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 three new special forms: `madlab`, `madblock`, and `infer`.

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 `madlab` special form is a variation of `begin`. Just like `begin`, `madblock` 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 availabe to `infer`. 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 `infer` 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, as its name suggests, `infer` 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)
 (infer 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 2 for details.) For example, only the first two inferences in

```
(madblock
 (curry list 'say)
 (define xs '(1 2 5 3-sir 3))
 (infer madmap)
 (infer madmap xs)
 (infer 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 `infer` is explicitly passed a list, and inference succeeds thanks to this constraint. By the time the third `infer` 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 `infer` 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 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.