution space. The condition $B$ is a predicate function of a potential solution that’s true for actual solutions.

One approach to solving problems of this sort is to represent the solution space as a stream, and use stream-filter to select the elements that satisfy the predicate:

(stream-filter sum-of-two-primes? even-integers)

(stream-filter Fermat? (pairs (pairs integers integers)
          (pairs integers integers)))

(stream-filter (lambda (x) (equal? (first x) (last x))) states)

The stream technique is particularly elegant for infinite problem spaces, because the program seems to be generating the entire solution space $A$ before checking the predicate $B$. (Of course we know that really the steps of the computation are reordered so that the elements are tested as they are generated.)

This week we consider a different way to express the same sort of computation, a way that makes the sequence of events in time more visible. In effect we’ll say:

• Pick a possible solution.
• See if it’s really a solution.
• If so, return it; if not, try another.

Here’s an example of the notation:

> (let ((a (amb 2 3 4))
          (b (amb 6 7 8)))
    (require (= (remainder b a) 0))
    (list a b))

(2 6)
> try-again
(2 8)
> try-again
(3 6)
> try-again
(4 8)
> try-again
There are no more solutions.

208
The main new thing here is the special form \texttt{amb}. This is not part of ordinary Scheme! We are adding it as a new feature in the metacircular evaluator. \texttt{Amb} takes any number of argument expressions and returns the value of one of them. You can think about this using either of two metaphors:

- The computer clones itself into as many copies as there are arguments; each clone gets a different value.
- The computer magically knows which argument will give rise to a solution to your problem and chooses that one.

What really happens is that the evaluator chooses the first argument and returns its value, but if the computation later \texttt{fails} then it tries again with the second argument, and so on until there are no more to try. This introduces another new idea: the possibility of the failure of a computation. That’s not the same thing as an error! Errors (such as taking the \texttt{car} of an empty list) are handled the same in this evaluator as in ordinary Scheme; they result in an error message and the computation stops. A failure is different; it’s what happens when you call \texttt{amb} with no arguments, or when all the arguments you gave have been tried and there are no more left.

In the example above I used \texttt{require} to cause a failure of the computation if the condition is not met. \texttt{Require} is a simple procedure in the metacircular Scheme-with-\texttt{amb}:

\begin{verbatim}
(define (require condition)
  (if (not condition) (amb)))
\end{verbatim}

So here’s the sequence of events in the computation above:

\begin{verbatim}
a=2
  b=6; 6 is a multiple of 2, so return (2 6)
  \[try-again\]
    b=7; 7 isn’t a multiple of 2, so fail.
    b=8; 8 is a multiple of 2, so return (2 8)
  \[try-again\]
    No more values for b, so fail.

a=3
  b=6; 6 is a multiple of 3, so return (3 6)
  \[try-again\]
    b=7; 7 isn’t a multiple of 3, so fail.
    b=8; 8 isn’t a multiple of 3, so fail.
    No more values for b, so fail.

a=4
  b=6; 6 isn’t a multiple of 4, so fail.
  b=7; 7 isn’t a multiple of 4, so fail.
  b=8; 8 is a multiple of 4, so return (4 8)
  \[try-again\]
    No more values for b, so fail.
No more values for a, so fail.
(No more pending AMBs, so report failure to user.)
\end{verbatim}
Recursive Amb

Since amb accepts any argument expressions, not just literal values as in the example above, it can be used recursively:

```
(define (an-integer-between from to)
  (if (> from to)
    (amb)
    (amb from (an-integer-between (+ from 1) to))))
```

or if you prefer:

```
(define (an-integer-between from to)
  (require (>= to from))
  (amb from (an-integer-between (+ from 1) to)))
```

Further, since amb is a special form and only evaluates one argument at a time, it has the same delaying effect as cons-stream and can be used to make infinite solution spaces:

```
(define (integers-from from)
  (amb from (integers-from (+ from 1))))
```

This integers-from computation never fails—there is always another integer—and so it won’t work to say

```
(let ((a (integers-from 1))
     (b (integers-from 1)))
  ...)
```

because a will never have any value other than 1, because the second amb never fails. This is analogous to the problem of trying to append infinite streams; in that case we could solve the problem with interleave but it’s harder here.

Footnote on order of evaluation

In describing the sequence of events in these examples, I’m assuming that Scheme will evaluate the arguments of the unnamed procedure created by a let from left to right. If I wanted to be sure of that, I should use let* instead of let. But it matters only in my description of the sequence of events; considered abstractly, the program will behave correctly regardless of the order of evaluation, because all possible solutions will eventually be tried—although maybe not in the order shown here.

Success or failure

In the implementation of amb, the most difficult change to the evaluator is that any computation may either succeed or fail. The most obvious way to try to represent this situation is to have eval return some special value, let’s say the symbol =failed=, if a computation fails. (This is analogous to the use of =no-value= in the Logo interpreter project.) The trouble is that if an amb fails, we don’t want to continue the computation; we want to “back up” to an earlier stage in the computation. Suppose we are trying to evaluate an expression such as

```
(a (b (c (d 4))))
```

and suppose that procedures b and c use amb. Procedure d is actually invoked first; then c is invoked with the value d returned as argument. The amb inside procedure c returns its first argument, and c uses that to compute a return value that becomes the argument to b. Now suppose that the amb inside b fails. We don’t want to invoke a with the value =failed= as its argument! In fact we don’t want to invoke a at all; we want to re-evaluate the body of c but using the second argument to its amb.

A&S take a different approach. If an amb fails, they want to be able to jump right back to the previous amb, without having to propagate the failure explicitly through several intervening calls to eval. To make this
work, intuitively, we have to give eval two different places to return to when it’s finished, one for a success and the other for a failure.

**Continuations**

Ordinarily a procedure doesn’t think explicitly about where to return; it returns to its caller, but Scheme takes care of that automatically. For example, when we compute

\[(\ast 3 (\text{square} 5))\]

the procedure square computes the value 25 and Scheme automatically returns that value to the eval invocation that’s waiting to use it as an argument to the multiplication. But we could tell square explicitly, “when you’ve figured out the answer, pass it on to be multiplied by 3” this way:

\[
\begin{align*}
\text{(define (square x continuation)} \\
\text{ (continuation (* x x)))}
\end{align*}
\]

\[
\begin{align*}
> (\text{square 5 (lambda (y) (* y 3))})
\end{align*}
\]

75

A *continuation* is a procedure that takes your result as argument and says what’s left to be done in the computation.

**Continuations for success and failure**

In the case of the nondeterministic evaluator, we give eval *two* continuations, one for success and one for failure. Note that these continuations are part of the implementation of the evaluator; the user of amb doesn’t deal explicitly with continuations.

Here’s a handwavy example. In the case of

\[(a (b (c (d 4))))\]

procedure b’s success continuation is something like

\[
\begin{align*}
\text{(lambda (value) (a value))}
\end{align*}
\]

but its failure continuation is

\[
\begin{align*}
\text{(lambda () (a (b (redo-amb-in-c)))))}
\end{align*}
\]

This example is handwavy because these “continuations” are from the point of view of the user of the metacircular Scheme, who doesn’t know anything about continuations, really. The true continuations are written in underlying Scheme, as part of the evaluator itself.

If a computation fails, the most recent amb wants to try another value. So a continuation failure will redo the amb with one fewer argument. There’s no information that the failing computation needs to send back to that amb except for the fact of failure itself, so the failure continuation procedure needs no arguments.

On the other hand, if the computation succeeds, we have to carry out the success continuation, and that continuation needs to know the value that we computed. It also needs to know what to do if the continuation itself fails; most of the time, this will be the same as the failure continuation we were given, but it might not be. So a success continuation must be a procedure that takes two arguments: a value and a failure continuation.

The book bases the nondeterministic evaluator on the analyzing one, but I’ll use a simplified version based on plain old eval (it’s in cs61a/lib/vambeval.scm).

Most kinds of evaluation always succeed, so they invoke their success continuation and pass on the failure one. I’ll start with a too-simplified version of eval-if in this form:
(define (eval-if exp env succeed fail) ; WRONG!
  (if (eval (if-predicate exp) env succeed fail)
      (eval (if-consequent exp) env succeed fail)
      (eval (if-alternative exp) env succeed fail)))

The trouble is, what if the evaluation of the predicate fails? We don’t then want to evaluate the consequent or the alternative. So instead, we just evaluate the predicate, giving it a success continuation that will evaluate the consequent or the alternative, supposing that evaluating the predicate succeeds.

In general, wherever the ordinary metacircular evaluator would say

(define (eval-foo exp env)
  (eval step-1 env)
  (eval step-2 env))

using eval twice for part of its work, this version has to eval the first part with a continuation that evals the second part:

(define (eval-foo exp env succeed fail)
  (eval step-1 env
   (lambda (value-1 fail-1)
     (eval step-2 env succeed fail-1))
   fail))

(In either case, step-2 presumably uses the result of evaluating step-1 somehow.)

Here’s how that works out for if:

(define (eval-if exp env succeed fail)
  (eval (if-predicate exp) ; test the predicate
      env
      (lambda (pred-value fail2) ; with this success continuation
        (if (true? pred-value)
          (eval (if-consequent exp) env succeed fail2)
          (eval (if-alternative exp) env succeed fail2)))
      fail)) ; and the same failure continuation

What’s fail2? It’s the failure continuation that the evaluation of the predicate will supply. Most of the time, that’ll be the same as our own failure continuation, just as eval-if uses fail as the failure continuation to pass on to the evaluation of the predicate. But if the predicate involves an amb expression, it will generate a new failure continuation. Think about an example like this one:

> (if (amb #t #f)
   (amb 1)
   (amb 2))
1

> try-again
2

(A more realistic example would have the predicate expression be some more complicated procedure call that had an amb in its body.) The first thing that happens is that the first amb returns #t, and so if evaluates its second argument, and that second amb returns 1. When the user says to try again, there are no more values for that amb to return, so it fails. What we must do is re-evaluate the first amb, but this time returning its second argument, #f. By now you’ve forgotten that we’re trying to work out what fail2 is for in eval-if, but this example shows why the failure continuation when we evaluate if-consequent (namely the (amb 1) expression) has to be different from the failure continuation for the entire if expression. If the entire if
fails (which will happen if we say try-again again) then its failure continuation will tell us that there are no more values. That continuation is bound to the name fail in eval-if. What ends up bound to the name fail2 is the continuation that re-evaluates the predicate amb.

How does fail2 get that binding? When eval-if evaluates the predicate, which turns out to be an amb expression, eval-amb will evaluate whatever argument it’s up to, but with a new failure continuation:

```
(define (eval-amb exp env succeed fail)
  (if (null? (cdr exp)) ; (car exp) is the word AMB
      (fail) ; no more args, call failure cont.
      (eval (cadr exp) ; Otherwise evaluate the first arg
        env
        succeed ; with my same success continuation
        (lambda () ; but with a new failure continuation:
          (eval-amb (cons 'amb (cddr exp)) ; try the next argument
            env
            succeed
            fail))))
```

Notice that eval-if, like most other cases, provides a new success continuation but passes on the same failure continuation that it was given as an argument. But eval-amb does the opposite: It passes on the same success continuation it was given, but provides a new failure continuation.

Of course there are a gazillion more details, but the book explains them, once you understand what a continuation is. The most important of these complications is that anything involving mutation is problematic. If we say

```
(define x 5)
(set! x (+ x (amb 2 3)))
```

it’s clear that the first time around x should end up with the value 7 (5 + 2). But if we try again, we’d like x to get the value 8 (5 + 3), not 10 (7 + 3). So set! must set up a failure continuation that undoes the change in the binding of x, restoring its original value of 5, before letting the amb provide its second argument.

Note: The second part of programming project 4 is this week.