6.945 Final Project
Dwimiykwim

Ziv Scully (ziv@mit.edu)
Daniel Shaar (dshaar@mit.edu)

1 Argument Inference

We have written an interpreter for a Scheme-like language that supports automatic inference of procedure arguments. The core of the argument inference system is in two new special forms: `madlab` and `madblock`.

The `madlab` special form is a variation of `lambda`. Specifically, the form creates a procedure that can take its arguments in any order. Instead of using position to match given arguments to the variables they are bound to, a `madlab` specifies a predicate for each input variable that the argument bound to that variable must satisfy. For example,

```
(define madmap
  (madlab ((xs list?) (f procedure?))
    (map f xs)))
```

is a variant of `map` that can take its arguments in either order. We call the resulting procedures "madlab procedures" or simply "madlabs". For most purposes, madlabs are ordinary compound procedures that happen to have unusually flexible interfaces. For example, if we define

```
(define (curry f . args)
  (lambda more-args
    (apply f (append args more-args))))
```

then both `(curry madmap exp)` and `(curry madmap (iota 16))` work as expected, raising e to each of the elements of an input list and mapping an input function over the integers from 0 to 15, respectively. Note that the argument matching is not done until the madlab is applied. To demonstrate this, consider

```
(define silly
  (madlab ((xy (member-of? '(x y)))
           (yz (member-of? '(y z))))
    (list xy yz)))
```

and the partial application `(curry silly 'y)`, where `member-of?` has the obvious namesake meaning. When it's applied to `'x`, the `'y` is matched with `yz`, but when it's applied to `'z`, the `'y` is matched with `xy`.

The `madblock` special form is a variation of `begin`. Just like `begin`, it groups together a sequence of expressions, evaluates each of them in turn, and returns the result of the last one. The only difference is that the result of each evaluation in the sequence is added to an *inference context*, which, as we will soon explain, is the list of values available to `??` (which performs inference). The inference context is dynamically bound (the interpreter uses MIT Scheme's `fluid-let`) to the empty list at the beginning of each `madblock`, so the inference context is empty at the start of the sequence. Our interpreter makes it easy to refer explicitly to values of expressions earlier in the sequence using the `define` special form by having `define` return the result of evaluating its body as opposed to an unspecified value or the name it was bound to.

The `??` special form is what Dwimiykwim is all about. It takes a madlab and any number of other expressions and applies the madlab to the given expressions plus any additional necessary values from the inference context. That is, it infers what arguments to pass to the given madlab. For instance,

```
(madblock
 (curry list 'say)
 'dont-pick-me-im-a-symbol-not-a-procedure-or-list
 "a very distracting string"
 '(1 2 5 3-sir 3)
 (?? madmap))
```

returns `((say 1) (say 2) (say 5) (say 3-sir) (say 3))`. Inference only succeeds if there is an unambiguous matching between the madlab's predicates and the union of the given expressions and values from the inference context with the additional constraint that every given expression is matched. (See Section 3 for details.) For example, only the first two inferences in

```
(madblock
 (curry list 'say)
 (define xs '(1 2 5 3-sir 3))
 (?? madmap)
 (?? madmap xs)
 (?? madmap))
```

will succeed. A single procedure, `(curry list 'say)`, is in the inference context the whole time. When the first `infer` happens, there is also only one

list, so inference succeeds. Note, however, that the resulting list is added to the inference context. The second `??` is explicitly passed a list, and inference succeeds thanks to this constraint. By the time the third `??` happens there are three lists in the inference context, so inference fails due to ambiguity.

Just as the body of a `lambda` with multiple expressions desugars to a single `begin` expression, the body of a `madlab` desugars to a single `madblock`. Additionally, each of the argument variables is added as an expression to the beginning of the madblock. The effect of this is that `??` can be used freely in the body of a `madlab` and the arguments are automatically added to the inference context. If for some reason we need to keep the arguments out of the context, we can write body as a `madblock` explicitly. There is nothing to worry about with regards to nesting because each `madblock` starts with a fresh inference context. In fact, the intended style is for `madblock` to be rare and mostly invoked implicitly through `madlab`. This is because `madlab` creates a new environment, which is a good idea given the intended synergy with `define`.

2 Tagging

The first piece of this project that is intended to set groundwork for inference and matching is a lightweight system for tagging data. The motivation for this system is that it creates a method for describing data with useful characteristics that can be used for performing argument inference with tag checks as predicates. Since tags can be arbitrarily created by the user and are not limited like type systems usually are, the language becomes more flexible.

Tagged data consists of a value and a list of tags, which can be used to describe the significance of the data. We can add more tags to data as it moves through a program, and remove the tags if we would just like the value. However, we can always perform default operations on tagged data without removing the tags in advanced because procedures untag data before processing it outside the interpreter. This is implemented by making a procedure "tag-aware". Being "tag aware" removes tags in logical places, so that Scheme functions and other necessary functions can operate as intended when the tags are not important. For example, car and cdr should be "tag aware" because the record type for tagged data is a list, but we only care for the actual data being tagged.

As stated earlier, since MIT Scheme does not come with a nice type system by default, tags allow us to work with variables by more generally describing their characteristics. The primary usefulness of these tags lies in argument matching. By giving values tags, we can match tags to tag predicates to determine if a variable is an appropriate argument to a function. Functions can request that

certain variables have certain tags, or more generally, that these variables satisfy arbitrary predicates. This allows us to verify that the user is performing the intended task by forcing him/her to think in advance about what really belongs in the function call.

A simple example to illustrate this point with out-of-order operands is ordering tagged operands after receiving them in the wrong order. The syntax for tagging data is `~~`. In this example, we tag the numbers 3 and 4 with `x` and `y`, respectively. We make a madlab `x-then-y` that given some `x` and some `y`, will make a list containing the `x` followed by the `y`. This uses the argument matching with the `has-tag?` predicate: `((has-tag? t) x)` is true when `x` has tag `t`.

```
;dwimiykwim>
(define (x-then-y (x (has-tag? 'x))
                  (y (has-tag? 'y)))
  (list x y))
;dwimiykwim>
(x-then-y (~~ 'y 4) (~~ 'x 3))
;=> (#(<tagged> 3 (x)) #(<tagged> 4 (y)))
```

Although this tag system does not have to be used in order to perform the core features of Dwimiykwim—argument inference and out of order operations—it is convenient in many cases, especially in describing the semantics of the arguments. For instance, we might use `x` and `y` as tags for x and y coordinates of points on a plane, which can help avoid bugs where we accidentally pass in the wrong coordinate.

3 Bipartite Matching

Inferring arguments of madlabs reduces to the following problem in graph theory. We are given a bipartite graph with vertex partitions A and B satisfying $|A| \leq |B|$ along with a subset $B^* \subseteq B$ of "required" vertices, and we ask whether there exists a unique matching of size $|A|$ such that every vertex of B^* is matched. A is the set of predicates of the madlab's arguments, B^* is the set of values passed in explicitly, B is the union of B^* and the set of values in the inference context, and there is an edge between $a \in A$ and $b \in B$ if and only if b satisfies a.

This problem is quite easily solved by a variation on the traditional maximum bipartite matching algorithm, which we review quickly here. We refer to vertices in A as "sources" and vertices in B as "targets". Let E be the edge set of the graph and $M \subseteq E$ be a matching. We think of edges in M as being

oriented from B to A and edges in $E \setminus M$ as being oriented from A to B. An *augmenting path* of M is a path from an unmatched source to an unmatched target that follows only edges in $E \setminus M$ from sources to targets and only edges in M from targets to sources. Given an augmenting path of a matching, swapping the orientations (that is, inserting or removing from the matching as appropriate) of all the edges in the path yields a larger matching: all intermediate vertices in the path remain matched, but the previously unmatched endpoints are now matched. This means a maximum matching has no augmenting paths. It is well-known that the converse also holds: if a matching has no augmenting path, it is not maximal. Therefore, to find a maximum matching, we repeatedly search for augmenting paths, flipping edge orientations when we find them, until there are no more, at which point the edges from B to A are the edges of the maximum matching.

Our problem differs from traditional maximum matching in two ways. First, we require that some targets $B^* \subseteq B$ be matched because we must use every argument that was passed in explicitly. To do this, we start by finding a maximum matching M of B^* with A, reporting failure if $|M| < |B^*|$, and then matching A with B using M as a starting point, guaranteeing that all of B^* will remain matched. Second, we are concerned with whether the maximum matching is unique because there must not be multiple ways to match all the predicates with arguments. To do this, given a maximum matching M, for each edge $e \in M$, we attempt to find an augmenting path of the matching $M \setminus \{e\}$ in a graph with reduced edge set $E \setminus \{e\}$, with the additional requirement that if e was incident with a vertex $b \in B^*$ then the augmenting path must finish at b. That these algorithms are valid follows from the fact that, given a non-maximum matching M, there is a maximum matching containing a vertex unmatched by M only if there is an augmenting path of M containing that vertex, which is a slight strengthening of the result mentioned earlier.

Once the graph is constructed, all of this can be done in quadratic time. Our implementation is purely functional and makes liberal use of linear-time list procedures, so it is slower than this by approximately another quadratic factor, but procedures generally don't take in more than, say, 8 arguments, and $O(8^4)$ is perfectly acceptable running time for our prototype.

4 Debugging

In addition to constructing mechanisms for tagging, inferring arguments in function calls, and matching function arguments, we have created a way for the user to easily correct ambiguity errors that arise from more than one pos-

sible way to interpret an inference. For example, consider the simple madlab below:

```
(define num-str
 (madlab ((x number?) (y string?))
         (list y x)))
```

If we were to call a madblock that had two number values and strings in its inference context, then upon inference, we would have an ambiguity error. To handle this case, we put the program into debug mode, and display the context, edges, and required args to the user. The context is indexed, so that the user does not have to spend time typing in every expression they would like to force in the matching. Using this information, the user is then expected to add new required args to settle any ambiguities. The reason we ask for required arguments is because those will be used in the matching and will usually eliminate multiple matchings. Once the user enters some combination of unambiguous args, we notify the user of the successful matching and exit debug mode. A sample workflow for `num-str` would be:

```
(madblock
 1
 "foo"
 2
 "bar"
 (?? num-str))
=== Dwimiykwim Tawimiydkwim ===
Context:
(0 "bar")
(1 2)
(2 "foo")
(3 1)
Edges:
(x 1 3)
(y 0 2)
Required:
()
New required:
(0)
Ambiguous matching!
New required:
(0 3)
Unambiguous matching! Terminate debugging mode!
```

In this example, the user was allowed to test multiple sets of required arguments, until one achieved a good matching. The user would then be expected to go and correct the code by specifying those as required arguments in the inference. The new program would now be:

```
(madblock
 1
 "foo"
 2
 "bar
 (?? num-str 1 "bar"))
```

Since the user is providing so many arguments, it would not be very useful to use inference unless they modified the code to remove some expressions.

Under the hood of the debugger, when we encounter a bad matching, we enter debug mode, indexing each item in the context, and print all the information we know about the matching that had an error. We ask the user to give us a list of newly required arguments, verifying that this is indeed a list of numbers, and checking using the matching function if the ambiguity resolves. In the case where the user has not specified enough required arguments, we ask the user for a new set of required arguments, not saving the initial choice. This continues until either the user exits the debugger, or a good match is found and program terminates with an error.

5 Demo and Discussion

As a taste of writing a real program with Dwimiykwim and a demonstration of how much data flow can be inferred, we devote this section to walking through the process of a sample program. We will start by building an evaluator for arithmetic expressions then extend it to include Scheme-style `let` bindings. Along the way, we'll discuss some further minor features of Dwimiykwim that would have been distracting details in the previous exposition but are nice to have in practice.

The main task for our arithmetic evaluator is evaluating binary operations. (We leave generalizing to n-ary operations as an exercise to the reader best tried after reading this entire section.) As a fist step, we should be able to identify when an expression is a binary operation and, if it is one, extract the operation name and arguments. Just as `(define (f x1 ... xn) ...)` desugars to a `lambda` expression, `(define (f (x1 p1?) ... (xn pn?)) ...)` desugars to a `madlab` expression where `x1` must satisfy `p1?` and so on.

```
(define (binop? exp)
  (if (pair? exp)
      (member (car exp) '(+ - * /))
      #f))
(define (binop-op (exp binop?))
  (car exp))
(define (binop-left (exp binop?))
  (cadr exp))
(define (binop-right (exp binop?))
  (caddr exp))
```

Here we use madlabs not for disambiguating arguments but to allow inference of the single argument when there's only a single value in the inference context that could possibly be a binary operation expression.

Our application procedure is straightforward. Both the left and right operands are numbers, so we need to tag them to disambiguate them.

```
(define (apply-binop (op symbol?)
                     (left (has-tag? 'left))
                     (right (has-tag? 'right)))
  ((cadr (assq op
               (list (list '+ +)
                     (list '- -)
                     (list '* *)
                     (list '/ /))))
   left
   right))
```

Even though `left` and `right` have tags, we can still apply procedures from the underlying Scheme to them. The tags are automatically stripped from the arguments of all such procedures with a short list of exceptions including `cons` and `list` that allow for tagged data to exist in larger data structures.

The plan for evaluation is to have a case each for the numberic primitives and a binary operations. We choose a case with the following madlabs, which are once again madlabs for the sake of inference rather than argument ordering.

```
(define (exp? x)
  (null? (tags x)))
(define (primitive? (exp exp?))
  (number? exp))
(define (operation? (exp exp?))
  (binop? exp))
```

Technical note: we can't just define `binop?` as a madlab that only accepts an expression because during inference we need to apply it to many values that aren't expressions, though this could definitely be handled more intelligently by a future version of Dwimiykwim. At this point it is probably clear to the reader that we have not gone to great lengths to make all of our predicates airtight, but our somewhat lenient definitions of `exp?` and `binop?` suffice for this demo.

We can now finally define our main procedure. Recall that `??` invokes argument inference, `~~` tags its second argument with its first, and that the body of `madlab` expressions desugar to madblocks. This procedure introduces `madblock-inherit`, which is a variant of `madblock` that uses the previous inference context as a starting point instead of an empty context. It is probably not wise to use `madblock-inherit` widely, but here it is mostly harmless and slightly reduces clutter. It is still somewhat safe in that values added to the inference context don't leak into the previous inference context.

```
(define (eval (exp exp?))
  (cond
   ((?? primitive?)
    exp)
   ((?? operation?)
    (madblock-inherit
     (~~ 'left (?? eval (?? binop-left)))
     (~~ 'right (?? eval (?? binop-right)))
     (?? binop-op)
     (?? apply-binop)))))
```

Notice that we almost all of the data flow is inferred: the only explicit arguments given are for evaluating the left and right subexpressions, both of which are expressions, so we need to disambiguate them somehow. The `??` isn't necessary for those `eval` applications, but the intended style is for essentially every application to be an inference, which, as we'll see shortly, makes it robust to future expansion. A quick test shows that everything works as expected.

```
;dwimiykwim>
(eval '(- (+ 3 4) 9))
;=> -2
```

Adding `let` bindings to our expression language turns out to be surprisingly little work. In particular, despite the fact that the evaluation procedure will pass around a variable context, the two cases we defined above won't change at all!

The basic operations for the `let` expressions are below. Given what we've seen so far, they are all straightforward.

```
(define (let? exp)
  (if (pair? exp)
      (eq? (car exp) 'let)
      #f))
(define (let-vars (exp let?))
  (map car (cadr exp)))
(define (let-vals (exp let?))
  (map cadr (cadr exp)))
(define (let-body (exp let?))
  (caddr exp))
```

The story is similar for context operations with one new toy: `~~:delq` removes its first argument from the tag list of its second.

```
(define ctx?
  (has-tag? 'ctx))
(define empty-ctx
  (~~ 'ctx '()))
(define (list-of p?)
  (lambda (xs)
    (if (list? xs)
        (every p? xs)
        #f)))
(define (bind (ctx ctx?)
              (vars (list-of symbol?))
              (vals (list-of number?)))
  (~~ 'ctx (append (zip list vars vals)
                   (~~:delq 'ctx ctx))))
(define (lookup (ctx ctx?)
                (var symbol?))
  (cadr (assq var ctx)))
```

We are almost ready to define evaluation, but first we need a new inference primitive, `??:apply`. It is actually not a primitive at all but defined in Dwimiykwim's standard library.

```
(define (??:apply proc . args)
  (lambda more-args
    (infer proc (append args more-args))))
```

In fact, not even ?? is a primitive.

```
(define (?? proc . args)
  (infer proc args))
```

The real primitive is `infer`, which acts like ?? but takes a list instead of a variable number of arguments. This enables the definition new inference primitives like `??:apply`. We use the word "primitive" not literally but to connote that it would be bad style to define too many of these context-dependent procedures.

With these pieces, we define predicates for the two new cases and a new evaluation procedure. The fact that we used ?? for the `eval` applications earlier means we don't need to change the first two cases.

```
(define (variable? (exp exp?))
  (symbol? exp))
(define (declaration? (exp exp?))
  (let? exp))
(define (eval (ctx ctx?)
              (exp exp?))
  (cond
   ((?? primitive?)
    exp)
   ((?? operation?)
    (madblock-inherit
     (~~ 'left (?? eval (?? binop-left)))
     (~~ 'right (?? eval (?? binop-right)))
     (?? binop-op)
     (?? apply-binop)))
   ((?? variable?)
    (?? lookup))
   ((?? declaration?)
    (madblock-inherit
     (map (??:apply eval) (?? let-vals))
     (?? let-vars)
     (eval (?? bind) (?? let-body))))))
```

Once again, we have to provide very few arguments explicitly, and once again, `eval` does what it's supposed to do.

```
;dwimiykwim>
(eval
 empty-ctx
 '(let ((x (+ 2 2))
        (y (- 6 3)))
    (+ (* x x) (* y y))))
;=> 25
```

What happened throughout this demo was that we shifted work from the top-level procedure `eval` to its helper procedures. While this first draft of Dwimiykwim is admittedly clunky, there is a glimmer of hope in the way that the helper procedures self-organize into the top-level procedure with just a few essential hints as guidance. The approach also benefits safety somewhat by requiring the programmer to specify very specifically how helper procedures are used. This information typically exists anyway in documentation (or, worse, in a single programmer's fallible memory), and making use of it to write the program is a potentially rich area for further research.