1 Scheme Subset 12

Program ⟶ *Expr*
Expr ⟶ *uvar*
| `(quote` *Immediate*`)`
| `(if` *Expr Expr Expr*`)`
| `(begin` *Expr*`*` *Expr*`)`
| `(let ([`*uvar Expr*`]*) ` *Expr*`)`
| `(letrec ([`*uvar* `(lambda (`*uvar*`*) ` *Expr*`)]*) ` *Expr*`)`
| `(`*prim Expr*`*)`
| `(`*Expr Expr*`*)`
Immediate ⟶ *fixnum* | `()` | `#t` | `#f`

The only difference is that *label* in `letrec` is replaced by *uvar*, and *uvar* can now appear free within a `lambda` expression.

2 Things to do

With free variables, it's impossible to represent our procedure by bare squence of instructions. We need to convert them to *closures*. A closure consists of the code to run and values of its free variables. The code to run, however, is not identical to `(lambda (`*uvar*`*) ` *Expr*`*)`: it must accept an additional argument, namely the closure itself, to retrieve free variables.

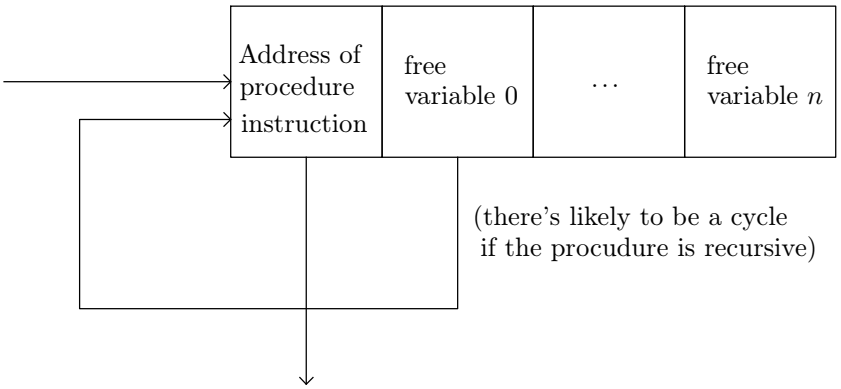

Figure 1. a typical (simplified) presentation of closure

We need to add three new passes `uncover-free`, `convert-closures`, `introduce-procedure-primitives`, and make changes to `normalize-context`, `specify-representation`.

2.1 `uncover-free`

This pass collects free variables in each `lambda` body and record it in a `free` form. For example,

```
(let ([x.1 (quote 10)])
  (letrec ([f.2 (lambda (y.3) (+ y.3 x.1))])
    (f.2 (quote 1))))
```

becomes

```
(let ([x.1 (quote 10)])
  (letrec ([f.2 (lambda (y.3)
                  (free (x.1)
                    (+ y.3 x.1)))])
```

```
(f.2 (quote 1))))
```

Remember the procedure name itself is free in its body.

2.2 convert-closures

This pass

- Replaces *uvar* with *label* in `letrec`. You can use `(unique-label f.1)` ⇒ `f$1`.
- Adds a new *uvar* to the formal parameter list of each `lambda`.
- Renames `(free` *uvar*`*)` to `(bind-free` *cp uvar*`*)` where *cp* is the added formal parameter.
- Wraps the body of `letrec` with a `closure` from. Specifically,

  `(letrec ...` *body*`)` ⇒ `(letrec ... (closure ([`*uvar label uvar*`*]) body))`

  where *uvar* is the name of the procedure, *uvar*`*` is formal parameters of the procedure (not including *cp*).
- Converts each procedure call to pass itself as the first argument.

For example, the code above becomes

```
(let ([x.1 '10])
  (letrec ([f$2 (lambda (cp.4 y.3)
                  (bind-free (cp.4 x.1)
                    (+ y.3 x.1)))])
    (closures ([f.2 f$2 x.1])
      (f.2 f.2 '1))))
```

2.3 introduce-procedure-primitives

This pass makes implicit uses of free variables and procedures explicit. To this end, we introduce five new primitives:

- `(make-procedure` *label n*`)` returns a closure with its address of code *label* and *n* slots to store free variables.
- `(procedure-ref` *proc n*`)` return the *n*th free variable in *proc*.
- `(procedure-code` *proc*`)` return the code field of *proc*.
- `(procedure-set!` *proc n expr*`)`, like `vector-set!`.
- `(procedure?` *proc*`)`

All of them except `procedure?` is not exposed to the user. They are only valid during compilation.

Every use of free variable is converted to a call to `procedure-ref`. Every procedure in procedure call should be wrapped by `procedure-code` (well, not actually, only if it's not a label. but at present it's impossible). `closure` form is converted to `procedure-make` and `procedure-set!`. `bind-free` is simply discarded. The output language should be the same as the input language of a11 (except the new primitives).

For example, the code above becomes

```
(let ([x.1 '10])
```

```
(letrec ([f$2 (lambda (cp.4 y.3)
                (+ y.3 (procedure-ref cp.4 '0)))])
  (let ([f.2 (make-procedure f$2 '1)])
    (begin
      (procedure-set! f.2 '0 x.1)
      ((procedure-code f.2) f.2 '1)))))
```

Remark: By the way I think binding all the free variables at the start of each procedure rather than replacing them with `procedure-ref` should also work and be faster. But I haven't tried yet…

2.4 `normalize-context`

This pass needs to handle added primitives.

2.5 `specify-representation`

This pass needs to handle added primitives. Please refer to helpers.scm for tags and masks.