6.821 Programming Languages Handout
Fall 2002

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Final Examination 2001

There are *five* problems on this examination. ***Make sure you don't skip over any of a problem's parts!*** They are followed by an appendix that contains reference material from the course notes. The appendix contains no problems; it is just a handy reference.

You will have *three hours* in which to work the problems. Some problems are easier than others: read all problems before beginning to work, and use your time wisely!

This examination is open-book: you may use whatever reference books or papers you have brought to the exam. The number of points awarded for each problem is placed in brackets next to the problem number. There are 100 points total on the exam.

Do all written work in your examination booklet – we will not collect the examination handout itself; you will only be graded for what appears in your examination booklet. It will be to your advantage to show your work – we will award partial credit for incorrect solutions that make use of the right techniques.

If you feel rushed, be sure to write a brief statement indicating the key idea you expect to use in your solutions. We understand time pressure, but we can't read your mind.

This examination has text printed on only one side of each page. Rather than flipping back and forth between pages, you may find it helpful to rip pages out of the exam so that you can look at more than one page at the same time.

Contents

Problem 1: Short Answer [20 points] 2
Problem 2: State: FLK# [20 points] 3
Problem 3: Explicit Types [20 points] 4
Problem 4: Type Reconstruction [20 points] 5
Problem 5: Compiling [20 points] 7
Appendix A: Standard Semantics of FLK! 8
Appendix B: Typing Rules for SCHEME/XSP 11
Appendix C: Typing Rules for SCHEME/R 13
Appendix D: Type Reconstruction Algorithm for SCHEME/R 14
Appendix E: Meta-CPS Conversion Rules 15
Appendix F: Match Desugaring 16

The figures in the Appendix are very similar to the ones in the course notes. Some bugs have been fixed, and some figures have been simplified to remove parts inessential for this exam. You will not be marked down if you use the corresponding figures in the course notes instead of the appendices.

Problem 1: Short Answer [20 points]

a. Given the following domains:

$$\mathrm{A} = \{1\}_\perp$$
$$\mathrm{B} = \{a,\ b\}_\perp$$

(i) [2 points]How many set theoretic functions are there in $A \to B$?

(ii) [2 points] How many continuous functions are there in $A \to B$?

b. In Scheme/R (Scheme with type reconstruction) give the most general type schemes of the following expressions, or state why the expression does not have a type:

(i) [1 points]

```
(lambda (f) (lambda (g) (lambda (x) (g (f x)))))
```

(ii) [1 points]

```
(lambda (x) (x x))
```

(iii) [1 points]

```
(letrec ((f (lambda (x) (if (id x) (id 0) (id 1))))
         (id (lambda (y) y)))
        (f #t))
```

(iv) [1 points]

```
(lambda (id) (if (id #t) (id 0) (id 1)))
```

c. Give the equivalent Scheme/XSP expression, and the type thereof, for each of the expressions in the previous part (part b).

(i) [2 points]

(ii) [2 points]

(iii) [2 points]

(iv) [2 points]

d. [2 points] Give the desugaring of the following Scheme/R expression (the match desugaring can be found in Appendix F)

```
(match z
   ((cons 1 x) x)
   (x (cons 1 x)))
```

e. [2 points] Use define-datatype to define the (queueof T) datatype that represents a queue with a list. For example, a (queueof int) would be represented by an integer list.

Problem 2: State: FLK# [20 points]

YOUR ANSWERS TO THIS PROBLEM SHOULD BE BASED ON THE STANDARD DENOTATIONAL SEMANTICS FOR FLK! AS PRESENTED IN APPENDIX A.

Sam Antics is working on a new language with hot new features that will appeal to government customers. He was going to base his language on Caffeine from Moon Microsystems, but negotiations broke down. He has therefore decided to extend FLK! and has hired you, a top FLK! consultant, to assist with modifying the language to support these new features. The new language is called FLK#, part of Sam Antics' new .GOV platform. The big feature of FLK# is user tracking and quotas in the store. An important customer observed that government users tended to use the store carelessly, resulting in expensive memory upgrades. To improve the situation, the FLK# store will maintain a per-user quota. (A quota restricts the number of cells a particular user can allocate.) The Standard Semantics of FLK! are changed as follows:

$$w \in \text{UserID} = \text{Int}$$
$$q \in \text{Quota} = \text{UserID} \to \text{Int}$$
$$\gamma \in \text{Cmdcont} = \text{UserID} \to \text{Quota} \to \text{Store} \to \text{Expressible}$$

error-cont : Error → Cmdcont
$= \lambda y\,.\ \lambda w\,.\ \lambda q\,.\ \lambda s\,.\ (\text{Error} \mapsto \text{Expressible}\ \ y)$

UserID is just an integer. User ID 0 is reserved for the case when no one is logged in. Quota is a function that when given a UserID returns the number of cells remaining in the user's quota. The quota starts at 100 cells, and a user's quota is tracked throughout the lifetime of the program (i.e., the quota is not reset upon logout). Cmdcont has been changed to take the currently logged in user ID, the current quota, and the current store to yield an answer. Plus, FLK# adds the following commands:

E	::=	...	[Classic FLK! expressions]
	|	`(login!` w`)`	[Log in user w]
	|	`(logout!)`	[Log out current user]
	|	`(check-quota)`	[Check user quota]

(`login!` w) - logs in the user associated with the identifier w; returns w (returns an error if a user is already logged in or if the UserID is 0)
(`logout!`) - logs the current user out; returns the last user's identifier (returns an error if there is no user logged in)
(`check-quota`) - returns the amount of quota remaining

The definition of $\mathcal{E}[\![$`(check-quota)`$]\!]$ is:

$\mathcal{E}[\![$`(check-quota)`$]\!] =$
$\lambda ekwq\,.$ **if** $w = 0$
then *error-cont* no-user-logged-in $\ w\ \ q$
else (k (Int $\mapsto$ Value $(q\ \ w)$) $w\ \ q$) **fi**

a. [5 points] Write the meaning function clause for $\mathcal{E}[\![$`(login!` E`)`$]\!]$.

b. [5 points] Write the meaning function clause for $\mathcal{E}[\![$`(logout!)`$]\!]$.

c. [5 points] Give the definition of $\mathcal{E}[\![$`(cell` E`)`$]\!]$. Remember you cannot create a cell unless you are logged in.

d. [5 points] Naturally, Sam Antics wants to embed some "trap doors" into the .GOV platform to enable him to "learn more about his customers." One of these trap doors is the undocumented `(raise-quota! n)` command, which adds n cells to the quota of the current user and returns 0. Give the definition of $\mathcal{E}[\![$`(raise-quota!` E`)`$]\!]$.

Problem 3: Explicit Types [20 points]

ANSWERS FOR THE FOLLOWING QUESTIONS SHOULD BE BASED ON THE SCHEME/XSP TYPING RULES GIVEN IN APPENDIX B.

Louis Reasoner has had a hard time implementing letrec in a call-by-name version of Scheme/XSP, and has decided to use the fixed point operator FIX instead. For example, here the the correct defintion of factorial in Louis' approach:

```
(let ((fact-gen (lambda ((fact (-> (int) int)))
         (lambda ((n int)) (if (= n 0) 1 (* n (fact (- n 1))))))))
  ((proj fix (-> (int) int)) fact-gen))
```

Thus `fix` is a procedure that computes the fixed point of a generating function. Ben Bitdiddle has been called on the scene to help, and he has ensured that Louis' Scheme/XSP supports recursive types using RECOF (see Appendix B).

a. [4 points] What is the type of `fact-gen`?

b. [3 points] What is the type of `fix`?

c. [3 points] What is the type of `((proj fix (-> (int) int)) fact-gen)`?

Ben Bitdiddle defined the call-by-name version of fix to be:

```
(let ((fix (plambda (t) (lambda ((f T1))
             (lambda ((x T2)) (f (x x))) (lambda ((x T2)) (f (x x)))))))
    ... fix can be used here ...

    )
```

d. [3 points] What is T1?

e. [4 points] What is T2?

f. [3 points] Louis has decided that he would like `(fix E)` to be a standard expression in his language. What is the typing rule for `(fix E)`?

Problem 4: Type Reconstruction [20 points]

ANSWERS FOR THE FOLLOWING QUESTIONS SHOULD BE BASED ON THE SCHEME/R TYPING RULES AND TYPE RECONSTRUCTION ALGORITHM GIVEN IN APPENDICES C AND D.

With sales declining and customers flocking to competitors' products, the board of directors at Prophet.com has decided to oust CTO Louis Reasoner and has assigned you and Alyssa P. Hacker as the pro tempore co-CTOs. Alyssa believes the secret to regaining market share is to make Scheme/R more Internet-friendly. The next generation product, code-named Scheme/R $9i$ (the i stands for Internet), contains socket functionality to make it easier to write Internet servers.

A socket is like a stream or a channel in that you can read data from and write data to sockets. Sockets are named by a port number and also have a specific data type associated with them that determines the type of data that can be transmitted or received over the socket. (For the purpose of this problem, you can ignore any problems involved with opening more than one socket on the same port.)
We introduce a new type `(socketof T)` and six new forms:

- `(int-socket` E_{port}`)` returns a new integer socket.
- `(bool-socket` E_{port}`)` returns a new boolean socket.
- `(unit-socket` E_{port}`)` returns a new unit socket.
- `(symbol-socket` E_{port}`)` returns a new symbol socket.
- `(read-all!` E_{socket} E_{reader}`)` takes a socket and calls procedure E_{reader} once for each item remaining in the socket to be read; returns `#u`.
- `(write!` E_{socket} E_{datum}`)` Writes E_{datum} into the socket and returns `#u`.

Alyssa has written the following Scheme/R type rules to get you started:

$$\frac{A \vdash E : \texttt{int}}{A \vdash \texttt{(int-socket } E\texttt{)} : \texttt{(socketof int)}} \quad [\textit{int-socket}]$$

$$\frac{A \vdash E : \texttt{int}}{A \vdash \texttt{(bool-socket } E\texttt{)} : \texttt{(socketof bool)}} \quad [\textit{bool-socket}]$$

$$\frac{A \vdash E : \texttt{int}}{A \vdash \texttt{(unit-socket } E\texttt{)} : \texttt{(socketof unit)}} \quad [\textit{unit-socket}]$$

$$\frac{A \vdash E : \texttt{int}}{A \vdash \texttt{(symbol-socket } E\texttt{)} : \texttt{(socketof symbol)}} \quad [\textit{symbol-socket}]$$

$$\frac{\begin{array}{c} A \vdash E_{socket} : \texttt{(socketof T)} \\ A \vdash E_{reader} : \texttt{(-> (T) unit)} \end{array}}{A \vdash \texttt{(read-all! } E_{socket}\ E_{reader}\texttt{)} : \texttt{unit}} \quad [\textit{read-all!}]$$

$$\frac{\begin{array}{c} A \vdash E_{socket} : \texttt{(socketof T)} \\ A \vdash E_{datum} : \texttt{T} \end{array}}{A \vdash \texttt{(write! } E_{socket}\ E_{datum}\texttt{)} : \texttt{unit}} \quad [\textit{write!}]$$

She has also agreed to write the implementation. Because you are a high-paid 6.821 consultant, your part is to write the type reconstruction algorithm for these constructs.

a. [4 points] Give the type reconstruction algorithm for `(int-socket` E_{port}`)`.

b. [4 points] Give the type reconstruction algorithm for `(write!` E_{socket} E_{datum}`)`.

c. [4 points] Give the type reconstruction algorithm for `(read-all!` E_{socket} E_{reader}`)`.

d. [4 points] As part of Louis's severance agreement, he agreed to stay on for one month to write a proxy server for Prophet.com's intranet (by proxy server we mean something that reads data on one socket and writes it to another). He wrote the following code:

```
(letrec ((proxy (lambda (socket-in socket-out)
                        (read-all! socket-in
                                   (lambda (x) (write! socket-out x)))))
         (do-proxy-http (lambda () (proxy (symbol-socket 80)
                                          (symbol-socket 8080))))
         (do-proxy-ftp (lambda () (proxy (int-socket 20)
                                         (int-socket 8020)))))
  (begin
    (do-proxy-http)
    (do-proxy-ftp)))
```

Unfortunately, on his way out on his last day, he gives you the code and tells you it doesn't type check. Give a semantically equivalent (i.e., preserves procedures and procedure calls) replacement for Louis' code that does type check.

e. [4 points] Being the astute 6.821 consultant that you are, you also discover that Louis has used a construct that doesn't have a type reconstruction algorithm in the book: `begin`. Give the type reconstruction algorithm for `(begin` E_1 E_2`)`.

Problem 5: Compiling [20 points]

ANSWERS FOR THE FOLLOWING QUESTIONS SHOULD BE BASED ON THE META CPS CONVERSION ALGORITHM GIVEN IN APPENDIX E.

a. [7 points] What source code generated the following output from the Tortoise compiler?

```
(program
 (define *top*
   (%closure (lambda (.closure8. x) x)))
 (call-closure
  *top*
  (%closure
   (lambda (.closure7. f .k1.)
     (call-closure
      .k1.
      (%closure
       (lambda (.closure6. x .k2.)
         (call-closure
          (%closure-ref .closure6. 1)
          x
          (%closure
           (lambda (.closure5. .t3.)
             (call-closure (%closure-ref .closure5. 1)
                           .t3.
                           (%closure-ref .closure5. 2)))
           (%closure-ref .closure6. 1)
           .k2.)))
       f))))))
```

b. [7 points] The meaning of `(COND (`P_1 E_1`) (`P_2 E_2`) (else` E_3`))` is E_1 if P_1 is true, E_2 is P_1 is false and P_2 is true, and E_3 otherwise.

What is $\mathcal{MCPS}[\![$`(COND (`P_1 E_1`) (`P_2 E_2`) (else` E_3`))`$]\!]$?

c. Louis Reasoner has decided to add garbage collection to a language that previously employed explicit storage allocation with MALLOC and FREE opeators. His new implementation ignores FREE and reclaims space using a brand new and correct garbage collector. The garbage collector has more than twice as much heap space as the old explictly managed heap. As soon as this new version of the language is released, several programs that used to run fine – crash!!

(i) [3 points] What is the problem?

(ii) [3 points] How can the programmers fix the problems with their programs?

Appendix A: Standard Semantics of FLK!

$v \in$ Value $=$ Unit + Bool + Int + Sym + Pair + Procedure + Location
$k \in$ Expcont $=$ Value $\to$ Cmdcont
$\gamma \in$ Cmdcont $=$ Store $\to$ Expressible
Expressible $=$ $(\text{Value} + \text{Error})_\perp$
Error $=$ Sym
$p \in$ Procedure $=$ Denotable $\to$ Expcont $\to$ Cmdcont
$d \in$ Denotable $=$ Value
$e \in$ Environment $=$ Identifier $\to$ Binding
$\beta \in$ Binding $=$ $(\text{Denotable} + \text{Unbound})_\perp$
Unbound $=$ $\{unbound\}$
$s \in$ Store $=$ Location $\to$ Assignment
$l \in$ Location $=$ Nat
$\alpha \in$ Assignment $=$ $(\text{Storable} + \text{Unassigned})_\perp$
$\sigma \in$ Storable $=$ Value
Unassigned $=$ $\{unassigned\}$

top-level-cont : Expcont
$= \lambda v\,.\ \lambda s\,.$ (Value $\mapsto$Expressible v)
error-cont : Error $\to$ Cmdcont
$= \lambda y\,.\ \lambda s\,.$ (Error $\mapsto$Expressible y)

empty-env : Environment $= \lambda I\,.$ (Unbound $\mapsto$Binding *unbound*)

test-boolean : (Bool $\to$ Cmdcont) $\to$ Expcont
$= \lambda f\,.$ $(\lambda v\,.$ **matching** v
$\triangleright$ (Bool $\mapsto$Value b) $[\![]$ $(f\ b)$
$\triangleright$ **else** (*error-cont* `non-boolean`)
endmatching)
Similarly for:
test-procedure : (Procedure $\to$ Cmdcont) $\to$ Expcont
test-location : (Location $\to$ Cmdcont) $\to$ Expcont
etc.

ensure-bound : Binding $\to$ Expcont $\to$ Cmdcont
$= \lambda\beta k\,.$ **matching** β
$\triangleright$ (Denotable $\mapsto$Binding v) $[\![]$ $(k\ v)$
$\triangleright$ (Unbound $\mapsto$Binding *unbound*) $[\![]$ (*error-cont* `unbound-variable`)
endmatching
Similarly for:
ensure-assigned : Assignment $\to$ Expcont $\to$ Cmdcont

Figure 1: Semantic algebras for standard semantics of strict CBV FLK!.

```
same-location? : Location → Location → Bool = λl1 l2 . (l1 =Nat l2)
next-location : Location → Location = λl . (l +Nat 1)

empty-store : Store = λl . (Unassigned ↦Assignment unassigned)

fetch : Location → Store → Assignment = λls . (s l)

assign : Location → Storable → Store → Store
 = λl1 σ s . λl2 . if (same-location? l1 l2)
                  then (Storable ↦Assignment σ)
                  else (fetch l2 s)

fresh-loc : Store → Location = λs . (first-fresh s 0)

first-fresh : Store → Location → Location
 = λs l . matching (fetch l s)
          ▷ (Unassigned ↦Assignment unassigned) ⟦ l
          ▷ else (first-fresh s (next-location l))
          endmatching

lookup : Environment → Identifier → Binding = λe I . (e I)
```

Figure 2: Store helper functions for standard semantics of strict CBV FLK!.

$\mathcal{TL}$: Exp → Expressible
$\mathcal{E}$: Exp → Environment → Expcont → Cmdcont
$\mathcal{L}$: Lit → Value *; Defined as usual*

$\mathcal{TL}[\![E]\!] = \mathcal{E}[\![E]\!]$ *empty-env top-level-cont empty-store*

$\mathcal{E}[\![L]\!] = \lambda ek\,.\ k\ \mathcal{L}[\![L]\!]$

$\mathcal{E}[\![I]\!] = \lambda ek\,.$ *ensure-bound* (*lookup* $e\ I$) k

$\mathcal{E}[\![\texttt{(proc}\ I\ E\texttt{)}]\!] = \lambda ek\,.\ k$ (Procedure $\mapsto$Value $(\lambda dk'\,.\,\mathcal{E}[\![E]\!]\ [I : d]e\ k')$)

$\mathcal{E}[\![\texttt{(call}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e$ (*test-procedure* $(\lambda p\,.\,\mathcal{E}[\![E_2]\!]\ e\ (\lambda v\,.\,p\ v\ k))$)

$\mathcal{E}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!] =$
$\lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e$ (*test-boolean* $(\lambda b\,.$ **if** b **then** $\mathcal{E}[\![E_2]\!]\ e\ k$ **else** $\mathcal{E}[\![E_3]\!]\ e\ k)$)

$\mathcal{E}[\![\texttt{(pair}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\lambda v_1\,.\,\mathcal{E}[\![E_2]\!]\ e\ (\lambda v_2\,.\,k$ (Pair $\mapsto$Value $\langle v_1, v_2\rangle))))$

$\mathcal{E}[\![\texttt{(cell}\ E\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E]\!]\ e\ \ (\lambda vs\,.\ k$ (Location $\mapsto$Value (*fresh-loc* s))
(*assign* (*fresh-loc* s) $v\ s$))

$\mathcal{E}[\![\texttt{(begin}\ E_1\ E_2\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\lambda v_{\text{ignore}}\,.\,\mathcal{E}[\![E_2]\!]\ e\ k)$

$\mathcal{E}[\![\texttt{(primop cell-ref}\ E\texttt{)}]\!] = \lambda ek\,.\ \mathcal{E}[\![E]\!]\ e$ (*test-location* $(\lambda ls\,.$ *ensure-assigned* (*fetch* $l\ s$) $k\ s)$)

$\mathcal{E}[\![\texttt{(primop cell-set!}\ E_1\ E_2\texttt{)}]\!]$
$= \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e$ (*test-location* $(\lambda l\,.\,\mathcal{E}[\![E_2]\!]\ e\ (\lambda vs\,.\,k$ (Unit $\mapsto$Value *unit*) (*assign* $l\ v\ s$))))

$\mathcal{E}[\![\texttt{(rec}\ I\ E\texttt{)}]\!] = \lambda eks\,.$ let f = $\mathbf{fix}_{\text{Expressible}}$ $(\lambda a\,.\ \mathcal{E}[\![E]\!]$ [I : (*extract-value* a)] e top-level-cont s)
matching f
▷ (Value $\mapsto$Expressible v) ‖ $\mathcal{E}[\![E]\!]$ [$I : v$] e k s
▷ **else** f
endmatching

extract-value : Expressible → Binding
$= \lambda a\,.$ **matching** a
▷ (Value $\mapsto$Expressible v) ‖ (Denotable $\mapsto$Binding v)
▷ **else** $\perp_{\text{Binding}}$
endmatching

Figure 3: Valuation clauses for standard semantics of strict CBV FLK!.

Appendix B: Typing Rules for SCHEME/XSP

SCHEME/X Rules

$$\vdash N : \texttt{int} \qquad [\textit{int}]$$

$$\vdash B : \texttt{bool} \qquad [\textit{bool}]$$

$$\vdash S : \texttt{string} \qquad [\textit{string}]$$

$$\vdash \texttt{(symbol } I\texttt{)} : \texttt{sym} \qquad [\textit{sym}]$$

$$A[I{:}T] \vdash I : T \qquad [\textit{var}]$$

$$\frac{\forall i\ (A \vdash E_i : T_i)}{A \vdash \texttt{(begin } E_1\ \ldots\ E_n\texttt{)} : T_n} \qquad [\textit{begin}]$$

$$\frac{A \vdash E : T}{A \vdash \texttt{(the } T\ E\texttt{)} : T} \qquad [\textit{the}]$$

$$\frac{A \vdash E_1 : \texttt{bool}\ ;\ A \vdash E_2 : T\ ;\ A \vdash E_3 : T}{A \vdash \texttt{(if } E_1\ E_2\ E_3\texttt{)} : T} \qquad [\textit{if}]$$

$$\frac{A[I_1{:}T_1,\ \ldots,\ I_n{:}T_n] \vdash E_B : T_B}{A \vdash \texttt{(lambda ((}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{)) } E_B\texttt{)} : \texttt{(->} \ (T_1\ \ldots\ T_n)\ T_B\texttt{)}} \qquad [\lambda]$$

$$\frac{A \vdash E_P : \texttt{(->} \ (T_1\ \ldots\ T_n)\ T_B\texttt{)} \quad \forall i\ (A \vdash E_i : T_i)}{A \vdash \texttt{(}E_P\ E_1\ \ldots\ E_n\texttt{)} : T_B} \qquad [\textit{call}]$$

$$\frac{\forall i\ (A \vdash E_i : T_i) \quad A[I_1{:}T_1,\ \ldots,\ I_n{:}T_n] \vdash E_B : T_B}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{)) } E_B\texttt{)} : T_B} \qquad [\textit{let}]$$

$$\frac{A' = A[I_1{:}T_1,\ \ldots,\ I_n{:}T_n] \quad \forall i\ (A' \vdash E_i : T_i) \quad A' \vdash E_B : T_B}{A \vdash \texttt{(letrec ((}I_1\ T_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_1\ E_n\texttt{)) } E_B\texttt{)} : T_B} \qquad [\textit{letrec}]$$

$$\frac{A \vdash (\forall i\ \ [T_i/I_i])E_{body} : T_{body}}{A \vdash \texttt{(tlet ((}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{)) } E_{body}\texttt{)} : T_{body}} \qquad [\textit{tlet}]$$

$$\frac{\forall i\ (A \vdash E_i : T_i)}{A \vdash \texttt{(record (}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{)) : (recordof (}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}} \qquad [\textit{record}]$$

$$\frac{A \vdash E : \texttt{(recordof} \ \ldots\ \texttt{(}I\ T\texttt{)}\ \ldots\texttt{)}}{A \vdash \texttt{(select } I\ E\texttt{)} : T} \qquad [\textit{select}]$$

$$\frac{A \vdash E : T_E\ ;\ T = \texttt{(oneof} \ \ldots\ \texttt{(}I\ T_E\texttt{)}\ \ldots\texttt{)}}{A \vdash \texttt{(one } T\ I\ E\texttt{)} : T} \qquad [\textit{one}]$$

$$\frac{A \vdash E_{disc} : \texttt{(oneof (}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))} \quad \forall i\ .\ \exists j\ .\ ((I_i = I_{tag_j}) \wedge (A[I_{val_j}{:}T_i] \vdash E_j : T))}{A \vdash \texttt{(tagcase } E_{disc}\ \texttt{(}I_{tag_1}\ I_{val_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}I_{tag_n}\ I_{val_n}\ E_n\texttt{))} : T} \qquad [\textit{tagcase1}]$$

$$\frac{A \vdash E_{disc} : \texttt{(oneof (}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))} \quad \forall i \mid (\exists j\ .\ (I_i = I_{tag_j}))\ .\ A[I_{val_j}{:}T_i] \vdash E_j : T \quad A \vdash E_{default} : T}{A \vdash \texttt{(tagcase } E_{disc}\ \texttt{(}I_{tag_1}\ I_{val_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}I_{tag_n}\ I_{val_n}\ E_n\texttt{) (else } E_{default}\texttt{))} : T} \qquad [\textit{tagcase2}]$$

Rules Introduced by SCHEME/XS to Handle Subtyping

$$T \sqsubseteq T \qquad [\textit{reflexive-}\sqsubseteq]$$

$$\frac{T_1 \sqsubseteq T_2 \;;\; T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3} \qquad [\textit{transitive-}\sqsubseteq]$$

$$\frac{(T_1 \sqsubseteq T_2)\quad (T_2 \sqsubseteq T_1)}{T_1 \equiv T_2} \qquad [\equiv]$$

$$\frac{\forall i\ \exists j\ ((I_i = J_j) \wedge (S_j \sqsubseteq T_i))}{\texttt{(recordof (}J_1\ S_1\texttt{) ...(}J_m\ S_m\texttt{))} \sqsubseteq \texttt{(recordof (}I_1\ T_1\texttt{) ...(}I_n\ T_n\texttt{))}} \qquad [\textit{recordof-}\sqsubseteq]$$

$$\frac{\forall j\ \exists i\ ((J_j = I_i) \wedge (S_j \sqsubseteq S_i))}{\texttt{(oneof (}J_1\ S_1\texttt{) ...(}J_m\ S_m\texttt{))} \sqsubseteq \texttt{(oneof (}I_1\ T_1\texttt{) ...(}I_n\ T_n\texttt{))}} \qquad [\textit{oneof-}\sqsubseteq]$$

$$\frac{\forall i\ (T_i \sqsubseteq S_i)\ ;\ S_{body} \sqsubseteq T_{body}}{\texttt{(->}\ \texttt{(}S_1\ \ldots S_n\texttt{)}\ S_{body}\texttt{)} \sqsubseteq \texttt{(->}\ \texttt{(}T_1\ \ldots T_n\texttt{)}\ T_{body}\texttt{)}} \qquad [\texttt{->}\text{-}\sqsubseteq]$$

$$\frac{\forall T\ ([T/I_1]T_1 \sqsubseteq [T/I_2]T_2)}{\texttt{(recof}\ I_1\ T_1\texttt{)} \sqsubseteq \texttt{(recof}\ I_2\ T_2\texttt{)}} \qquad [\textit{recof-}\sqsubseteq]$$

$$\frac{A \vdash E_{rator} : \texttt{(->}\ \texttt{(}T_1\ \ldots T_n\texttt{)}\ T_{body}\texttt{)}\quad \forall i\ ((A \vdash E_i : S_i) \wedge (S_i \sqsubseteq T_i))}{A \vdash \texttt{(}E_{rator}\ E_1\ \ldots E_n\texttt{)} : T_{body}} \qquad [\textit{call-inclusion}]$$

$$\frac{A \vdash E : S\quad S \sqsubseteq T}{A \vdash \texttt{(the}\ T\ E\texttt{)} : T} \qquad [\textit{the-inclusion}]$$

Rules Introduced by SCHEME/XSP to Handle Polymorphism

$$\frac{A \vdash E : T;\quad \forall i\ (I_i \notin (\textit{FTV}\ \ \texttt{(}\textit{Free-Ids}[\![E]\!]\texttt{)}A))}{A \vdash \texttt{(plambda (}I_1\ \ldots\ I_n\texttt{)}\ E\texttt{)} : \texttt{(poly (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}} \qquad [p\lambda]$$

$$\frac{A \vdash E : \texttt{(poly (}I_1\ \ldots\ I_n\texttt{)}\ T_E\texttt{)}}{A \vdash \texttt{(proj}\ E\ T_1\ \ldots\ T_n\texttt{)} : (\forall i\ \ [T_i/I_i])\ T_E} \qquad [\textit{project}]$$

$$\frac{(\forall i\ \ [I_i/J_i])\ S \sqsubseteq T,\ \ \forall i\ (I_i \notin \textit{Free-Ids}[\![S]\!])}{\texttt{(poly (}J_1\ \ldots\ J_n\texttt{)}\ S\texttt{)} \sqsubseteq \texttt{(poly (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}} \qquad [\textit{poly-}\sqsubseteq]$$

recof Equivalence

$$(\text{recof}\ \ I\ \ T) \equiv [(\textit{recof}\ \ I\ \ T)/I]T$$

Appendix C: Typing Rules for SCHEME/R

$$\vdash \texttt{\#u} : \texttt{unit} \qquad [\textit{unit}]$$

$$\vdash B : \texttt{bool} \qquad [\textit{bool}]$$

$$\vdash N : \texttt{int} \qquad [\textit{int}]$$

$$\vdash \texttt{(symbol } I\texttt{)} : \texttt{sym} \qquad [\textit{symbol}]$$

$$[\ldots, I\texttt{:}T, \ldots] \vdash I : T \qquad [\textit{var}]$$

$$[\ldots, I\texttt{:(generic (}I_1\ \ldots\ I_n\texttt{)}\ T_{body}\texttt{)}, \ldots] \vdash I : (\forall i\ [T_i/I_i])T_{body} \qquad [\textit{genvar}]$$

$$\frac{A \vdash E_{test} : \texttt{bool}\ ;\ A \vdash E_{con} : T\ ;\ A \vdash E_{alt} : T}{A \vdash \texttt{(if } E_{test}\ E_{con}\ E_{alt}\texttt{)} : T} \qquad [\textit{if}]$$

$$\frac{A[I_1\texttt{:}T_1\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{:}T_n] \vdash E_{body} : T_{body}}{A \vdash \texttt{(lambda (}I_1\ \ldots\ I_n\texttt{)}\ E_{body}\texttt{)} : \texttt{(->}\ \texttt{(}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)}} \qquad [\lambda]$$

$$\frac{A \vdash E_{rator} : \texttt{(->}\ \texttt{(}T_1\ \ldots\ T_n\texttt{)}\ T_{body}\texttt{)} \quad \forall i\ .\ (A \vdash E_i : T_i)}{A \vdash \texttt{(}E_{rator}\ E_1\ \ldots\ E_n\texttt{)} : T_{body}} \qquad [\textit{apply}]$$

$$\frac{\forall i\ .\ (A \vdash E_i : T_i) \quad A[I_1\texttt{:}\mathit{Gen}(T_1\texttt{,}\ A)\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{:}\mathit{Gen}(T_n\texttt{,}\ A)] \vdash E_{body} : T_{body}}{A \vdash \texttt{(let ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} : T_{body}} \qquad [\textit{let}]$$

$$\frac{\forall i\ .\ (A[I_1\texttt{:}T_1\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{:}T_n] \vdash E_i : T_i) \quad A[I_1\texttt{:}\mathit{Gen}(T_1\texttt{,}\ A)\texttt{,}\ \ldots\ I_n\texttt{:}\mathit{Gen}(T_n\texttt{,}\ A)] \vdash E_{body} : T_{body}}{A \vdash \texttt{(letrec ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} : T_{body}} \qquad [\textit{letrec}]$$

$$\frac{\forall i\ .\ (A \vdash E_i : T_i)}{A \vdash \texttt{(record (}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))} : \texttt{(recordof (}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))}} \qquad [\textit{record}]$$

$$\frac{A \vdash E_r : \texttt{(recordof (}I_1\ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ T_n\texttt{))} \quad A[I_1\texttt{:}T_1\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{:}T_n] \vdash E_b : T}{A \vdash \texttt{(with (}I_1\ \ldots\ I_n\texttt{)}\ E_r\ E_b\texttt{)} : T} \qquad [\textit{with}]$$

$$\frac{A \vdash \texttt{(letrec ((}I_1\ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ E_n\texttt{))}\ E_{body}\texttt{)} : T}{A \vdash \texttt{(program (define }I_1\ E_1\texttt{)}\ \ldots\ \texttt{(define }I_n\ E_n\texttt{)}\ E_{body}\texttt{)} : T} \qquad [\textit{program}]$$

$$\mathit{Gen}(T, A) = \texttt{(generic (}I_1\ \ldots\ I_n\texttt{)}\ T\texttt{)}, \text{where } \{I_i\} = \mathit{FTV}\ (T) - \mathit{FTE}(A)$$

Appendix D: Type Reconstruction Algorithm for SCHEME/R

$R[\![\texttt{\#u}]\!]\ A\ S = \langle \texttt{unit}, S\rangle$

$R[\![B]\!]\ A\ S = \langle \texttt{bool}, S\rangle$

$R[\![N]\!]\ A\ S = \langle \texttt{int}, S\rangle$

$R[\![\texttt{(symbol } I\texttt{)}]\!]\ A\ S = \langle \texttt{sym}, S\rangle$

$R[\![I]\!]\ A[I : T]\ S = \langle T, S\rangle$

$R[\![I]\!]\ A[I : \texttt{(generic (}I_1\ \ldots\ I_n\texttt{) } T\texttt{)}]\ S = \langle T[?v_i/I_i], S\rangle$ ($?v_i$ are new)

$R[\![I]\!]\ A\ S =$ fail (when I is unbound)

$$R[\![\texttt{(if } E_t\ E_c\ E_a\texttt{)}]\!]\ A\ S = \textbf{let}\ \langle T_t, S_t\rangle = R[\![E_t]\!]A\ S\ \textbf{in}\ \textbf{let}\ S'_t = U(T_t, \texttt{bool}, S_t)\ \textbf{in}\ \textbf{let}\ \langle T_c, S_c\rangle = R[\![E_c]\!]A\ S'_t\ \textbf{in}\ \textbf{let}\ \langle T_a, S_a\rangle = R[\![E_a]\!]A\ S_c\ \textbf{in}\ \textbf{let}\ S'_a = U(T_c, T_a, S_a)\ \textbf{in}\ \langle T_a, S'_a\rangle$$

$$R[\![\texttt{(lambda (}I_1\ \ldots\ I_n\texttt{) } E_b\texttt{)}]\!]\ A\ S = \textbf{let}\ \langle T_b, S_b\rangle = R[\![E_b]\!]A[I_i :?v_i]\ S\ \textbf{in}\ \langle \texttt{(->} (?v_1\ \ldots\ ?v_n\texttt{) } T_b\texttt{)}, S_b\rangle \quad (?v_i \text{ are new})$$

$$R[\![\texttt{(}E_0\ E_1\ \ldots\ E_n\texttt{)}]\!]\ A\ S = \textbf{let}\ \langle T_0, S_0\rangle = R[\![E_0]\!]A\ S\ \textbf{in}\ \ldots\ \textbf{let}\ \langle T_n, S_n\rangle = R[\![E_n]\!]A\ S_{n-1}\ \textbf{in}\ \textbf{let}\ S_f = U(T_0, \texttt{(->} (T_1\ \ldots\ T_n\texttt{) } ?v_f\texttt{)}, S_n)\ \textbf{in}\ \langle ?v_f, S_f\rangle \quad (?v_f \text{ is new})$$

$$R[\![\texttt{(let ((}I_1\ E_1\texttt{) } \ldots\ \texttt{(}I_n\ E_n\texttt{)) } E_b\texttt{)}]\!]\ A\ S = \textbf{let}\ \langle T_1, S_1\rangle = R[\![E_1]\!]A\ S\ \textbf{in}\ \ldots\ \textbf{let}\ \langle T_n, S_n\rangle = R[\![E_n]\!]A\ S_{n-1}\ \textbf{in}\ R[\![E_b]\!]A[I_i : \mathit{Rgen}(T_i, A, S_n)]S_n$$

$$R[\![\texttt{(letrec ((}I_1\ E_1\texttt{) } \ldots\ \texttt{(}I_n\ E_n\texttt{)) } E_b\texttt{)}]\!]\ A\ S = \textbf{let}\ A_1 = A[I_i :?v_i] \quad (?v_i \text{ are new})\ \textbf{in}\ \textbf{let}\ \langle T_1, S_1\rangle = R[\![E_1]\!]A_1\ S\ \textbf{in}\ \ldots\ \textbf{let}\ \langle T_n, S_n\rangle = R[\![E_n]\!]A_1\ S_{n-1}\ \textbf{in}\ \textbf{let}\ S_b = U(?v_i, T_i, S_n)\ \textbf{in}\ R[\![E_b]\!]A[I_i : \mathit{Rgen}(T_i, A, S_b)]\ S_b$$

$$R[\![\texttt{(record (}I_1\ E_1\texttt{) } \ldots\ \texttt{(}I_n\ E_n\texttt{))}]\!]\ A\ S = \textbf{let}\ \langle T_1, S_1\rangle = R[\![E_1]\!]A\ S\ \textbf{in}\ \ldots\ \textbf{let}\ \langle T_n, S_n\rangle = R[\![E_n]\!]A\ S_{n-1}\ \textbf{in}\ \langle \texttt{(recordof (}I_1\ T_1\texttt{) } \ldots\ \texttt{(}I_n\ T_n\texttt{))}, S_n\rangle$$

$$R[\![\texttt{(with (}I_1\ \ldots\ I_n\texttt{) } E_r\ E_b\texttt{)}]\!]\ A\ S = \textbf{let}\ \langle T_r, S_r\rangle = R[\![E_r]\!]A\ S\ \textbf{in}\ \textbf{let}\ S_b = U(T_r, \texttt{(recordof (}I_1\ ?v_i\texttt{) } \ldots\ \texttt{(}I_n\ ?v_n\texttt{))}, S_r) \quad (?v_i \text{ are new})\ \textbf{in}\ R[\![E_b]\!]A[I_i :?v_i]\ S_b$$

$$\mathit{Rgen}(T, A, S) = \mathit{Gen}((S\ T), (\textit{subst-in-type-env}\ S\ A))$$

Appendix E: Meta-CPS Conversion Rules

In the following rules, grey mathematical notation (like λv) and square brackets [ ] are used for "meta-application", which is evaluated *as part of meta-CPS conversion.* Code in `BLACK TYPEWRITER FONT` is part of the output program; meta-CPS conversion does *not* evaluate any of this code. Therefore, you can think of meta-CPS-converting an expression E as rewriting $\mathcal{MCPS}[\![E]\!]$ until no grey is left.

E $\in$ Exp
m $\in$ Meta-Continuation $=$ Exp $\to$ Exp

meta-cont $\to$ *exp* : (Exp $\to$ Exp) $\to$ Exp $= [\lambda m\,.\,$ `(LAMBDA (t)`$[m\ $`t)`$]$
exp $\to$ *meta-cont* : Exp $\to$ (Exp $\to$ Exp) $= [\lambda E\,.\,[\lambda V\,.\,$ `(CALL` $E\ V$`)`$]]$

meta-cont$\to$exp $[\lambda V\,.\,$ `(CALL K` V`)`$] = K$

$\mathcal{MCPS}$: Exp $\to$ Meta-Continuation $\to$ Exp

$\mathcal{MCPS}[\![I]\!]$ = $[\lambda m\,.\,[m\ I]]$

$\mathcal{MCPS}[\![L]\!]$ = $[\lambda m\,.\,[m\ L]]$

$\mathcal{MCPS}[\![$`(LAMBDA (`I_1 `...` I_n`) `E`)`$]\!]$
= $[\lambda m\,.\,[m\ $ `(LAMBDA (`I_1 `...` I_n `.K`i`.)`
$[\mathcal{MCPS}[\![E]\!]\ [exp \to meta\text{-}cont$ `.K`i`.`$]]$`)`$]]$

$\mathcal{MCPS}[\![$`(CALL `E_1 E_2`)`$]\!]$
= $[\lambda m\,.\,[\mathcal{MCPS}[\![E_1]\!]\ [\lambda v_1\,.$
$[\mathcal{MCPS}[\![E_2]\!]\ [\lambda v_2\,.$
`(CALL` v_1 v_2 $[meta\text{-}cont \to exp\ \ m]$`)`$]]]]]$

$\mathcal{MCPS}[\![$`(PRIMOP `P E_1 E_2`)`$]\!]$
= $[\lambda m\,.\,[\mathcal{MCPS}[\![E_1]\!]\ [\lambda v_1\,.$
$[\mathcal{MCPS}[\![E_2]\!]\ [\lambda v_2\,.$
`(LET ((.T`i`. (PRIMOP `P v_1 v_2`)))`
$[m\ \ $ `.T`i`.`$])]]]]$

$\mathcal{MCPS}[\![$`(IF `E_c E_t E_f`)`$]\!]$
= $[\lambda m\,.\,[\mathcal{MCPS}[\![E_c]\!]\ [\lambda v_1\,.$
`(LET ((K` $[meta\text{-}cont \to exp\ m]$`))`
`(IF` v_1
$[\mathcal{MCPS}[\![E_t]\!]\ [exp \to meta\text{-}cont$ `K`$]]$
$[\mathcal{MCPS}[\![E_f]\!]\ [exp \to meta\text{-}cont$ `K`$]]$`))`$]]]$
$\mathcal{MCPS}[\![$`(LET ((`I E_{def}`)) `E_{body}`)`$]\!]$
= $[\lambda m\,.\,[\mathcal{MCPS}[\![E_{\text{def}}]\!]\ [\lambda v\,.$
`(LET ((I` v`))` $[\mathcal{MCPS}[\![E_{\text{body}}]\!]\ m]$`)`$]]]$

Appendix F: Match Desugaring

$\mathcal{D}[\![$`(match` E `(`P_1 E_1`) ... (`P_n E_n`))`$]\!]$ =
`(let ((`I_{top} E`))`
$\mathcal{D}_{\text{clauseseq}}[\![[P_1, \ldots, P_n],\ [E_1, \ldots, E_n],\ I_{\text{top}},$ `(lambda () (error "No pattern matches!"))`$]\!]$)

$\mathcal{D}_{\text{clauseseq}}[\![[\,],\ [\,],\ V,\ F]\!]$ = `(`F`)`
$\mathcal{D}_{\text{clauseseq}}[\![P_1 \,.\, P_{\text{rest}},\ E_1 \,.\, E_{\text{rest}},\ V,\ F]\!]$ =
`(let ((`I_{fail} `(lambda () ;` If P_1 doesn't match, try the other clauses
$\mathcal{D}_{\text{clauseseq}}[\![P_{\text{rest}},\ E_{\text{rest}},\ V,\ F]\!]$`)))`
$\mathcal{D}_{\text{pat}}[\![P_1,\ V,\ E_1,\ I_{\text{fail}}]\!]$`)`

$\mathcal{D}_{\text{pat}}[\![L,\ V,\ S,\ F]\!]$ = `(if (equal?` $_{LIT}$ L V`)` S `(`F`))`
$\mathcal{D}_{\text{pat}}[\![\_,\ V,\ S,\ F]\!] = S$
$\mathcal{D}_{\text{pat}}[\![I,\ V,\ S,\ F]\!]$ = `(let ((`I V`))` S`)`
$\mathcal{D}_{\text{pat}}[\![$`(`$I$ P_1 `...` P_n`)`$,\ V,\ S,\ F]\!]$ =
`(`I`~` V
`(lambda (`I_1 `...` I_n`) ;` Match the object's component parts
$\mathcal{D}_{\text{patseq}}[\![[P_1, \ldots, P_n],\ [I_1, \ldots, I_n],\ S,\ F]\!]$`)`
F`)`

$\mathcal{D}_{\text{patseq}}[\![[\,],\ [\,],\ S,\ F]\!] = S$
$\mathcal{D}_{\text{patseq}}[\![P_1 \,.\, P_{\text{rest}},\ I_1 \,.\, I_{\text{rest}},\ S,\ F]\!]$ =
$\mathcal{D}_{\text{pat}}[\![P_1,\ I_1,$
$\mathcal{D}_{\text{patseq}}[\![P_{\text{rest}},\ I_{\text{rest}},\ S,\ F]\!]$, ; If P_1 matches, continue trying to match the rest
$F]\!]$