Generic Operator Discovery:
10,000 Monkeys with 10,000 Lambdas

Laura Harris and Bryan Newbold for 6.945
{lch,bnewbold}@mit.edu

$May 11, 2009$

Abstract

We have implemented a simple system which enables the discovery and exploration of generic operators and brute force predicate satisfaction. Our procedures build on top of existing predicate-based operator dispatch databases; this allows existing code to be reused in useful and unexpected ways. In this write up we describe our code, give a few simple demonstrations (including one with a native graphic user interface), and mention some potential applications.

Contents

Generic Operator Discovery System

The normal purpose of a generic operator dispatch system is to allow the programmer or user to use a single operator with many different object types or combinations of object types. Mature libraries and codebases may have dozens of generic operators defined for domain-specific data structures; these generic operations often represent the core functionality offered by the system. For large systems, those with which the user is unfamiliar, or those with poor documentation, it can be daunting to find the operation desired. By using "operator discovery" techniques, the operator dispatch system can be reverse engineered to find all of the generic operations which can be applied to given arguments. In addition to facilitating user exploration, these techniques can be used to improve the robustness of computing systems, as part of automated problem solving, as a testing tool, and for the automated generation of higher level programs.

The generic operator system we have built upon uses predicate dispatch; for an overview of this strategy see Ernst, Kaplan, and Chamber's paper "Predicate dispatching: A unified theory of dispatch" (1998). The version we used for MIT/GNU Scheme was distributed by the 6.945 staff and is included in the appendix as `ghelper.scm`. The exact same dispatch system is used in the `scmutils` classical mechanics software package, which allowed us to experiment with an existing software system.

Implementation

For examples and demonstrations of the system, see the applications section and the file `discovery-examples.scm` in the appendix.

Review of Predicate Dispatch

Predicate dispatch works by choosing the first *handler* whose associated *predicates* all return true for a given set of arguments; a list of predicate/handler pairs is stored in a tree structure for each generic operator.

A few crucial procedures, globals, and data structures are defined in `ghelper.scm:`

generic-operator-table This is the global table of generic operators. It is an `eq-hash` table which associates operator record *keys* (which define the arity) with predicate/handler tree *values*. In addition, for "named" operators, the symbol representing an operator is added as a second *key* pointing at the same predicate/handler tree *value*.

(make-generic-operator arity default-operation #!optional name) This procedure creates a new record in the `*generic-operator-table*` for the given arity; it returns an operator procedure which is usually bound in the user's environment and when applied initiates the procedure dispatch process. If not null, the default-operation is bound (using assign-operation) as an any-argument-accepting default handler. If passed, the name (which should be a symbol) is bound as a redundant key in the `*generic-operator-table*`. **defhandler** is an alias for make-generic-operator.

(assign-operation operator handler . argument-predicates) This procedure adds a new predicate/handler pair to an operator's tree in the `*generic-operator-table*`. The binding is done with `bind-in-tree` (see below).

(bind-in-tree keys handler tree) This procedure simply adds a new handler (with the argument predicates *keys*) in a given generic operator's dispatch *tree*.

Procedures

The actual implementations of these procedures can be found in the appendix.

(discover:opers-for . args)

This procedure returns all of the operators which can be applied to the arguments. The return value is a list of the keys from *generic-operator-table* which are associated with predicate/handler trees matching the arguments. This is the core of the discovery system.

(discover:named-opers-for . args)

This procedure is the same as discover:opers-for except that it only returns lookup keys which are symbols (thus the original operator record was defined with a name symbol).

(discover:named-opers) This procedure returns a list of *all* the "named" generic operators in the `*generic-operator-table*` ; it is useful to determine the size of scope of an unknown software system.

(discover:apply-name name . args) This procedure allows "named" operator symbols to be treated like actual operator procedures: it initiates the dispatch process for the predicate/handler tree associated with *name* for the given *args*.

(discover:apply-all . args) This procedure finds all of the operators which can act on the given args, then returns a list with the results of applying each of these operators.

(discover:apply-all-name . args) This is identical to `discover:apply-all` except that it only applies "named" operators.

(discover:satisfy pred? . args) This procedure attempts to satisfy the given predicate by repeatedly applying all possible operators the arguments (and the return values of these applications recursively). It operates as a breadth first search and returns the first matching return value.

(discover:satisfy-sequence pred? . args) This procedure is similar to `discover:satisfy` except that it only applies "named" operators and it maintains a record of which operators were applied to obtain a given return value; it will also return all of the matching return values for a given "depth" of search.

Room for improvement

The code for all of these procedures is rather ugly and complicated due to the crude data structures used: for example discover:satisfy-sequence has an internal variable to store potential solutions as a list with the first argument being a list of arguments (always a single element after the first application of operators) and all subsequent operators being a record of the operators applied to obtain those arguments. This could almost certainly be re-implemented in a more elegant functional style.

The predicate/handler tree format does not currently include a name symbol for the given operator. Perhaps the name symbol could also be determined by searching the environment bindings, but this does not seem like a great idea (search would be slow?).

Almost all of the implementations are ripe for trivial optimization: for example `discover:named-opers-for` just filters the results of `discover:opers-for`; it could be much more efficient if it filtered out non-symbol operators earlier in the search process.

Applications

scmutils Package

`scmutils` is an MIT/GNU Scheme package for math and physics, focused on classical mechanics[1]. It uses the predicate dispatch system defined in `ghelper.scm`, so it is a natural candidate for experimentation with the discovery tools.

The default scmutils band file has over 90 generic operators; the named ones can be discovered:

```
(discover:named-opers)
;Value:  (+ one-like cos dot-product expt one?  * gcd
partial-derivative acos exp atan2 cosh imag-part one = conjugate
zero?  / zero-like abs sinh identity?  sin asin derivative angle
magnitude inexact?  type apply identity make-polar arity real-part -
invert negate identity-like trace determinant sqrt zero log square
make-rectangular type-predicate atan1)
```

The operators applicable on the 3x3 identity matrix are:

```
(discover:named-opers-for
   (matrix-by-rows '(1 0 0) '(0 1 0) '(0 0 1)))
;Value:  (one-like cos exp conjugate zero?  zero-like identity?  sin
inexact?  type arity invert negate identity-like trace determinant
type-predicate)
```

The operators applicable for a single variable $'a$ are:

```
(discover:named-opers-for 'a)
;Value:  (one-like cos acos exp cosh imag-part conjugate zero-like
sinh sin asin angle magnitude inexact?  type arity real-part invert
negate identity-like sqrt log type-predicate atan1)
```

[1]See http://groups.csail.mit.edu/mac/users/gjs/6946/linux-install.htm

`scmutils` has special treatment for mathematical functions:

```
(discover:named-opers-for (compose sin cos))
;Value:  (one-like cos acos exp cosh imag-part zero-like abs sinh
sin asin angle magnitude inexact?  type arity real-part invert
negate identity-like sqrt log square type-predicate atan1)
```

Other Applications

The discovery system combined with the predicate satisfaction search makes a very useful general purpose tool. Even when using such a simple system as Scheme, it can be difficult to remember the particular names of simple procedures. For instance, you might have list of hundreds of numbers and want to turn it into a vector, or a 20 digit floating point number that you want to print as a string: are the desired operations `list->vector` and `number->string`? Are they generic `to-string` and `to-vector`? You could read the documentation or search by hand through the bound procedures in the environment, or you could do a search with the predicates `vector?` and `string?`.

With proper memoization and error catching, predicate satisfaction could be the basis for robust computing: if compiled code ever failed, the system could elegantly search for an alternative (perhaps much less efficient but still satisfactory) procedure from the system libraries or even among network peers in a network instead of halting computation and giving the user an error.

A Graphical User Interface

The graphical user interface (GUI) was implemented using a new foreign function interface (FFI) and GUI toolkit bindings for MIT/GNU Scheme written by Matt Birkholz[2].

The GUI essentially displays a object (a collection of arguments) and allows the user to select an applicable operator from a drop down box. The result of this operation is then itself displayed as an object and the user can recursively select an operation for *that* object. Figure 1 shows a screenshot of the interface with a chain of example operations.

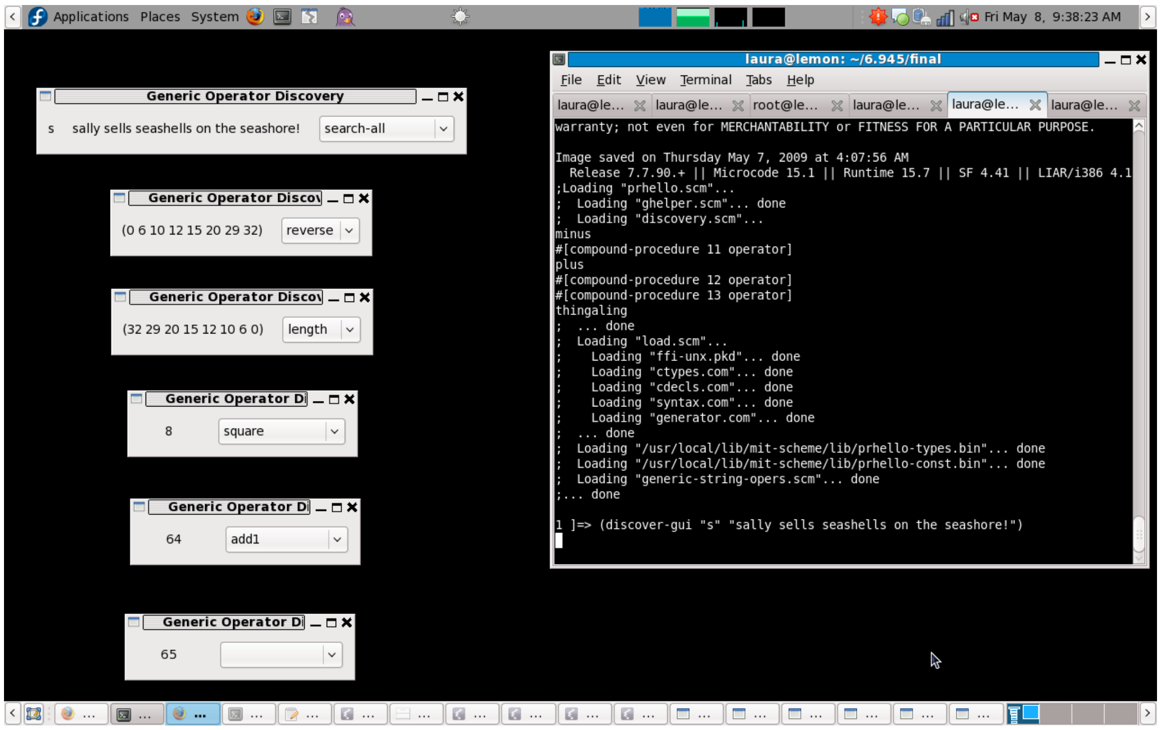
Procedures

Most of the scheme GUI code is included in the appendix as `prhello.scm`; additional declaration and shim files were used for compilation but are not included.

(discover:thunklist-for . args) This is a special purpose function (in discover.scm) which creates a data structure (a list) containing both the passed arguments and a series of delayed thunks: each thunk is the application of an appropriate "named" generic operator on the arguments. Each thunk has the operator's name symbol attached.

(discover-gui . args) This generates the actual GUI for the given arguments. `discover:thunklist-for` generates the set of possible operators which are displayed as a pull-down list for the user: selecting an operator evaluates the thunk and calls `discover-gui` on the result.

[2]See http://birkholz.chandler.az.us/~matt/Scheme/


Applications Places System
Fri May 8, 9:38:23 AM
Generic Operator Discovery
s
sally sells seashells on the seashore!
search-all
Generic Operator Discov
(0 6 10 12 15 20 29 32)
reverse
Generic Operator Discov
(32 29 20 15 12 10 6 0)
length
Generic Operator D
8
square
Generic Operator D
64
add1
Generic Operator D
65
laura@lemon: ~/6.945/final
File Edit View Terminal Tabs Help
laura@le… laura@le… root@le… laura@le… laura@le… laura@le…

```
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Image saved on Thursday May 7, 2009 at 4:07:56 AM
  Release 7.7.90.+ || Microcode 15.1 || Runtime 15.7 || SF 4.41 || LIAR/i386 4.1
;Loading "prhello.scm"...
;  Loading "ghelper.scm"... done
;  Loading "discovery.scm"...
minus
#[compound-procedure 11 operator]
plus
#[compound-procedure 12 operator]
#[compound-procedure 13 operator]
thingaling
;  ... done
;  Loading "load.scm"...
;    Loading "ffi-unx.pkd"... done
;    Loading "ctypes.com"... done
;    Loading "cdecls.com"... done
;    Loading "syntax.com"... done
;    Loading "generator.com"... done
;  ... done
;  Loading "/usr/local/lib/mit-scheme/lib/prhello-types.bin"... done
;  Loading "/usr/local/lib/mit-scheme/lib/prhello-const.bin"... done
;  Loading "generic-string-opers.scm"... done
;... done

1 ]=> (discover-gui "s" "sally sells seashells on the seashore!")
```

Figure 1: GUI Screenshot

Appendix: Code Listing

ghelper.scm

```
;;; From 6.945 Staff, with minor edit by bnewbold (May 2009):
;;; the optional name argument is handled in the style of
;;; the scmutils implementation
;;;;           Most General Generic-Operator Dispatch
(declare (usual-integrations))
;;; Generic-operator dispatch is implemented here by a discrimination
;;; list, where the arguments passed to the operator are examined by
;;; predicates that are supplied at the point of attachment of a
;;; handler (by ASSIGN-OPERATION).
;;; To be the correct branch all arguments must be accepted by
;;; the branch predicates, so this makes it necessary to
;;; backtrack to find another branch where the first argument
;;; is accepted if the second argument is rejected.  Here
;;; backtracking is implemented by OR.
(define (make-generic-operator arity default-operation #!optional name)
  (let ((record (make-operator-record arity)))
    (define (operator . arguments)
      (if (not (= (length arguments) arity))
          (error:wrong-number-of-arguments operator arity arguments))
      (let ((succeed
             (lambda (handler)
               (apply handler arguments))))
        (let per-arg
            ((tree (operator-record-tree record))
             (args arguments)
             (fail
              (lambda ()
                (error:no-applicable-methods operator arguments))))
          (let per-pred ((tree tree) (fail fail))
            (cond ((pair? tree)
                   (if ((caar tree) (car args))
                       (if (pair? (cdr args))
                           (per-arg (cdar tree)
                                    (cdr args)
                                    (lambda ()
                                      (per-pred (cdr tree) fail)))
                           (succeed (cdar tree)))
                       (per-pred (cdr tree) fail)))
                  ((null? tree)
                   (fail))
                  (else
                   (succeed tree)))))))
    (hash-table/put! *generic-operator-table* operator record)
    (if default-operation
        (assign-operation operator default-operation))
    (if (not (default-object? name))
        (hash-table/put! *generic-operator-table* name record))
    operator))
(define *generic-operator-table*
  (make-eq-hash-table))
(define (make-operator-record arity) (cons arity '()))
(define (operator-record-arity record) (car record))
(define (operator-record-tree record) (cdr record))
(define (set-operator-record-tree! record tree) (set-cdr! record tree))
(define (assign-operation operator handler . argument-predicates)
  (let ((record
         (let ((record (hash-table/get *generic-operator-table* operator #f))
               (arity (length argument-predicates)))
           (if record
               (begin
                 (if (not (<= arity (operator-record-arity record)))
                     (error "Incorrect operator arity:" operator))
                 record)
               (let ((record (make-operator-record arity)))
                 (hash-table/put! *generic-operator-table* operator record)
                 record)))))
    (set-operator-record-tree! record
                               (bind-in-tree argument-predicates
                                             handler
                                             (operator-record-tree record))))
  operator)
(define defhandler assign-operation)
(define (bind-in-tree keys handler tree)
  (let loop ((keys keys) (tree tree))
    (if (pair? keys)
        (let find-key ((tree* tree))
          (if (pair? tree*)
              (if (eq? (caar tree*) (car keys))
                  (begin
                    (set-cdr! (car tree*)
                              (loop (cdr keys) (cdar tree*)))
                    tree)
                  (find-key (cdr tree*)))
              (cons (cons (car keys)
                          (loop (cdr keys) '()))
                    tree)))
        (if (pair? tree)
            (let ((p (last-pair tree)))
              (if (not (null? (cdr p)))
                    (warn "Replacing a handler:" (cdr p) handler))
                (set-cdr! p handler)
                tree)
              (begin
                (if (not (null? tree))
                    (warn "Replacing top-level handler:" tree handler))
                handler)))))
```

discovery.scm

```
; discovery.scm
; author: bnewbold @ mit (with lch @ mit)
; for 6.945
; circa 04/2009
; For speed?
;(declare (usual-integrations))
; If it isn't already....
;(load "ghelper")
; takes two lists: the first is a set of predicates and the second a set
; of arguments; if any of the predicates are #t for the args, win, else fail
(define (for-any? preds args)
  (cond ((null? preds) #f)
        ((null? (car preds)) #f)
        ((apply (car preds) args) #t)
        (else (for-any? (cdr preds) args))))
; Test
(for-any? (list list? null? vector?) '(5))
; #f
(for-any? (list list? null? vector?) '('(1 2 3)))
; #t
; finds all the operators which can be applied to the args; returns a list
; of operators (not the actual procedures; will include duplicate symbols and
; operator stubs for named operators)
(define (discover:opers-for . args)
  (let* ((arity (length args))
         (opers (hash-table->alist *generic-operator-table*))
         (check
          (lambda (op)
            (if (not (eq? arity (cadr op)))
                #f
                (let per-arg ((tree (operator-record-tree (cdr op)))
                              (args args)
                              (fail (lambda () #f)))
                  (let per-pred ((tree tree) (fail fail))
                    (cond ((pair? tree)
                           (if ((caar tree) (car args))
                               (if (pair? (cdr args))
                                   (per-arg (cdar tree)
                                            (cdr args)
                                            (lambda ()
                                              (per-pred (cdr tree) fail)))
                                   #t)
                               (per-pred (cdr tree) fail)))
                          ((null? tree) (fail))
                          (else #t))))))))
    (map car (filter check opers))))
; same as the above but only grabs the symboled ones
(define (discover:named-opers-for . args)
  (filter symbol? (apply discover:opers-for args)))
; returns a list of
(define (discover:named-opers)
  (let ((check (lambda (x) (cond ((null? x) '())
                                 ((symbol? x) x)
                                 (else '()))))))
    (filter (lambda (x) (not (null? x)))
            (map check (hash-table-keys *generic-operator-table*)))))
; this is just what operators do
(define (discover:apply-name name . args)
  (let ((record (hash-table/get *generic-operator-table* name #f)))
    (let ((succeed
           (lambda (handler)
             (apply handler args))))
      (let per-arg
          ((tree (operator-record-tree record))
           (args args)
           (fail
            (lambda ()
              (error:no-applicable-methods operator args))))
        (let per-pred ((tree tree) (fail fail))
          (cond ((pair? tree)
                 (if ((caar tree) (car args))
                     (if (pair? (cdr args))
                         (per-arg (cdar tree)
                                  (cdr args)
                                  (lambda ()
                                    (per-pred (cdr tree) fail)))
                         (succeed (cdar tree)))
                     (per-pred (cdr tree) fail)))
                ((null? tree)
                 (fail))
                (else
                 (succeed tree))))))))
```

```
(define (discover:thunklist-for . args)
  (let ((names (apply discover:named-opers-for args)))
    (cons args
          (map (lambda (x)
                 (list x
                       (lambda ()
                         (apply discover:apply-name (cons x args)))))
               names))))
(define (discover:apply-all . args)
  (let ((names (apply discover:named-opers-for args)))
    (map (lambda (x)
           (apply discover:apply-name (cons x args)))
         names)))
(define (discover:apply-all-name . args)
  (let ((names (apply discover:named-opers-for args)))
    (map (lambda (x)
           (list (apply discover:apply-name (cons x args)) x))
         names)))
(define (discover:satisfy pred? . args)
  (let try ((objs (list args)))
    (let ((goodies (filter (lambda (x) (apply pred? x)) objs)))
      (if (not (null? goodies))
          (car goodies)
          (try (fold-right append
                           '()
                           (map (lambda (x)
                                  (map list
                                       (apply discover:apply-all x)))
                                objs)))))))
(define (discover:satisfy-sequence pred? . args)
  (let try ((objs (list (list args))))
    (let ((goodies (filter (lambda (x) (apply pred? (car x))) objs)))
      (if (not (null? goodies))
          goodies
          (try (fold-right append
                           '()
                           (map (lambda (x)
                                  (map (lambda (y)
                                         (cons (list (car y)) (cons (cadr y)
                                                                    (cdr x))))
                                       (apply discover:apply-all-name (car x))))
                                objs)))))))
; see discovery-examples.scm for testing and examples
```

discovery-examples.scm

```
(load "ghelper")
(load "discovery")
(define inverse
  (make-generic-operator 1 #f 'inverse))
(define plus
  (make-generic-operator 2 #f 'plus))
(define minus
  (make-generic-operator 2 #f 'minus))

(assign-operation inverse
          (lambda (x) (/ 1 x))
          (lambda (x) (and (number? x)
                   (not (integer? x)))))
; actually a transpose, but meh
(assign-operation inverse
          (lambda (x) (apply zip x))
          (lambda (x)
            (and (list? x)
             (for-all? x list?))))
(define any? (lambda (x) #t))
(assign-operation minus - any? any?)
(assign-operation plus + any? any?)
(plus 1 2)
; 3
;(minus 3)
; ERROR
(inverse 6.5)
;Value: .15384615384615385
(discover:opers-for 6.5)
;Value 52: (inverse #[compound-procedure 49 operator])
(discover:named-opers-for 6.5)
;Value 53: (inverse)
(discover:named-opers-for 1 2)
;Value 54: (plus minus)
(environment-lookup (the-environment) 'inverse)
;Value 49: #[compound-procedure 49 operator]
(hash-table/get *generic-operator-table* inverse #f)
;Value 59: (1 (#[compound-procedure 57] . #[compound-procedure 60]) (#[compound-procedure 61]
(hash-table/get *generic-operator-table* minus #f)
;Value 63: (2 (#[compound-procedure 56 any?] (#[compound-procedure 56 any?] . #[arity-dispatc
(hash-table-size *generic-operator-table*)
;Value: 6    ; for this file
;Value: 92   ; for scmutils
;this prints all keys line by line
(for-each
 (lambda (x) (newline)
     (display x))
 (hash-table/key-list *generic-operator-table*))
(define add1 (make-generic-operator 1 #f 'add1))
(define sub1 (make-generic-operator 1 #f 'sub1))
(define double (make-generic-operator 1 #f 'double))
(define square (make-generic-operator 1 #f 'square))
(define inverse (make-generic-operator 1 #f 'inverse))
(defhandler add1 (lambda (x) (+ x 1)) number?)
(defhandler sub1 (lambda (x) (- x 1)) number?)
(defhandler double (lambda (x) (* 2 x)) number?)
(defhandler square (lambda (x) (* x x)) number?)
(defhandler inverse (lambda (x) (/ 1 x)) (lambda (n)
                       (and (number? n)
                        (not (zero? n)))))
(discover:apply-all 3)
;Value 89: (1/3 4 9 2 6)
(discover:satisfy (lambda (x) (eq? x 9)) (/ 1 2))
;Value 35: (9)
(discover:satisfy-sequence (lambda (x) (eq? x 9)) (/ 1 2))
;Value 36: (((9) square double add1) ((9) square add1 inverse))
(discover:satisfy-sequence (lambda (x) (eq? x 49)) (/ 5 6))
;Value 37: (((49) square sub1 inverse sub1))
(define (prime? n)
  (cond ((null? n) #f)
        ((not (integer? n)) #f)
        ((> 0 n) #f)
        (else (let lp ((m 2))
                (cond ((> m (sqrt n)) #t)
                      ((integer? (/ n m)) #f)
                      (else (lp (+ m 1))))))))
(prime? 47)
; #t
(discover:satisfy-sequence prime? (/ 5 6))
;Value 39: (((5) inverse sub1 inverse))
(discover:satisfy-sequence prime? 923)
;Value 44: (((1847) add1 double))
(discover:named-opers)
```

prhello.scm

```
(declare (usual-integrations))
(load "ghelper.scm")
(load "discovery.scm")
(load-option 'FFI)
(C-include "prhello")
```

```
(load "generic-string-opers.scm") ; extra generic string operations
(define get-vals car)
(define (get-proc-symbols input) (map car (cdr input)))
(define (apply-ith-thunk input i) ((cadr (list-ref (cdr input) i))))
(define (thing-to-string thing)
 (let ((buff (open-output-string)))
   (display thing buff)
   (get-output-string buff)))
(define (discover-gui . input)
  (C-call "gtk_init" 0 null-alien)
  (let* ((discovered-opers (apply discover:thunklist-for input))
         (window (let ((alien (make-alien '|GtkWidget|)))
                   (C-call "gtk_window_new" alien
                           (C-enum "GTK_WINDOW_TOPLEVEL"))
                   (if (alien-null? alien) (error "Could not create window."))
                       alien))
         (hbox (let ((alien (make-alien '|GtkWidget|)))
                 (C-call "gtk_hbox_new" alien 0 20)
                 (if (alien-null? alien) (error "Could not create hbox."))
                 alien))
         (combo (let ((alien (make-alien '|GtkWidget|)))
                  (C-call "gtk_combo_box_new_text" alien)
                  (if (alien-null? alien) (error "Could not create combo."))
                  alien))
         (labels (map (lambda (val)
                        (let ((alien (make-alien '|GtkWidget|)))
                          (C-call "gtk_label_new" alien (thing-to-string val))
                          (if (alien-null? alien) (error "Could not create label."))
                          alien))
                      (get-vals discovered-opers))))
    (for-each (lambda (proc-symbol)
                (C-call "gtk_combo_box_append_text" combo (thing-to-string proc-symbol)))
              (get-proc-symbols discovered-opers))

    (for-each (lambda (label) (C-call "gtk_container_add" hbox label)) labels)
    (C-call "gtk_container_add" hbox combo)
    (C-call "gtk_container_add" window hbox)
    (C-call "gtk_window_set_title" window "Generic Operator Discovery")
    (C-call "gtk_container_set_border_width" window 10)
    (C-call "gtk_window_resize" window 250 20)
    (C-call "g_signal_connect" combo "changed"
            (C-callback "changed")      ;trampoline
            (C-callback                 ;callback ID
             (lambda (w)
               (let ((i (C-call "gtk_combo_box_get_active" combo)))
                 (discover-gui (apply-ith-thunk discovered-opers i))))))
(C-call "g_signal_connect" window "delete_event"
        (C-callback "delete_event") ;trampoline
        (C-callback                 ;callback ID
         (lambda (w e)
           (begin
             (C-call "gtk_main_quit")
             0))))
(C-call "gtk_widget_show_all" window)
(C-call "gtk_main")
window))
```