Cactus

Write Yourself a Haskell... in Lisp

17 February 2013 (programming haskell lisp) (3 comments)

For me, the best way to understand something is usually to implement it myself. So when I first started getting serious about Haskell, I implemented (in Common Lisp) a simple lazy functional programming language that used graph rewriting for the interpretation. Then, years later, I mentioned this implementation in a Reddit thread about writing a Lisp interpreter in Haskell (maybe inspired by the well-known Haskell tutorial and this blogpost's namesake). I now decided to clean up that old code, throw out all the unnecessary cruft, and arrive at something easy to understand. The finished interpreter is less than 500 lines of Common Lisp (not including the type checker which will be the subject of my next post).

I should also add that of course we're not going to actually implement Haskell here. As we'll see, the language is a very much simplified model of pure, lazy functional languages. It's actually closer to GHC's Core than Haskell.

Syntax

We're not going to bother with parsing: our input programs will be S-expressions. For example, instead of the following Haskell program:

```
data List a = Nil | Cons a (List a)

map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

we will write:

```
(defdata list (a) nil (cons a (list a)))

(deffun map
   ((f nil) nil)
   ((f (cons x xs)) (cons (f x) (map f xs))))
```

As you can see, there is no syntactic distinction between variable names and constructor names — when translating these S-expressions, we'll take special care to always register constructor names before processing expressions.

Internally, we'll represent programs in an even more simplified way, by making all function applications explicit. Multi-parameter functions and constructors, of course, will be implemented via schönfinkeling. The syntax tree itself is represented using two class hierarchies: subclasses of `expr` for expressions, and subclasses of `pattern` for patterns:

```
(defclass expr ()
  ()
  (:documentation "Expression"))

(defclass symbol-expr (expr)
  ((symbol :initarg :symbol :reader expr-symbol)))

(defclass var-expr (symbol-expr)
  ()
  (:documentation "Variable occurance"))

(defclass cons-expr (symbol-expr)
  ()
  (:documentation "Constructor"))

(defclass apply-expr (expr)
  ((fun :initarg :fun :reader expr-fun)
   (arg :initarg :arg :reader expr-arg))
  (:documentation "Function application"))

(defclass let-expr (expr)
  ((bindings :initarg :bindings :reader expr-bindings)
   (body :initarg :body :reader expr-body))
  (:documentation "Let"))

(defclass lambda-expr (expr)
  ((formals :initarg :formals :reader expr-formals)
   (body :initarg :body :reader expr-body))
  (:documentation "Lambda abstraction"))
```

Contents

- Open Source
- Blog
- ELTE CS
- CV
- Photos

Blog tags

- advogato
- agda
- android
- brainfuck
- correctness
- electronics
- ELTE
- movies
- FPGA
- gadget
- haskell
- iphone
- ISC
- games
- java
- food
- exhibition
- books
- lisp
- math
- meta
- miata
- language
- programming
- retro
- SCB
- Singapore
- personal
- sziget
- theater
- Tilos
- titanic
- history
- unix
- trip
- VPG
- windows
- XML
- zene

```lisp
(defclass pattern ()
  ()
  (:documentation "Pattern"))

(defclass wildcard-pattern (pattern)
  ()
  (:documentation "Wildcard pattern"))

(defclass var-pattern (pattern)
  ((symbol :initarg :symbol :reader pattern-symbol))
  (:documentation "Variable binder"))

(defclass cons-pattern (pattern)
  ((cons :initarg :cons :reader pattern-cons)
   (args :initarg :args :initform nil :reader pattern-args))
  (:documentation "Constructor pattern"))
```

syntax-defs.lisp

Going from S-expressions to this object-oriented representation is straightforward:

```lisp
(defun normalize-pat (pat)
  (typecase pat
    (atom
     (cond ((eq pat :wildcard) (make-instance 'wildcard-pattern))
           ((constructorp pat) (make-instance 'cons-pattern :cons pat))
           (t                  (make-instance 'var-pattern :symbol pat))))
    (cons
     (make-instance 'cons-pattern
                    :cons (first pat)
                    :args (mapcar #'normalize-pat (rest pat))))))

(defun normalize-expr (expr)
  (typecase expr
    (atom
     (make-instance (if (constructorp expr) 'cons-expr 'var-expr) :symbol expr))
    (cons
     (case (first expr)
       ((let)
        (destructuring-bind ((&rest bindings) body) (rest expr)
          (make-instance 'let-expr
                         :bindings (loop for (name val) in bindings
                                         collect (cons name (normalize-expr val)))
                         :body (normalize-expr body))))
       ((lambda)
        (destructuring-bind ((&rest patterns) body) (rest expr)
          (make-instance 'lambda-expr
                         :formals (mapcar #'normalize-pat patterns)
                         :body (normalize-expr body))))
       (otherwise
        (reduce (lambda (f x) (make-instance 'apply-expr :fun f :arg x))
                (mapcar #'normalize-expr expr)))))))
```

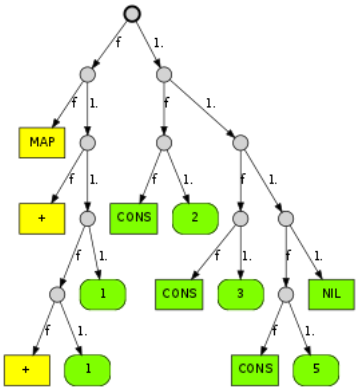
syntax.lisp

Graph rewriting semantics

The basic idea behind implementing a lazy functional language using graph rewriting is to represent terms as directed graphs, and reducing a function application is implemented by replacing the application node with the graph corresponding to the right-hand side of the function definition, of course with all the references to formal arguments instantiated to the actual arguments.

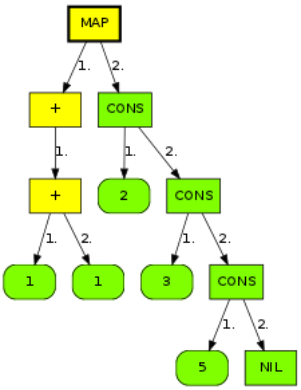
Let's look at a simple example first, with no sharing or recursion:

```lisp
(defvar x
  (let ((primes (cons 2 (cons 3 (cons 5 nil)))))
    (map (+ (+ 1 1) primes))))
```

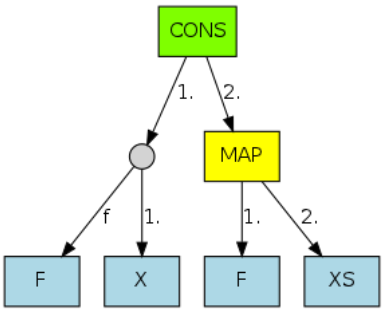


As you can see, the green boxes represent constructors (with rounded corners for primitive values), yellow ones are functions, and the small gray circles are function applications. There are no variables, since references are resolved when building this graph.

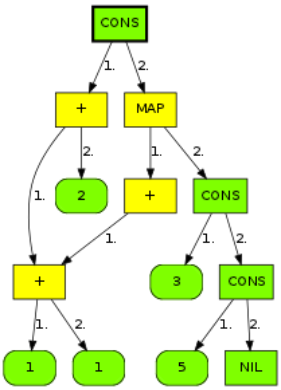
We can simplify this format by omitting application nodes and just pushing arguments under their function node (until it becomes saturated), like this:



Reducing this entails instantiating the following graph (from the second case of `map`), with `f` bound to `(+ (+ 1 1)`, `x` bound to `2`, and `xs` bound to `(cons 3 (cons 5 nil))`. Note the explicit application node: we won't know the arity of `f` until it's bound to something.



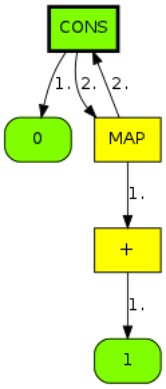
Resulting in



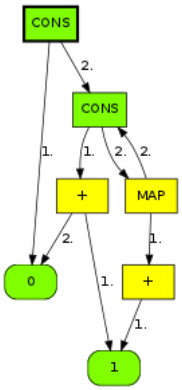
Note how the first argument to + is shared in the result. Also note how `(+ (+ 1 1) 2)` is not reduced by `map`, since it is lazy in its first argument.

Our second example shows recursively-defined terms:

```
(defvar nats (cons 0 (map (+ 1) nats)))
```



This is already in **WHNF**, but we can nevertheless reduce the tail of the list, to get this:



and so on.

Representing the graphs

Since we want to be able to replace function application subgraphs easily (when reducing those applications), the representation uses an extra level of indirection: `gnode`s contain the payload: the type of the node and pointers to its children `gref`s; and each `gref` contains a single reference to its content `gnode`. `gref`s can be shared, so when its content `gnode` is replaced, all the shared references are updated.

```
(defclass dotable ()
  ((dot-name :initform (gensym "V") :reader dot-name))
  (:documentation "Something that can be turned into a GraphViz node"))

(defclass gnode (dotable)
  ()
  (:documentation "Node in the graph representation"))

(defclass gref (dotable)
  ((node :initarg :node :accessor gderef))
  (:documentation "Pointer in the graph representation"))

(defun make-gref (gnode)
  (make-instance 'gref :node gnode))

(defclass cons-gnode (gnode)
  ((cons :initarg :cons :reader gnode-cons)
   (args :initarg :args :initform nil :accessor gnode-args)))

(defclass param-gnode (gnode)
  ((var :initarg :var :reader gnode-var)))

(defclass apply-gnode (gnode)
  ((fun :initarg :fun :accessor gnode-fun)
   (args :initarg :args :accessor gnode-args)))

(defclass fun-gnode (gnode)
  ((fun-name :initarg :fun-name :reader gnode-fun-name)
   (arity :initarg :arity :reader gnode-fun-arity)
```

```
    (args :initarg :args :initform (list) :accessor gnode-args)))
```

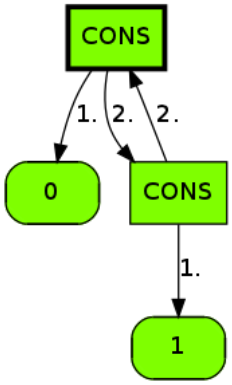
graph-defs.lisp

Variable nodes

Earlier, we said variables are not present as such in the graph representation, since they are inlined (respecting sharing, of course). But we still have a `var-gnode` class defined above. The reason for that is simply to mark occurances of formal variables in function definitions, which will be filled in when reducing function applications.

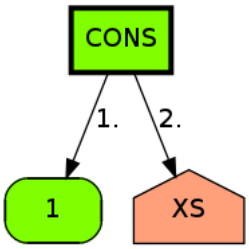
But there's another, more compilcated problem with variables. Our language's let construct is mutually recursive, so we can't just build up the variables' graphs one by one:

```
(defvar main
  (let ((zig (cons 0 zag))
        (zag (cons 1 zig)))
    zig))
```



Of course, not everything can be helped by this:

```
(defvar silly-list
  (let ((xs xs))
    (cons 1 xs)))
```



So we'll add a `gnode` subclass for temporarily storing let-bound variable occurances, and replace them after the fact. Unfortunately, this also causes the code that translates from `expr` to `gnode` to become a little spaghetti-y. On the other hand, we don't implement lambda lifting, as it's not strictly essential — the programmer will have to do it by hand, by writing top-level functions instead.

```
(defgeneric graph-from-expr (expr))

(defmethod graph-from-expr ((expr var-expr))
  (let* ((var-name (expr-symbol expr))
         (var-data (cdr (assoc var-name *vars*))))
    (if var-data (graph-from-var var-name var-data)
        (make-gref
         (or (let ((fun (lookup-function var-name)))
               (and fun (make-instance 'fun-gnode :fun-name var-name :arity (function-arity fun))))
             (make-instance 'param-gnode :var var-name))))))

(defmethod graph-from-expr ((expr cons-expr))
  (make-gref (make-instance 'cons-gnode :cons (expr-symbol expr))))

(defmethod graph-from-expr ((expr apply-expr))
  (make-gref
   (let ((fun (graph-from-expr (expr-fun expr)))
         (arg (graph-from-expr (expr-arg expr) )))
     (make-instance 'apply-gnode :fun fun :args (list arg)))))
```

```
(defmethod graph-from-expr ((expr let-expr))
  (let ((*vars* *vars*))
    (add-vars (expr-bindings expr))
    (fill-var-gref (graph-from-expr (expr-body expr)))))

(defmethod graph-from-expr ((expr lambda-expr))
  (error "Not implemented: lambdas"))

(defvar *vars-built*)

(defclass bottom-gnode (gnode)
  ((var :initarg :var :reader gnode-var)
   (ptr :initarg :ptr :reader gnode-var-ptr)))

(defun graph-from-var (var-name var-data)
  (or (car (second var-data))
      (let* ((gref (make-instance 'gref))
             (gref-ptr (list gref)))
        (setf (gderef gref) (make-instance 'bottom-gnode :var var-name :ptr gref-ptr))
        (setf (second var-data) gref-ptr)
        (let ((gref* (graph-from-expr (first var-data))))
          (rplaca gref-ptr gref*)
          gref*))))

(defvar *filled*)

(defun fill-var-gref (gref)
  (let ((*filled* (list)))
    (fill-var-gref* gref)))

(defgeneric fill-var-gref** (gref gnode))

(defmethod fill-var-gref** (gref (gnode bottom-gnode))
  (let ((gref* (car (gnode-var-ptr gnode))))
    (fill-var-gref* gref*)))

(defmethod fill-var-gref** (gref gnode)
  (fill-var-gnode (gderef gref))
  gref)

(defun fill-var-gref* (gref)
  (or (cdr (assoc gref *filled*))
      (progn
        (push (cons gref gref) *filled*)
        (let ((gref* (fill-var-gref** gref (gderef gref))))
          (push (cons gref gref*) *filled*)
          gref*))))

(defgeneric fill-var-gnode (gnode))

(defmethod fill-var-gnode ((gnode gnode))
  (values))

(defmethod fill-var-gnode ((gnode cons-gnode))
  (setf (gnode-args gnode) (mapcar #'fill-var-gref* (gnode-args gnode)))
  (values))

(defmethod fill-var-gnode ((gnode apply-gnode))
  (setf (gnode-fun gnode) (fill-var-gref* (gnode-fun gnode)))
  (setf (gnode-args gnode) (mapcar #'fill-var-gref* (gnode-args gnode)))
  (values))
```

graph.lisp

Making it tick

There are three parts to making reductions work: first, a way to do pattern matching against function alternatives. Second, given a mapping from this match, instantiating a function definition by replacing `var-gnode`s. The third part is orchestrating all of this by taking care of choosing which nodes to reduce.

Pattern matching is a relatively straightforward matter: a variable pattern always succeeds (and binds the subgraph) and a constructor pattern can either succeed and recurse, fail, or (if the actual node is a function application) force reduction. This latter is done via raising a Lisp **condition**.

```
(define-condition no-match ()
  ())
```

```
(define-condition need-reduce ()
  ((gref :initarg :gref :reader need-reduce-gref))
  (:report (lambda (condition stream)
             (format stream "Need to reduce ~A" (need-reduce-gref condition)))))

(defgeneric match-pattern* (pat gnode gref))

(defun match-pattern (pat gref)
  (match-pattern* pat (gderef gref) gref))

(defmethod match-pattern* ((pat wildcard-pattern) gnode gref)
  (list))

(defmethod match-pattern* ((pat var-pattern) gnode gref)
  (list (cons (pattern-symbol pat) gref)))

(defmethod match-pattern* ((pat cons-pattern) (gnode cons-gnode) gref)
  (unless (eq (pattern-cons pat) (gnode-cons gnode))
    (error 'no-match))
  ;; Because of well-typedness, there is no need to check arity here
  (mapcan #'match-pattern (pattern-args pat) (gnode-args gnode)))

(defmethod match-pattern* ((pat cons-pattern) gnode gref)
  (error 'need-reduce :gref gref))
```

match.lisp

Once we have the bindings in the format returned by `match-pattern`, we can easily instantiate function bodies, we just have to maintain a mapping from old node to new to avoid diverging on cycles.

```
(defgeneric deepclone-gnode (gnode vars mapping))

(defmethod deepclone-gnode ((gnode cons-gnode) vars mapping)
  (make-instance 'cons-gnode
                 :cons (gnode-cons gnode)
                 :args (loop for arg in (gnode-args gnode)
                             collect (deepclone-gref arg vars mapping))))

(defmethod deepclone-gnode ((gnode param-gnode) vars mapping)
  gnode)

(defmethod deepclone-gnode ((gnode apply-gnode) vars mapping)
  (make-instance 'apply-gnode
                 :fun (deepclone-gref (gnode-fun gnode) vars mapping)
                 :args (loop for arg in (gnode-args gnode)
                             collect (deepclone-gref arg vars mapping))))

(defmethod deepclone-gnode ((gnode fun-gnode) vars mapping)
  (make-instance 'fun-gnode
                 :fun-name (gnode-fun-name gnode)
                 :arity (gnode-fun-arity gnode)
                 :args (loop for arg in (gnode-args gnode)
                             collect (deepclone-gref arg vars mapping))))

(defun deepclone-gref (gref &optional vars (mapping (make-hash-table)))
  (or
   (and (typep (gderef gref) 'param-gnode)
        (cdr (assoc (gnode-var (gderef gref)) vars)))
   (gethash gref mapping)
   (let ((gref/copy (make-instance 'gref)))
     (setf (gethash gref mapping) gref/copy)
     (setf (gderef gref/copy) (deepclone-gnode (gderef gref) vars mapping))
     gref/copy)))
```

deepclone.lisp

Actual reduction then becomes just a matter of putting these two modules together. Several functions are provided with varying granularity: `reduce-graph` tries direct reduction, `reduce-graph*` catches `need-reduce` conditions and recurses on those nodes, in effect making sure a single reduction step at the target site can be made; and `reduce-to-whnf` repeatedly uses `reduce-graph*` until the head is either a constructor, or a non-saturated function application. `simplify-apps` is not really a reduction step, it just removes superfluous `apply-gnode`s.

```
(defgeneric gnode-force-cons (gnode gref))

(defmethod gnode-force-cons ((gnode cons-gnode) gref)
  (gnode-cons gnode))

(defmethod gnode-force-cons ((gnode gnode) gref)
```

```
    (error 'need-reduce :gref gref))

(defgeneric reduce-function (fun-info args))

(defmethod reduce-function ((fun-info prim-function-info) arg-grefs)
  (let ((args (loop for arg-gref in arg-grefs
                    collect (gnode-force-cons (gderef arg-gref) arg-gref))))
    (make-instance 'cons-gnode
                   :cons (apply (prim-function fun-info) args))))

(defmethod reduce-function ((fun-info match-function-info) args)
  (loop for (pats . body) in (function-matches fun-info)
        do (handler-case (return (gderef (deepclone-gref body (mapcan #'match-pattern pats args))))
             (no-match ()))
        finally (error 'no-match)))

(defgeneric reduce-graph-node (gnode))

(defun reduce-graph (gref)
  (let ((new-gnode (reduce-graph-node (gderef gref))))
    (when new-gnode
      (setf (gderef gref) new-gnode)
      t)))

(defun reduce-graph* (gref)
  (prog1
      (handler-case (reduce-graph gref)
        (need-reduce (req)
          (when (reduce-graph* (need-reduce-gref req))
            (reduce-graph* gref))))
    (simplify-apps gref)))

(defun reduce-to-whnf (gref)
  (when (reduce-graph* gref)
    (reduce-to-whnf gref)))

(defmethod reduce-graph-node ((gnode cons-gnode))
  nil)

(defmethod reduce-graph-node ((gnode bottom-gnode))
  nil)

(defmethod reduce-graph-node ((gnode param-gnode))
  (error "Internal error: parameter reference in head"))

(defun split-at (lst i)
  (labels ((aux (lst i before)
             (cond
               ((null lst) (values (reverse before) nil (= i 0)))
               ((= i 0)    (values (reverse before) lst t))
               (t          (aux (cdr lst) (1- i) (cons (car lst) before))))))
    (aux lst i nil)))

(defmethod reduce-graph-node ((fun fun-gnode))
  (multiple-value-bind (actuals remaining saturated) (split-at (gnode-args fun) (gnode-fun-arity fun))
    (when saturated
      (let* ((fun-info (lookup-function (gnode-fun-name fun)))
             (result-gnode (reduce-function fun-info actuals)))
        (if (null remaining) result-gnode
            (make-instance 'apply-gnode :fun (make-gref result-gnode) :args remaining))))))

(defgeneric gnode-add-args (gnode args))

(defmethod gnode-add-args ((gnode gnode) args)
  nil)

(defmethod gnode-add-args ((gnode cons-gnode) args)
  (make-instance 'cons-gnode
                 :cons (gnode-cons gnode)
                 :args (append (gnode-args gnode) args)))

(defmethod gnode-add-args ((gnode fun-gnode) args)
  (make-instance 'fun-gnode
                 :fun-name (gnode-fun-name gnode)
                 :arity (gnode-fun-arity gnode)
                 :args (append (gnode-args gnode) args)))
```

```
(defmethod gnode-add-args ((gnode apply-gnode) args)
  (make-instance 'apply-gnode
                 :fun (gnode-fun gnode)
                 :args (append (gnode-args gnode) args)))

(defmethod reduce-graph-node ((app apply-gnode))
  (let ((fun (gnode-fun app)))
    (or (gnode-add-args (gderef fun) (gnode-args app))
        (error 'need-reduce :gref fun))))

(defgeneric simplify-app (gnode))

(defmethod simplify-app ((gnode gnode))
  nil)

(defmethod simplify-app ((app apply-gnode))
  (mapcar #'simplify-app-gref (cons (gnode-fun app) (gnode-args app)))
  (gnode-add-args (gderef (gnode-fun app)) (gnode-args app)))

(defvar *grefs*)

(defun simplify-app-gref (gref)
  (unless (gethash gref *grefs*)
    (setf (gethash gref *grefs*) t)
    (let ((new-gnode (simplify-app (gderef gref))))
      (when new-gnode
        (setf (gderef gref) new-gnode)))))

(defun simplify-apps (gref)
  (let ((*grefs* (make-hash-table)))
    (simplify-app-gref gref)))
```

reduce.lisp

Housekeeping

The rest of the code just keeps a registry of functions and constructors, and defines some primitive functions (note how we need the very simple `bool` ADT to be a built-in just so we have a return type for >=).

```
(defvar *constructors*)
(defvar *functions*)
(defvar *vars*)

(defmacro in-fresh-context (&body body)
  `(let ((*constructors* (make-hash-table))
         (*functions* (make-hash-table))
         (*vars* (list)))
     (register-builtin-types)
     (register-prim-functions)
     ,@body))

(defun add-constructors (constructors)
  (loop for (cons-name . cons-type) in constructors
        do (setf (gethash cons-name *constructors*) cons-type)))

(defun add-vars (vars)
  (setf *vars*
        (append
         (loop for (var-name . var-expr) in vars
               collect (list var-name var-expr nil))
         *vars*)))

(defun constructorp (x)
  (typecase x
    (integer t)
    (string t)
    (atom
     (nth-value 1 (gethash x *constructors*)))))

(defclass function-info ()
  ())

(defclass proto-function-info (function-info)
  ((arity :initarg :arity :reader function-arity)))

(defclass prim-function-info (proto-function-info)
  ((fun :initarg :fun :reader prim-function)))
```

```lisp
(defclass match-function-info (function-info)
  ((matches :initarg :matches :reader function-matches)))

(defmethod function-arity ((function-info match-function-info))
  (length (caar (function-matches function-info))))

(defun register-proto-function (fun-name arity)
  (setf (gethash fun-name *functions*)
        (make-instance 'proto-function-info :arity arity)))

(defun register-match-function (fun-name matches)
  (setf (gethash fun-name *functions*)
        (make-instance 'match-function-info :matches matches)))

(defun register-prim-function (fun-name arity fun)
  (setf (gethash fun-name *functions*)
        (make-instance 'prim-function-info :arity arity :fun fun)))

(defun lookup-function (fun-name)
  (format nil "~A" *functions*)
  (gethash fun-name *functions*))
```

registry.lisp

```lisp
(defun register-builtin-types ()
  (add-constructors '((true . bool)
                      (false . bool))))

(defun register-prim-functions ()
  (labels
      ((pred (p)
         (lambda (&rest args)
           (if (apply p args) 'true 'false))))
   (register-prim-function '>= 2 (pred #'>=))
   (register-prim-function '+ 2 #'+)
   (register-prim-function '- 2 #'-)
   (register-prim-function '* 2 #'*)
   (register-prim-function '/ 2 #'/)
   (register-prim-function 'show-int 1 (lambda (x) (format nil "~D" x)))
   (register-prim-function 'string-append 2 (lambda (x y) (format nil "~A~A" x y)))))
```

prim.lisp

Putting it all together

Given an S-expression containing an Alef program, we can transform it into a graph suitable for reduction by doing the following steps:

1. Split into datatype definitions, variable definitions and function definitions
2. Process datatype definitions by registering the constructor names (since we don't do typechecking yet, the only information needed about constructors is that they are not function calls/variable references)
3. Register top-level variable names
4. Register function names
5. Process top-level variable definitions into graphs
6. Process function definitions into graph templates
7. Return the graph of the top-level variable called `main` as the actual program

```lisp
(defun partition-by (f xs)
  (let ((table (make-hash-table)))
    (loop for x in xs
          for key = (funcall f x)
          do (push x (gethash key table)))
    (maphash (lambda (key group) (setf (gethash key table) (reverse group))) table)
    table))

(defun parse-program (program)
  (let ((program-parts (partition-by #'first program)))

    (add-constructors
     (loop for (defdata type-name type-args . constructors) in (gethash 'defdata program-parts)
           append (loop for (cons-name . cons-args) in constructors
                        collect (cons cons-name type-name))))

    (add-vars
     (loop for (defvar var-name var-body) in (gethash 'defvar program-parts)
           collect (cons var-name (normalize-expr var-body))))
```

```
      (let ((fun-defs
             (loop for (deffun name . matches) in (gethash 'deffun program-parts)
                   do (register-proto-function name (length (caar matches)))
                   collect (cons name (loop for (pats expr) in matches
                                            collect (cons (mapcar #'normalize-pat pats) (normalize-expr expr))))))

        ;; Build graphs for functions
        (loop for (fun-name . matches) in fun-defs
              for match-nodes = (loop for (pats . expr) in matches
                                      collect (cons pats (fill-var-gref (graph-from-expr expr))))
              do (register-match-function fun-name match-nodes)))

      (fill-var-gref (graph-from-var 'main (cdr (assoc 'main *vars*))))))
```

program.lisp

Visualization

The nice plots in this blogpost were created by dumping the program's graph in **GraphViz** format, and running **dot** on the output. The visualization code is a straightforward traversal of the graph. Note how we store a stable name (generated using `gensym`) in each node, to ensure a correspondence in the generated GraphViz nodes between reduction steps. In the future, this could be used to e.g. animate the graph to show each reduction step.

```
(defvar *dot-mapping*)
(defvar *dot-stream*)

(defparameter *visualize-grefs* nil)

(defgeneric to-dot (x rootp))

(defmethod to-dot ((gref gref) rootp)
  (or (gethash gref *dot-mapping*)
      (let ((dot (dot-name (if *visualize-grefs* gref (gderef gref)))))
        (setf (gethash gref *dot-mapping*) dot)
        (to-dot (gderef gref) rootp)
        (when *visualize-grefs*
          (dot-node (dot-name gref) "" "shape=point")
          (dot-edge (dot-name gref) (dot-name (gderef gref)) ""))
        dot)))

(defun format-symbol (x)
  (if (symbolp x) (symbol-name x)
      (format nil "~S" x)))

(defconstant +dot-escape-chars+ "\"")

(defun dot-escape (str)
  (labels ((escape-p (char)
             (find char +dot-escape-chars+))

           (escape-char (char)
             (format nil "\\~A" char)))
    (with-output-to-string (s)
      (loop for start = 0 then (1+ pos)
            for pos = (position-if #'escape-p str :start start)
            do (write-sequence str s :start start :end pos)
            when pos do (write-sequence (escape-char (char str pos)) s)
            while pos))))

(defun dot-node (dot label rootp &optional style)
  (format *dot-stream* "~&~A [label=\"~A\" ~A ~A]"
          dot
          (dot-escape label)
          (if rootp "penwidth=3" "")
          (if style (format nil ", ~A" style) "")))

(defun dot-edge (dot/from dot/to label)
  (format *dot-stream* "~&~A -> ~A[label=\"~A\"]" dot/from dot/to (dot-escape label)))

(defmethod to-dot ((gnode bottom-gnode) rootp)
  (dot-node (dot-name gnode) (format-symbol (gnode-var gnode)) rootp "shape=house, fillcolor=lightsalmon")
  ;; (dot-node dot "" "shape=house, fillcolor=lightsalmon")
  ;; (dot-node dot "" "shape=point")
  ;; (dot-edge dot dot "")
  )

(defmethod to-dot ((gnode cons-gnode) rootp)
```

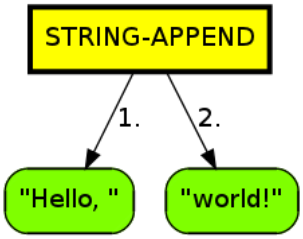
```
    (let* ((prim (typecase (gnode-cons gnode)
                   (integer t)
                   (string t)))
           (style (if prim "\"filled, rounded\"" "filled"))
           (color "chartreuse")
           (style (format nil "shape=box, style=~A, fillcolor=~A" style color)))
      (dot-node (dot-name gnode) (format-symbol (gnode-cons gnode)) rootp style))
    (loop for arg-dot in (mapcar #'(lambda (x) (to-dot x nil)) (gnode-args gnode))
          for index = 1 then (1+ index)
          do (dot-edge (dot-name gnode) arg-dot (format nil "~D." index))))

  (defmethod to-dot ((gnode param-gnode) rootp)
    (dot-node (dot-name gnode) (format-symbol (gnode-var gnode)) rootp "shape=box, fillcolor=lightblue"))

  (defmethod to-dot ((gnode apply-gnode) rootp)
    (dot-node (dot-name gnode) "" rootp "shape=circle, ordering=out, fixedsize=true, width=.25")
    (dot-edge (dot-name gnode) (to-dot (gnode-fun gnode) nil) "f")
    (loop for arg-dot in (mapcar #'(lambda (x) (to-dot x nil)) (gnode-args gnode))
          for index = 1 then (1+ index)
          do (dot-edge (dot-name gnode) arg-dot (format nil "~D." index))))

  (defmethod to-dot ((gnode fun-gnode) rootp)
    (dot-node (dot-name gnode) (format-symbol (gnode-fun-name gnode)) rootp "shape=box, fillcolor=yellow")
    (loop for arg-dot in (mapcar #'(lambda (x) (to-dot x nil)) (gnode-args gnode))
          for index = 1 then (1+ index)
          do (dot-edge (dot-name gnode) arg-dot (format nil "~D." index))))

  (defun dot-from-graph (gref &optional (stream *standard-output*))
    (let ((*dot-mapping* (make-hash-table))
          (*dot-stream* stream))
      (format *dot-stream* "digraph G{")
      (format *dot-stream* "~&graph[size=\"3,3\", bgcolor=\"transparent\"]")
      (format *dot-stream* "~&node[style=filled]")
      (to-dot gref t)
      (format *dot-stream* "~&}")))
```

visualize.lisp

We can use this like so:

```
  (in-fresh-context
    (let ((g (parse-program '((defvar main (string-append "Hello, " "world!"))))))
      (simplify-apps g)
      (with-open-file (s "hello.dot" :direction :output :if-exists :supersede)
        (dot-from-graph g s))
      (reduce-to-whnf g)
      (with-open-file (s "hello-reduced.dot" :direction :output :if-exists :supersede)
        (dot-from-graph g s))))
```

Which results in the very uninteresting graphs:



and

In conclusion

SLOCCount tells me the whole code presented in this blogpost is 471 lines of Common Lisp; you can check out the full source code on GitHub. It implements lazy semantics for a pure functional programming language with pattern matching on algebraic datatypes; of course, by changing the definition of `reduce-function`, we could easily make it strict.

Next time

In my next blog post, I'll be adding **Hindley-Milner** type inference / typechecking. Because of the **type-erasure semantics** of our language, we could implement our evaluator without any type system implementation, simply by assuming the input program to be well-typed. So all that we'll need is an extra typechecking step between parsing and graph building that either rejects or accepts an Alef program.

« Simply Typed Lambda Calculus in Agda, Without Shortcuts **All entries** **A Brainfuck CPU in FPGA »**

Theo 2013-02-17 17:01:15

Missing a close paren:

(defvar x

(let ((primes (cons 2 (cons 3 (cons 5 nil)))))

(map (+ (+ 1 1) **)** primes)))

BMeph (**http://impandins.blogspot.com**) 2013-02-18 18:43:09

Sounds more as-if this should have been titled "Write yourself a Clean...in Lisp," although Haskell is getting all of the press coverage.

cactus 2013-02-19 12:48:47

Theo: Thanks, fixed.