TUTORIAL

Using an Abstracted Interpreter to Understand Abstract Interpretation

Daniel P. Friedman and Anurag Mendhekar
Computer Science Department, Indiana University
Bloomington, IN 47405, USA

1 Introduction

The goal of abstract interpretation is to allow the user to do program analysis with a set of values that abstract another set of values. We present an example of abstract interpretation from real world experience. Suppose that we wish to travel from Nice to Paris. We are not sure how long the trip will take. Naturally, there is a very simple way to find out. Check the time just before we start our trip and check it again when we arrive and calculate the time based on these two numbers. That works, but we probably wanted to know how long it would be without actually taking the trip. What we needed was an abstract characterization of the road from Nice to Paris. Such things exist. They are called road maps. So, had we looked at such a map and used its scale, we could calculate the distance and by guessing our average speed, we likely would determine a reasonable answer to our question without actually taking the trip.

We can think about the actual taking of the trip as real computing and the analysis of the road map as abstract computing. What we learn when we do abstract computing can be incorporated into our knowledge when we do real computing. For example, knowing approximately how long the trip will take, will help us plan where we should stop to eat enroute to Paris.

Perhaps the most well-known example of the use of abstract interpretation is strictness analysis. Here functional programmers analyze their programs ahead of time so that some of the parameters to procedures can be considered, not as by need (or by name), but as by value parameters. Functional programming requires, however, that no terminating program fail to terminate even if the parameter is by value. There are functional programs in Scheme (or ML) that do not terminate, but the equivalent by need functional programs do. Therefore, it is clear that in by need functional programmming not all the procedures can take their variables by value. Nevertheless, the more such parameters an analysis can discover, the more efficient will be the resultant program. So, strictness analysis does a reasonable job of finding out which variables to make by value.

In addition, partial evaluation is a classic example of abstract interpretation. A partial evaluator generates specializations of general functions. Here's one way

to think about it. Imagine someone placed constants randomly in our programs. Then we might ask. "Now that we know that this is going to be a constant, how can we improve the program?" It is not exactly like that, but close. The variables appear in place of the constants, and the actual constants don't appear until they can be utilized. Once we know the values of those constants, then we can produce specialized programs. For example, suppose that we are doing member and we know that we are going to know the length of the list in advance of compiling the program. Then, we could generate `(or (eq? x (car l)) (eq? x (cadr l)) ...)` and avoid the explicit setting up of a loop and testing for the end of the list on each iteration. It is that kind of information that lets partial evaluation improve the quality of code. Remember, we do not know the length when we analyze the program, but we "know that we are going to know it in time to use it to good advantage.

These are uses of abstract interpretation. There are others, but most of them have to do with analyzing programs to collect information that can be used by a compiler, interpreter or user. This activity has been going on for some time in computer science, so there is nothing particularly new except that most of the time this analysis has been for first-order languages. Using higher-order languges, ones that allow for the values of expressions to be functions, becomes a much more interesting problem.

In order to both understand what abstract interpretation is and how it is implemented, there are a few concepts that must be absorbed. Specifically, they are the concept of a "potentially recursive" procedure and also the concept of a domain, least upper bounds, and least fixed points. There are other concepts that are assumed, but most of them are common to the gereral study of computer science, such as stacks and caches, and operations on sets.

2 Potentially recursive procedures

As mentioned in the introduction, we must characterize the notion of a *potentially recursive* procedure. Here is a potentially recursive definition of the sum of two non-negative integers.

```
(define potentially-recursive-sum
  (lambda (sum)
    (lambda (n m)
      (cond
        [(zero? n) m]
        [else (+ 1 (sum (- n 1) m))]))))
```

This is clearly not the definition of sum. It is rather tricky, but what is

```
((potentially-recursive-sum *) 3 8)
```

It is 17. That certainly is not the sum (using recursion) of 3 and 8. Consider another possibility. What is

```
((potentially-recursive-sum (lambda (x y) y)) 3 8)
```

It is 9. A way that this program will produce the expected value of 11 would be to

```
((potentially-recursive-sum sum) 3 8)
```

But, we do not have `sum` yet. Instead, we write a program, `fix`, which takes `potentially-recursive-sum` as an argument and turn it into `sum`. Here it is.

```
(define fix
  (lambda (potentially-recursive-sum)
    (letrec
      ([sum
         (potentially-recursive-sum
           (lambda (n m)
             (sum n m)))])
      sum)))
```

Of course, we can choose more generic names.

```
(define fix
  (lambda (potentially-recursive-function)
    (letrec
      ([function
         (potentially-recursive-function
           (lambda (n m)
             (function n m)))])
      function)))
```

or have it work on procedures of any number of arguments.

```
(define fix
  (lambda (potentially-recursive-function)
    (letrec
      ([function
         (potentially-recursive-function
           (lambda args
             (apply function args)))])
      function)))
```

Now using a `fix` operator, like any one of these, will allow us to take a potentially recursive program and make it recursive. So,

```
> (let ([sum (fix potentially-recursive-sum)])
    (sum 3 8))
```

does, indeed, evaluate to 11.

3 Potentially recursive eval

We will be writing our interpreter in a style that will require this `fix` in order to make it recursive. We can actually start to envision its shape: it takes its own evaluator and, of course, it is *not* recursive.

```
(define potentially-recursive-eval
  (lambda (eval)
    (lambda (exp env)
      (match exp
        [,int (guard (integer? int)) (... int)]
        [,bool (guard (boolean? bool)) (... bool)]
        [,var (guard (symbol? var)) (apply-env env var)]
        [(zero? ,e1) ...]
        [(+ ,e1 ,e2) ...]
        [(- ,e1 ,e2) ...]
        [(* ,e1 ,e2) ...]
        [(if ,test-exp ,then-exp ,else-exp) ...]
        [(lambda ,formals ,body) (... `(closure ,formals ,body ,env))]
        [(,rator ,rands ...)
         (... (eval rator env)
              (map (lambda (x) (eval x env)) `(,rands ...)))])))))
```

We have left some of the code out of this interpreter because this is really many interpreters. And what we fill the ellipses with, will determine the nature of this call-by-value interpreter. Keep in mind, however, this is not a recursive interpreter unless we feed it to `fix`, and then the result of doing that is a recursive interpreter.

We, of course, need a procedure to apply procedures, so we create a similar curried version of `apply-procedure`, which is also *not* recursive. It just must get the right `eval`.

```
(define potentially-apply-procedure
  (lambda (eval)
    (lambda (proc args)
      (match proc
        [(closure ,formals ,body ,env)
         (eval body (extend-env formals args env))]
        [,else (error 'potentially-apply-procedure
                "Cannot apply closure: ~s" proc)])))))
```

The way that we can use the potentially recursive interpreter is up to us. Moreover, the theory says that there are many `fixes`, not just the one, above. In fact the `fix` can be very unusual and if the language satisfies some very reasonable mathematical properties, a non-standard `fix` may be used. Using a non-standard `fix` in the presence of domains and primitives whose values are elements of the domain will give us an abstract interpreter. But, before we get to that kind of `fix`, let us look at a variant of the simple `fix`. Here is a `fix` that could be used to build a *stepper*.

```
(define fix
  (lambda (potentially-recursive-eval)
    (letrec
      ([eval
         (potentially-recursive-eval
           (lambda (exp env)
             (stepper exp (lambda () (eval exp env)))))])
      eval)))
```

and the most rudimentary of steppers:

```
(define stepper
  (lambda (exp continue)
    (case (prompt-and-read exp)
      [(n) (let ([x (continue)])
             (display x)
             (newline)
             x)]
      [(q) (abort exp)]
      [else (error 'stepper "Invalid command: ~s" exp)])))
```

This stepper is not recursive. On each invocation of `eval`, `stepper` is called. It prints the expression and prompts for either an n or a q. The n implies that the user wants to see the next expression, etc. The q is used to quit.

4 Abstract Domains

We have two goals. Describe domains and characterize the non-standard `fix`. Let us focus on two domains. First domain D_i is the integers and all the functions that can be defined on the integers. The second domain D_s is similar but restricts the base values to a subset of the integers: -1, 0, and 1. If a number in D_i is positive (negative or 0), then in D_s it is 1 (-1 or 0). Let us consider several simple expressions using both domains.

a. `(+ 5 3)` in D_i is 8 and in D_s is 1. It is 1 because the 5 maps to 1 and the 3 maps to 1. Then adding them together yields a number 8 that also maps to 1.

b. `(* 5 3)` in D_i is 15 and in D_s is 1, since multiplication of two positive numbers yields a positive number.

c. `(* -5 -3)` in D_i is 15 and in D_s is 1, since multipication of two negative numbers yields a negative number.

d. `(* -5 3)` in D_i is -15 and in D_s is -1, since muliplication of numbers of different signs (the reason for the s in D_s) yields a negative number.

At this point, it would be convenient to say, "etc.". Unfortunately, that is not the case. Consider

e. `(+ -5 3)` in D_i is -2 and in D_s is ...

There are two ways to reason about this. -5 maps to -1 and 3 maps to 1. Thus, `(+ -1 1)` must be 0. No, of course not. We were sort of hoping, however, that it might be -1. But, a similar example

f. `(+ -3 5)` in D_i is 2 and in D_s is ...

Once we throw away the magnitude of a number, we cannot recover it. So, `(+ -5 3)` and `(+ -3 5)` must map to the same point in our domain, and it cannot be -1, 0, or 1.

We must add something that says "We have too much information to determine the answer. It is ambiguous." Mathematicians have chosen the word *top* to indicate this case. Now, we no longer have just three elements in D_s, but we are forced to add a new value. Furthermore, we must decide what we want it to mean when it is compared against zero, added to, subtracted from, and multiplied by any other element of our four element domain. There is actually one more element in our domain that is put there for completeness. It is called *bottom*. It represents the opposite of top. Top is too much information and bottom is too little information to determine an answer. Actually, bottom is the complete absense of information with respect to a domain.

4.1 Implementing the sign domain primitives

From the above discussion, we can derive the following procedures. We have written them so that their arguments and values are chosen from the set composed of `bottom`, -1, 0, 1, and `top`.

```
(define Ds-zero?
  (lambda (x)
    (match x
      [bottom 'bottom]
      [,x (guard (integer? x)) (zero? x)]
      [,else 'top])))
```

```
(define Ds-plus
  (lambda (x y)
    (match `(,x ,y)
      [(bottom ,y) 'bottom]
      [(,x bottom) 'bottom]
      [(,x ,y) (guard (integer? x) (integer? y))
       (cond
         [(or (zero? x) (zero? y)) (+ x y))
         [(= x y) x)
         [else 'top])]
      [,else 'top])))
```

```
(define Ds-minus
  (lambda (x y)
    (match `(,x ,y)
      [(bottom ,y) 'bottom]
      [(,y bottom) 'bottom]
      [(,any ,y) (guard (integer? y)) (Ds-plus any (- y))]
      [,else 'top])))
```

```
(define Ds-times
  (lambda (x y)
    (match `(,x ,y)
      [(bottom ,y) 'bottom]
      [(,x bottom) 'bottom]
      [(,x ,y) (guard (integer? x) (integer? y)) (* x y)]
      [,else 'top])))
```

5 Literals

It is now time to consider the first six ellipses in our potentially recursive interpreter. We would like to use exactly the same programs for both domains. We just expect different results. What should we do with an expression like `(+ -3 5)`? In D_i, the `-3` and `5` stay as they are, but in D_s, we would expect them to evaluate to `-1` and `1`, respectively. What should we do with true and false. These are managed by including domain-specific arguments to our interpreter.

Thus, the definition of `potentially-recursive-eval-`*maker* becomes

```
(define potentially-recursive-eval-maker
  (lambda (D-int D-bool D-zero? D-plus D-minus D-times)
    (lambda (eval)
      (lambda (exp env)
        (match exp
          [,int (guard (integer? int)) (D-int int)]
          [,bool (guard (boolean? bool)) (D-bool bool)]
          [,var (guard (symbol? var)) (apply-env env var)]
          [(zero? ,e1) (D-zero? (eval e1 env))]
          [(+ ,e1 ,e2) (D-plus (eval e1 env) (eval e2 env))]
          [(- ,e1 ,e2) (D-minus (eval e1 env) (eval e2 env))]
          [(* ,e1 ,e2) (D-times (eval e1 env) (eval e2 env))]
          [(if ,test-exp ,then-exp ,else-exp) ...]
          [(lambda ,formals ,body) (... `(closure ,formals ,body ,env))]
          [(,rator ,rands ...)
           (... (eval rator env)
             (map (lambda (x) (eval x env)) `(,rands ...)))])))))
```

And `D-int`, `D-bool`, `D-zero?`, `D-plus`, `D-minus`, and `D-times` are respectively the identity function, the identity function, `zero?`, `+`, `-`, and `*` when we are using the values in D_i. In D_s these are, respectively, `Ds-int`, `Ds-bool`, below, and `Ds-zero?`, `Ds-plus`, `Ds-minus`, and `Ds-times`, which we have already defined.

```
(define Ds-int
  (lambda (x)
    (cond
      [(positive? x) 1]
      [(negative? x) -1]
      [else 0])))
```

and

```
(define Ds-bool
  (lambda (x)
    'bottom))
```

With respect to D_s, there is no information that true or false can contribute, so it is bottom.

5.1 Finishing the Integer domain interpreter

At this point, we have three ellipses to fill in and we must also present the definition of the non-standard fix.

We now fill in the remaining ellipses for the standard interpreter and by doing so, we introduce names (`D-if`, `D-closure`, and `D-apply-hov`) for the actions, which we define later in the non-standard interpreter.

```
(define potentially-recursive-eval-maker
  (lambda (D-int D-bool D-zero? D-plus D-minus D-times D-if D-closure D-apply-hov)
    (lambda (eval)
      (lambda (exp env)
        (match exp
          [,int (guard (integer? int)) (D-int int)]
          [,bool (guard (boolean? bool)) (D-bool bool)]
          [,var (guard (symbol? var)) (apply-env env var)]
          [(zero? ,e1) (D-zero? (eval e1 env))]
          [(+ ,e1 ,e2) (D-plus (eval e1 env) (eval e2 env))]
          [(- ,e1 ,e2) (D-minus (eval e1 env) (eval e2 env))]
          [(* ,e1 ,e2) (D-times (eval e1 env) (eval e2 env))]
          [(if ,test-exp ,then-exp ,else-exp)
           ((D-if eval) test-exp then-exp else-exp env)]
          [(lambda ,formals ,body) (D-closure `(closure ,formals ,body ,env))]
          [(,rator ,rands ...)
           ((D-apply-hov eval) (eval rator env)
            (map (lambda (x) (eval x env)) `(,rands ...)))])))))
```

The procedures `D-if` and `D-apply-hov` are curried. This currying pays off by allowing us to lift the applications of the curried functions above the `(lambda (exp env) ...)`. But, this is only an efficiency, so we do not include it here.

We start with the simplest of these: `Di-if`.

```
(define Di-if
  (lambda (eval)
    (lambda (test-exp then-exp else-exp env)
      (if (eval test-exp env)
        (eval then-exp env)
        (eval else-exp env)))))
```

Next we look at `Di-apply-hov`.

```
(define Di-apply-hov
  (lambda (eval)
    (let ([apply-procedure (potentially-apply-procedure eval)])
      (lambda (hov args)
        (apply-procedure hov args)))))
```

This definition can be simplified by first doing an η, changing

```
(lambda (hov args) (apply-procedure hov args))
```

to `apply-procedure`. Then, it can be further improved by deleting the `let` expression, since `(let ([x e]) x)` is the same as `e`. Finally, it can be simplified to

```
(define Di-apply-hov potentially-apply-procedure)
```

But, we prefer not to since we want to compare the D_i and D_s versions.

Then we can define `potentially-recursive-eval` by passing these nine domain-specific functions to `potentially-recursive-eval-maker`.

```
(define potentially-recursive-eval
  (potentially-recursive-eval-maker
    Di-int Di-bool Di-zero? Di-plus Di-minus Di-times Di-if Di-closure Di-apply-hov))
```

6 Introducing the closure cache

The definition of `Di-closure` could be the identity, but we shall introduce a new idea at this time. Here is the definition, which relies on the meaning of the higher-order-value cache `**hov-cache**`.

```
(define cache-maker
  (lambda (binary-predicate? not-found-value-constructor)
    (let ([cache '()])
      (lambda (target)
        (letrec
          ([lookup
             (lambda (table)
               (cond
                 [(null? table)
                  (let ([value (not-found-value-constructor target)])
                    (set! cache (cons `(,target . ,value) cache))
                    value)]
                 [(binary-predicate? target (caar table)) (cdar table)]
                 [else (lookup (cdr table))]))])
          (lookup cache))))))
```

```
(define initialize-hov-cache
  (lambda ()
    (set! **hov-cache** (cache-maker Di-closure-eq? (lambda (target) target)))))

(define Di-closure
  (lambda (closure)
    (**hov-cache** closure)))
```

Every time we build a closure we are creating a data structure that might already exist. If we keep all the closures that we ever build in a table, then we may discover that we have such a closure in the table and we are free to return that one.

We have packaged the behavior of this table in a one-argument procedure called a cache. If the argument to the procedure is found in the table, the associated value is returned. If not, a new table entry is added to the table composed of the argument and the result of a constructor applied to the argument. Then that newly constructed value is returned. To build the cache we must pass a predicate and a constructor. For this cache, we use the identity function for the constructor and we write a closure equality predicate, below.

Consider the following.

```
(let ([curry-plus
        (lambda (x)
          (lambda (y)
            (+ x y)))])
  (let ([plus-5 (curry-plus 5)])
    (let ([plus-five (curry-plus 5)])
      (eq? plus-5 plus-five))))
```

There is no reason to allocate a new closure for `plus-five`, since we already have `plus-5`. By using the cache, we can guarantee that this expresssion always returns true.

6.1 A predicate that tests if two closures are equal

Because the language does not support side-effects, we use a naive definition of `Di-closure-eq?`. Two closures are equal if they contain the same (up to `eq?`) body and their free variables, which will be the same in both closures, have equivalent bindings in their respective evironments.

```
(define Di-closure-equiv?
  (lambda (cl-x cl-y)
    (match `(,cl-x ,cl-y)
      [((closure ,formals-x ,body-x ,env-x)
        (closure ,formals-y ,body-y ,env-y))
       (and
         (eq? body-x body-y)
         (andmap
           (lambda (var)
             (Di-equiv? (apply-env env-x var) (apply-env env-y var)))
           (set-diff (free-vars body-x) formals-x)))])))

(define Di-equiv?
  (lambda (x y)
    (match `(,x ,y)
      [(,x ,y) (guard (integer? x) (integer? y)) (= x y)]
      [((closure ,formals-x ,body-x ,env-x)
        (closure ,formals-y ,body-y ,env-y))
       (Di-closure-equiv? x y)]
      [else #f])))
```

The predicate `Di-equiv?` used in the definition of `Di-closure-equiv?` is domain dependent. If the arguments are both numbers, they must be the same. If they are both closures, they must be `Di-closure-equiv?`. Otherwise it is false.

We now have enough to build the entire standard interpreter.

7 Making test programs recursive

At this point our test programs cannot be very complicated, since we cannot write recursive ones. Actually, we can. We can write fix without using recursion. Thus, we can take a potentially recursive definition of factorial:

```
(define potentially-recursive-factorial
  (lambda (factorial)
    (lambda (n)
      (if (zero? n) 1 (* n (factorial (- n 1)))))))
```

and give it to a procedure `fix`, written without `letrec`, that would allow us to find the `factorial` of 4:

```
> (let ([factorial (fix potentially-recursive-factorial)])
    (factorial 4)).
```

Here is a version of `fix`, which is not recursive, that works.

```
(define fix
  (lambda (potentially-recursive-function)
    (let ([prf* (lambda (prf**)
                  (potentially-recursive-function
                    (lambda (arg) ((prf** prf**) arg))))])
      (prf* prf*))))
```

This fix is also called the applicative-order `Y` combinator. It works and will allow us to test functions like `factorial`.

8 An interesting while loop

Because we use approximations, that is, D_s approximates D_i, we are not thinking about the result as what one would usually want at run-time. So, it is fair to think of abstract interpretation as more generally useful at an earlier time. Abstract interpretation allows us to ask the question "What information can we glean from our program before we run it, possibly sharing the answers with an interpreter or a compiler?" Furthermore, it demands that we construct a domain related to the particular information that we are looking for and that we define the primitives over that domain. But, once we have done that, we need not do anything else.

We are now ready to look at what results are obtained when we run our abstract interpreter. One of the attributes that will be surprisng is that since our domains are finite, even programs that appear not to terminate, will terminate. For example, using the definition of `factorial` above, but passing in -4 instead of 4 will terminate in D_s, but will loop forever in D_i. Also, if instead of decreasing the value of `n` by one on each recursion, we increase it by 1, we would loop indefinitely in D_i, but we do terminate in D_s. So, built-in intuitions will often be useless.

How do we take programs that appear that they will loop indefinitely and have them terminate? To understand this we will have to look at the procedure `finite-fix`, that does the work of `fix`, but in a non-standard way, however, before we do that, we need to study a generally useful function, like `map`, called `fix-maker`.

```
(define fix-maker
  (lambda (binary-op predicate?)
    (lambda (memo thunk)
      (letrec
        ([fix-loop
           (lambda ()
             (let ([value (binary-op (thunk) (cdr memo))])
               (cond
                 [(predicate? value (cdr memo)) value]
                 [else (begin
                         (set-cdr! memo value)
                         (fix-loop))])))])
        (fix-loop)))))
```

This program takes a `binary-op` and a binary `predicate?`, and returns a function. The function returned, takes a cons cell, whose cdr contains a starting value, and a thunk (procedure of zero arguments to repeatedly invoke until the predicate holds. This is sort of like a while loop in more conventinal languages. What is the result of evaluating this expression?

```
((fix-maker max (lambda (m x) (= (- m 1) x)))
 (cons 'a 0)
 (lambda () (random 10)))
```

It yields the first random number between 1 and 9 that is exactly one larger than the largest generated random number. Since, it is initialized to 0, only if there is a (possibly-empty) sequence of 0's followed by a 1, can the result be a 1. If the previously largest generated number is other than an 8 and a 9 is generated, then it loops indefinitely, since 9 is the largest random number.

A natural question is "Why did we use a cons cell, instead of just a starting value, and simply update the value instead of side-effecting the cons-cell?" Such a program would look like this and work the same just passing in 0, instead of `(cons 'a 0)` for same test expression.

```
(define fix-maker
  (lambda (binary-op predicate?)
    (lambda (memo thunk)
      (letrec
        ([fix-loop
           (lambda ()
             (let ([value (binary-op (thunk) memo)])
               (cond
                 [(predicate? value memo) value]
                 [else (begin
                         (set! memo value)
                         (fix-loop))])))])
        (fix-loop)))))
```

and this can be easily rewritten without side-effects:

```
(define fix-maker
  (lambda (binary-op predicate?)
    (lambda (memo thunk)
      (letrec
        ([fix-loop
           (lambda (memo)
             (let ([value (binary-op (thunk) memo)])
               (cond
                 [(predicate? value memo) value]
                 [else (fix-loop value)])))])
        (fix-loop memo)))))
```

Therefore, there must be some special reason for using a cons-cell. Generally, when a cell is used in place of a value, it is because more than one part of the program is going to have access to the cell and see the results being stored in the cell. And that is exactly the case here. One part of the program will be busy fixing the cell, while another part of the program will be reaping the benefits of the side-effects that are occurring while the fixing is taking place.

9 A fix for D_s

We now have enough tools to look at the heart of this system: `finite-fix`:

```
(define fix-finite
  (lambda (potentially-recursive-eval)
    (let ([stack '()])
      (letrec
        ([eval (potentially-recursive-eval
                 (lambda (exp env)
                   (cond
                     [(assq exp stack) => cdr]
                     [else (let ([memo (cons exp 'bottom)]
                                 [thunk (lambda () (eval exp env))])
                             (set! stack (cons memo stack))
                             (let ([value (fix-memo memo thunk)])
                               (set! stack (cdr stack))
                               value))])))])
        eval))))
```

First of all, compare it to the recursive `fix` that we defined above. The definition of `eval` uses `letrec` and passes a procedure of the form `(lambda (exp env) ...)` to `potentially-recursive-eval`. So, structurally they are the same. That is a good start. So, only the body of the procedure `(lambda (exp env) ...)` changes. Both have the same goal and that is to find the value of the expression in the environment. That is where the similarites end, however.

But, let us analyze what is going here a bit further. There is the stack, which is an association list of pairs composed of an expression and the current value of the expression. If an expression is in the stack, then it is one of the expressions being evaluated. If not, we add it to the stack and associate bottom with it.

Let us assume for the moment that we have an expression that is not in the stack Therefore, we push a pair of the expression and bottom onto the stack of pairs. Then we evaluate the expression by invoking `fix-memo`, which we define below. When it returns with a value, we pop the stack and return the value. By pushing the expression onto the stack and testing for its membership in the stack when we re-enter `eval` with the same expression, we avoid re-evaluating the expression; it is already being evaluated. But, couldn't it be a recursive call of the same expression? That's the point, we are avoiding the evaluation of recursive calls with this test, so

that any other attempts to evaluate this expression will be thwarted. The value of the recursive call is the current value of the expression.

10 Least upper bounds and least fixed points

Before, we can define the procedure `fix-memo`, we must learn about least upper bounds and least fixed points. To some extent, we have already seen both ideas in the evaluatin of

```
((fix-maker max (lambda (m x) (= (- m 1) x)))
 (cons 'a 0)
 (lambda () (random 10))).
```

For example, when we see the first random number that is larger by one than the largest thus far generated random number, we have reached a fixed point. The answer to our program cannot change. And, `max` is a kind of naive least upper bound. With the definition of `max`, one of the two arguments must be the result. With least upper bounds, however, that need not be the case. We can get a better understanding of what least upper bounds are by using a simple metaphor related to organizations.

Let us imagine a corporation. Two friends, P and Q, working in different departments get an idea that they think would profit the organization. To whom should they take their idea? P says, "I will take it to my supervisor." Q responds "Then you will get all the credit for the idea." They want the credit for the idea to be shared equally. Therefore, they decide to take the idea to the supervisor that most directly manages both of their departments. That supervisor is the least upper bound of P and Q. Because we stipulated that P and Q are in different departments, we have ignored some important special cases. If P (Q) directly or indirectly supervises Q (P) then P (Q) is the least upper bound. Also, if P is Q, then P is the least upper bound. (These special cases are where least upper bound is like `max`.) As we take least upper bounds of different values, we continue to move higher and higher in the organization. For example, let's say that R is P's and Q's least upper bound. He likes the idea. So, he tells one of his friends, S, who significantly improves the idea. Thus, they we have the same situation again. R and S have to find their least upper bound to communicate the idea to, etc. The idea is moving up the organization. Does it necessarily reach the CEO, or might some lesser supervisor turn the idea into profit for the corporation? The person who acts on the idea instead of seeking counsel from friends or supervisors is a fixed point. The idea is fixed with the person who did not transmit it further. It is the least fixed point because we are only taking least upper bounds.

Now, we can define `fix-memo`.

```
(define fix-memo (fix-maker Ds-lub Ds-equiv?))
```

On each iteration, the expression with the environment, packaged as a thunk, is evaluated. The least upper bound (`Ds-lub`) of that value and the value in the alist pair is determined. If no change has occurred (it has reached a fixed point), then

we are done and we return the value. Otherwise, we use the least upper bound as our next approximation by making it the current value of the expression. Each time we re-enter the code of `eval`, however, we are going to obtain the benefits of this new approximation. Thus, inside `eval`, even though the expression is still in the stack we get all the advantages of the side-effects that occur in `fix-memo`. If not, then bottom would be the result of all recursive calls. This is precisely where the sharing that was mentioned above becomes important.

So, in our program, we keep taking least upper bounds until we reach a least fixed point. That fixed point is the value of the target expression in the environment. Below is a definition of `Ds-lub` for D_s.

```
(define Ds-lub
  (lambda (x y)
    (match `(,x ,y)
      [(,x ,y) (guard (Ds-equiv? x y)) x]
      [(bottom ,y) y]
      [(,x bottom) x]
      [(,x ,y) (guard (integer? x) (integer? y)) (Ds-bass-lub x y)]
      [((hov ,closures-x) (hov ,closures-y))
       (**hov-cache** (union closures-x closures-y))]
      [((hov ,closures-x) ,y) `(top ,closures-x)]
      [(,x (hov ,clsoures-y)) `(top ,closures-y)]
      [((top ,closures-x) (top ,closures-y))
       `(top ,(union closures-x closures-y))]
      [(top ,y) 'top]
      [(,x top) 'top]
      [,else (error 'Ds-lub "Cannot take lub of ~s" "and ~s" x y)])))
```

In order to understand this program, we shall look at this definition closely. If the two arguments are the same, then either will do as the least upper bound. If one of them is bottom, then the other one is surely the least upper bound, even if it, too, were bottom. If they are both one of the three basic values, then we consult the domain specific `Ds-base-lub` for that domain. It will return top in all cases, since we know from the first test that they are not *Ds-equiv*. Because we are working in a world that contains functions as values, we must have a notion of what it means to take least upper bounds of such values. Interestingly, when we form a D_s domain value from a `lambda` expression, it starts out, not as a closure as in the standard case, but as a set of just one closure. As it moves up the value domain with other functions, we form the union of their respective sets. In the case where we have two such sets, we form their union, but if it has been previously built, just as in the stardard interpreter, we need not build it. That is why `**hov-cache**` is used in the definition of `Ds-lub`. Since the argument to `**hov-cache**` is a *set* of closures, the cache predicate is a bit more involved. Nevertheless, it is just "Are two sets of closures equal if we use `closure-eq?` when comparing individual closures." The rest of the cases have a similar flavor and we need not worry about them. The procedure `Ds-base-lub` is obvious when the shape of the organization is presented.

Bottom (top) is, not surprisingly, at the bottom (top) and the other values are directly above (below) bottom (top). In order to understand `Ds-equiv?`, we need to know that it is only concerned with the non-artificial values: -1, 0, and 1. and with higher-order function values (i.e., sets of closures).

```
(define Ds-equiv?
  (lambda (x y)
    (match `(,x ,y)
      [(,x ,y) (guard (eq? x y)) #t]
      [((hov ,closures-x) (hov ,closures-y)) (set-equiv? closures-x closures-y)]
      [((top .closures-x) (top ,closures-y)) (set-equiv? closures-x closures-y)]
      [,else #f])))
```

If the arguments are identical, they are value equivalent. If they are one of the two types of sets, than we see if the sets are made of the same closures relative to this definition of `Ds-closure-equiv?`.

```
(define Ds-closure-equiv?
  (lambda (cl-x cl-y)
    (match `(,cl-x ,cl-y)
      [((closure ,formals-x ,body-x ,env-x)
        (closure ,formals-y ,body-y ,env-y))
       (and
         (eq? body-x body-y)
         (andmap
           (lambda (var)
             (Ds-equiv? (apply-env env-x var) (apply-env env-y var)))
           (set-diff (free-vars body-x) formals-x)))])))
```

11 Finishing potential recursive eval for finite domains

We next define the three remaining procedures that were introduced in the last definition of `potentially-recursive-eval`, this time, however, we do it for D_s. We start with `Ds-closure`. Since the value of a lambda expression is a singleton set of closures, we must consult the cache to see if that set already exists.

```
(define initialize-hov-cache
  (lambda ()
    (set! **hov-cache** (cache-maker Ds-closure-equiv? (lambda (target) target)))))

(define Ds-closure
  (lambda (closure)
    (**hov-cache** (list closure))))
```

What does it mean to evaluate an application of such a procedure? It is quite simple. We merely apply all the closures in the set to the arguments one at a time, and accumulate their least upper bounds as the value of the entire application: Compare this with the definition given above for infinite domains.

```
(define Ds-lub-map
  (lambda (f l)
    (cond
      [(null? l) bottom]
      [else (Ds-lub (f (car l)) (Ds-lub-map f (cdr l)))])))

(define Ds-apply-hov
  (lambda (eval)
    (let ([apply-procedure (potentially-apply-procedure eval)])
      (lambda (hov args)
        (match hov
          [bottom 'bottom]
          [(hov ,closures)
           (Ds-lub-map (lambda (proc)
                         (apply-procedure proc args))
             closures)])))))
```

We are left with just one more operation. We must decide what it means to evaluate conditional expressions. Do the rules change when we are assured of termination? Yes. What we do now is form the least upper bound of the two arms of the conditional. So, unlike the standard interpretation, we can (and must) evaluate both the *then* and the *else* expressions of a conditional:

```
(define Ds-if
  (lambda (eval)
    (lambda (test-exp then-exp else-exp env)
      (let ([test (eval test-exp env)])
        (Ds-lub (eval then-exp env) (eval else-exp env))))))
```

This is where we lose information. We do not determine if the test evaluates to true or false, so we have to take the least upper bound of both branches and this is less accurate than either branch, since `Ds-lub` tends to move us up the organization toward the top.

Thus, our new interpreter is built by first passing in the domain-specific arguments then that is passed to `fix-finite`.

```
(define Ds-eval
  (fix-finite
    (potentially-recursive-eval-maker
      Ds-int Ds-bool Ds-zero? Ds-plus Ds-minus Ds-times Ds-if Ds-closure Ds-apply-hov)))
```

Of course, we mustn't forget to initialize the **hov-cache** for this domain before we invoke `Ds-eval`.

This completes the details of the abstract interpreter. While studying this code, consider the following questions. Is the `**hov-cache**` necessary, optional, or useless in D_i and D_s? Would the programs terminate without the `**hov-cache**` in D_s? Is there any reason to use `eval` on the `test-exp` in `Ds-if`?