MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

2000 Final Examination and Solutions

1. Final Examination

There are *four* 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 *ninety minutes* 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

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: Parameter Passing [21 points]

Give the meaning of the following FLAVAR! expression under each parameter passing scheme. Hint: try to figure out the values of `(begin (f a) a)` and `(f (begin (set! b (+ b 2)) b))` separately, then find the sum.

```
(let ((a 4) (b 0))
  (let ((f (lambda (x)
             (begin (set! x (* x x))
                    (/ x 2)))))
    (+ (begin (f a) a)
       (f (begin (set! b (+ b 2)) b)))))
```

a. [7 points] call-by-value

b. [7 points] call-by-name

c. [7 points] call-by-reference

Problem 2: Explicit Types [24 points]

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

Louis Reasoner likes both dynamic scoping and explicit types, and thus decides to create a new language, Scheme/DX, that includes both! However, certain problems arise and you are called into rescue Louis' attempt.

Louis revised a procedure definition to be:

E := ... | (lambda (((I_1 T_1) ... (I_n T_n)) ((I'_1 T'_1) ... (I'_m T'_m))) E_B)

with the new type:

T := ... | (-> ((T_1 ... T_n) ((I'_1 T'_1) ... (I'_m T'_m))) T_B)

The first list of identifiers $\{I_i\}$ and types $\{T_i\}$ in LAMBDA specifies the formal parameters to LAMBDA, and the second list of identifiers $\{I'_i\}$ and types $\{T'_i\}$ specifies all of the dynamically bound identifiers used by E and their types. Thus when a procedure is called, the types of BOTH the actual parameters and the dynamically bound variables must match.

For example:

```
(let ((x 1))
  (let ((p (lambda (((y int)) ((x bool))) (if x y 0))))
     (let ((x #t))
        (p 1))))
```

⇒ *1*

```
(let ((x #t))
  (let ((p (lambda (((y int)) ((x bool))) (if x y 0))))
     (let ((x 1))
        (p 1))))
```

⇒ *NOT WELL TYPED*

For an expression E, let S be the set of dynamically bound identifiers in E. We can extend our typing framework to be

$A \vdash E : T \,@\, S$

In this framework, "@" means "E uses dynamic variables" just like ":" means "has type".

Our new combined typing and dynamic variable rule for identifiers is:

$A[I : T] \vdash I : T \,@\, \{I\}$

Here are two examples to give you an idea of what we mean:

$A[x : \texttt{int}] \vdash$ (+ 1 x) : $\texttt{int} \,@\, \{x\}$

$A[x : \texttt{int}] \vdash$ (let ((x 1)) (+ 1 x)) : $\texttt{int} \,@\, \{\}$

In this framework:

a. [6 points] Give a combined typing and dynamic variable rule for LET.

b. [6 points] Give a combined typing and dynamic variable rule for LAMBDA.

c. [6 points] Give a combined typing and dynamic variable rule for application.

d. [6 points] Briefly argue that your rules always guarantee that in well-typed programs references to dynamic variables are bound.

Problem 3: Type Reconstruction [30 points]

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

Ben Bitdiddle is at it again, further enhancing Scheme/R. In this new and improved version he has added a new construct called go that executes all of its constituent expressions $E1...En$ in parallel:

$$\texttt{E := ... | (go (}I_1 \texttt{ ... } I_n\texttt{) } E_1 \texttt{ ... } E_m\texttt{) | (talk! } I\ E\texttt{) | (listen } I\texttt{)}$$

go terminates when all of $E_1 \ldots E_m$ terminate, and it returns the value of E_1. go includes the ability to use communication variables $I_1 \ldots I_n$ in a parallel computation. A communication variable can be assigned a value by `talk!`. An expression in go can wait for a communication variable to be given a value with `listen`. `listen` returns the value of the variable once it is set with `talk!`. For a program to be well typed, all $E_1 \ldots E_n$ in go must be well typed.

Communication variables will have the unique type `(commof T)` where `T` is the type of value they hold. This will ensure that only communication variables can be used with `talk!` and `listen`, and that communication variables can not be used in any other expression.

Ben has given you the Scheme/R typing rules for `talk!` and `listen`:

$$\frac{\begin{array}{c} A \vdash E : \texttt{T} \\ A \vdash I : \texttt{(commof T)} \end{array}}{A \vdash \texttt{(talk! } I\ E\texttt{)} : \texttt{unit}} \qquad [\textit{talk!}]$$

$$\frac{A \vdash I : \texttt{(commof T)}}{A \vdash \texttt{(listen } I\texttt{)} : \texttt{T}} \qquad [\textit{listen}]$$

a. [8 points] Give the Scheme/R typing rule for go.

b. [7 points] Give the Scheme/R reconstruction algorithm for `talk!`.

c. [7 points] Give the Scheme/R reconstruction algorithm for `listen`.

d. [8 points] Give the Scheme/R reconstruction algorithm for go.

Problem 4: Pragmatics [25 points]

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

This problem contains two independent parts:

a. Ben Bitdiddle, the engineer in charge of the MCPS phase in the Tortoise compiler, looked over the book and the previous years' finals and couldn't find the meta-cps rule for `label` and `jump`. As Ben is very rushed – the new Tortoise compiler should hit the market in the middle of the holiday season – he's asking for your help.

   Here is a quick reminder of the semantics of `label` and `jump`:

   (`label` I E**)** evaluates E; inside E, I is bound to the continuation of (`label` I E). The labels are statically scoped (as the normal Scheme variables are).

   (`jump` E_1 E_2**)** calls the continuation resulted from evaluating E_1, passing to it the result of evaluating E_2. E_1 should evaluate to a label (i.e. a continuation introduced by `label`). The behavior of (`jump` E_1 E_2) is unspecified if E_1 doesn't evaluate to a label (this is considered to be a programming error).

   E.g.: The expression `(label foo (+ 1 (jump foo (+ 2 (jump foo 3)))))` should evaluate to 3.

   Ben even wrote the SCPS rules for `label` and `jump`:

$$\mathcal{SCPS}[\![\texttt{(label } I\ E\texttt{)}]\!] = \texttt{(lambda (k) (let ((I k)) (call } \mathcal{SCPS}[\![E]\!] \texttt{ k)))}$$

$$\mathcal{SCPS}[\![\texttt{(jump } E_1\ E_2\texttt{)}]\!] = \texttt{(lambda (k1) (call } \mathcal{SCPS}[\![E_1]\!] \texttt{ (lambda (k2) (call } \mathcal{SCPS}[\![E_2]\!] \texttt{ k2))))}$$

   (i) [10 points] What is $\mathcal{MCPS}[\![\texttt{(LABEL } I\ E\texttt{)}]\!]$? Be careful to avoid code bloat.

   (ii) [10 points] What is $\mathcal{MCPS}[\![\texttt{(JUMP } E_1\ E_2\texttt{)}]\!]$?

b. [5 points] In class, we've mentioned a couple of times that type safety is impossible without automatic memory management (i.e. garbage collection). Please explain why this is true.

Appendix A: Standard Semantics of FLK!

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

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

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

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

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

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

$$
\begin{array}{l}
\textit{same-location?} : \text{Location} \rightarrow \text{Location} \rightarrow \text{Bool} = \lambda l_1 l_2 \,.\, (l_1 =_{\text{Nat}} l_2) \\
\textit{next-location} : \text{Location} \rightarrow \text{Location} = \lambda l \,.\, (l +_{\text{Nat}} 1) \\[4pt]
\textit{empty-store} : \text{Store} = \lambda l \,.\, (\text{Unassigned} \mapsto \text{Assignment } \textit{unassigned}) \\[4pt]
\textit{fetch} : \text{Location} \rightarrow \text{Store} \rightarrow \text{Assignment} = \lambda l s \,.\, (s \ l) \\[4pt]
\textit{assign} : \text{Location} \rightarrow \text{Storable} \rightarrow \text{Store} \rightarrow \text{Store} \\
\quad = \lambda l_1 \sigma s \,.\, \lambda l_2 \,.\, \textbf{if } (\textit{same-location? } l_1 \ l_2) \\
\qquad\qquad\qquad \textbf{then } (\text{Storable} \mapsto \text{Assignment } \sigma) \\
\qquad\qquad\qquad \textbf{else } (\textit{fetch } l_2 \ s) \\[4pt]
\textit{fresh-loc} : \text{Store} \rightarrow \text{Location} = \lambda s \,.\, (\textit{first-fresh } s \ 0) \\[4pt]
\textit{first-fresh} : \text{Store} \rightarrow \text{Location} \rightarrow \text{Location} \\
\quad = \lambda s l \,.\, \textbf{matching } (\textit{fetch } l \ s) \\
\qquad\qquad \triangleright (\text{Unassigned} \mapsto \text{Assignment } \textit{unassigned}) \, [\!] \, l \\
\qquad\qquad \triangleright \textbf{ else } (\textit{first-fresh } s \ (\textit{next-location } l)) \\
\qquad\qquad \textbf{endmatching} \\[4pt]
\textit{lookup} : \text{Environment} \rightarrow \text{Identifier} \rightarrow \text{Binding} = \lambda e I \,.\, (e \ I)
\end{array}
$$

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*

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

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

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

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

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

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

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

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

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

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

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

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

$$\textit{extract-value} : \text{Expressible} \to \text{Binding} \\ = \lambda a\,.\ \textbf{matching}\ \text{a} \\ \triangleright (\text{Value} \mapsto \text{Expressible}\ v)\ [\!]\ (\text{Denotable} \mapsto \text{Binding}\ v) \\ \triangleright\ \textbf{else}\ \bot_{\text{Binding}} \\ \textbf{endmatching}$$

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

Appendix B: Parameter Passing Semantics for FLAVAR!

$\sigma \in \text{Storable} = \text{Value}$

$$\mathit{val\text{-}to\text{-}storable} = \lambda v \,.\, v$$

$$\begin{aligned}\mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e \,.\, & (\mathit{with\text{-}procedure}\ \ (\mathcal{E}[\![E_1]\!]\ \ e) \\ & \quad (\lambda p \,.\, (\mathit{with\text{-}value}\ \ (\mathcal{E}[\![E_2]\!]\ \ e) \\ & \qquad (\lambda v \,.\, (\mathit{allocating}\ \ v\ \ p)))))\end{aligned}$$

$$\mathcal{E}[\![I]\!] = \lambda e \,.\, (\mathit{with\text{-}denotable}\ \ (\mathit{lookup}\ \ e\ \ I)\ \ (\lambda l \,.\, (\mathit{fetching}\ \ l\ \ \mathit{val\text{-}to\text{-}comp})))$$

Call-by-Value

$\sigma \in \text{Storable} = \text{Computation}$

$$\mathit{val\text{-}to\text{-}storable} = \mathit{val\text{-}to\text{-}comp}$$

$$\begin{aligned}\mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e \,.\, & (\mathit{with\text{-}procedure}\ \ (\mathcal{E}[\![E_1]\!]\ \ e) \\ & \quad (\lambda p \,.\, (\mathit{allocating}\ \ (\mathcal{E}[\![E_2]\!]\ \ e)\ \ p)))\end{aligned}$$

$$\mathcal{E}[\![I]\!] = \lambda e \,.\, (\mathit{with\text{-}denotable}\ \ (\mathit{lookup}\ \ e\ \ I)\ \ (\lambda l \,.\, (\mathit{fetching}\ \ l\ \ (\lambda c \,.\, c))))$$

Call-by-Name

Figure 4: Parameter passing mechanisms in FLAVAR!, part I.

$$\sigma \in \text{Storable} = \text{Memo}$$
$$m \in \text{Memo} = \text{Computation} + \text{Value}$$

$$\textit{val-to-storable} = \lambda v\,.\,(\text{Value} \mapsto \text{Memo}\ \ v)$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ \ E_2\texttt{)}]\!] = \lambda e\,.\ (\textit{with-procedure}\ \ (\mathcal{E}[\![E_1]\!]\ \ e)$$
$$(\lambda p\,.\ (\textit{allocating}\ \ (\text{Computation} \mapsto \text{Memo}\ \ (\mathcal{E}[\![E_2]\!]\ \ e))\ \ p)))$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\ (\textit{with-denotable}\ \ (\textit{lookup}\ \ e\ \ I)$$
$$(\lambda l\,.\ (\textit{fetching}\ \ l$$
$$(\lambda m\,.\ \ \textbf{matching}\ m$$
$$\triangleright\ (\text{Computation} \mapsto \text{Memo}\ \ c)$$
$$[\!]\ \ (\textit{with-value}\ c$$
$$(\lambda v\,.\ \ (\textit{sequence}\ \ (\textit{update}\ \ l\ \ (\text{Value} \mapsto \text{Memo}\ \ v))$$
$$(\textit{val-to-comp}\ \ v))))$$
$$\triangleright\ (\text{Value} \mapsto \text{Memo}\ \ v)\ [\!]\ \ (\textit{val-to-comp}\ \ v)$$
$$\textbf{endmatching}\ ))))$$

Call-by-Need (Lazy Evaluation)

$$\sigma \in \text{Storable} = \text{Value}$$

$$\mathcal{E} : \text{Exp} \rightarrow \text{Environment} \rightarrow \text{Computation}$$
$$\mathcal{LV} : \text{Exp} \rightarrow \text{Environment} \rightarrow \text{Computation}$$

$$\textit{val-to-storable} = \lambda v\,.\,v$$

$$\mathcal{E}[\![\texttt{(call}\ E_1\ \ E_2\texttt{)}]\!] = \lambda e\,.\ (\textit{with-procedure}\ \ (\mathcal{E}[\![E_1]\!]\ \ e)$$
$$(\lambda p\,.\ (\textit{with-location}\ \ (\mathcal{LV}[\![E_2]\!]\ \ e)\ \ p)))$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\ (\textit{with-denotable}\ \ (\textit{lookup}\ \ e\ \ I)\ \ (\lambda l\,.\,(\textit{fetching}\ \ l\ \ \textit{val-to-comp})))$$

$$\mathcal{LV}[\![I]\!] = \lambda e\,.\ (\textit{with-denotable}\ \ (\textit{lookup}\ \ e\ \ I)\ \ (\lambda l\,.\,(\textit{val-to-comp}\ \ (\text{Location} \mapsto \text{Value}\ \ l))))$$

$\mathcal{LV}[\![E_{other}]\!]$; where E_{other} is not I
$$= \lambda e\,.\ (\textit{with-value}\ \ (\mathcal{E}[\![E_{other}]\!]\ \ e)$$
$$(\lambda v\,.\ (\textit{allocating}\ \ v\ \ (\lambda l\,.\,(\textit{val-to-comp}\ \ (\text{Location} \mapsto \text{Value}\ \ l)))))$$

Call-by-Reference

Figure 5: Parameter passing mechanisms in FLAVAR!, part II.

Appendix C: Typing Rules for SCHEME/XSP

SCHEME/X Rules

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

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

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

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

$$A[I\!:\!T] \vdash I : T \qquad [var]$$

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

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

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

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

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

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

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

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

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

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

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

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

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

Rules Introduced by SCHEME/XS to Handle Subtyping

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

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

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

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

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

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

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

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

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

Rules Introduced by SCHEME/XSP to Handle Polymorphism

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

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

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

recof Equivalence

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

Appendix D: Typing Rules for SCHEME/R

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix E: 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)

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

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

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

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

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

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

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

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

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

Appendix F: Simple-CPS Conversion Rules

```
SCPS⟦I⟧ = (lambda (k) (call k I))

SCPS⟦L⟧ = (lambda (k) (call k L))

SCPS⟦(lambda (I) E)⟧ = (lambda (k)
                          (call k
                            (lambda (I k-call)
                              (call SCPS⟦E⟧ k-call))))

SCPS⟦(call E1 E2)⟧ = (lambda (k)
                        (call SCPS⟦E1⟧
                          (lambda (v1)
                            (call SCPS⟦E2⟧
                              (lambda (v2)
                                (call v1 v2 k))))))

SCPS⟦(let ((I1 E1) ... (I2 E2)) E)⟧ = (lambda (k)
                                         (call SCPS⟦E1⟧
                                           (lambda (I1)
                                             ⋱
                                               (call SCPS⟦En⟧
                                                 (lambda (In)
                                                   (call SCPS⟦E⟧ k)))...)))

SCPS⟦(label I E)⟧ = (lambda (k)
                       (let ((I k))
                           (call SCPS⟦E⟧ k)))

SCPS⟦(jump E1 E2)⟧ = (lambda (k1)
                        (call SCPS⟦E1⟧
                              (lambda (k2)
                                (call SCPS⟦E2⟧ k2))))
```

Appendix G: 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.

$$\begin{aligned} \text{E} &\in \text{Exp} \\ \text{m} &\in \text{Meta-Continuation} = \text{Exp} \to \text{Exp} \end{aligned}$$

$\textit{meta-cont} \to \textit{exp}$: (Exp → Exp) → Exp = $[\lambda m\,.\,\texttt{(LAMBDA (t)}[m\ \texttt{t}]\texttt{)}]$
$\textit{exp} \to \textit{meta-cont}$: Exp → (Exp → Exp) = $[\lambda E\,.\,[\lambda V\,.\,\texttt{(CALL}\ E\ \ V\texttt{)}]]$

meta-cont→exp $[\lambda V\,.\,\texttt{(CALL K}\ V\texttt{)}] = K$

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

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

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

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

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

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

$\mathcal{MCPS}[\![\texttt{(IF}\ E_c\ \ E_t\ \ E_f\texttt{)}]\!]$
$= [\lambda m\,.\,[\mathcal{MCPS}[\![E_c]\!]\ \ [\lambda v_1\,.$
$\quad \texttt{(LET ((K}\ [\textit{meta-cont} \to \textit{exp}\ \ m]\texttt{))}$
$\quad\quad \texttt{(IF}\ v_1$
$\quad\quad\quad [\mathcal{MCPS}[\![E_t]\!]\ \ [\textit{exp} \to \textit{meta-cont}\ \ \texttt{K}]]$
$\quad\quad\quad [\mathcal{MCPS}[\![E_f]\!]\ \ [\textit{exp} \to \textit{meta-cont}\ \ \texttt{K}]]\texttt{))}]]]$

$\mathcal{MCPS}[\![\texttt{(LET ((}I\ \ E_{\text{def}}\texttt{))}\ E_{\text{body}}\texttt{)}]\!]$
$= [\lambda m\,.\,[\mathcal{MCPS}[\![E_{\text{def}}]\!]\ \ [\lambda v\,.$
$\quad \texttt{(LET ((I}\ \ v\texttt{))}\ \ [\mathcal{MCPS}[\![E_{\text{body}}]\!]\ \ m]\texttt{)}]]]$

2. Final Examination Solutions

Problem 1: Parameter Passing

a. 6

b. 8

c. 18

Problem 2: Explicit Types

a.

$$\frac{\begin{array}{c}\forall i\ (A \vdash E_i : T_i \ \ @\ \ S_i) \\ A[I_1 {:} T_1\texttt{,}\ \ \ldots\texttt{,}\ \ I_n {:} T_n] \vdash E_B : T_B \ \ @\ \ S_B\end{array}}{A \vdash \texttt{(let ((}I_1\ \ E_1\texttt{)}\ \ldots\ \texttt{(}I_n\ \ E_n\texttt{))}\ E_B\texttt{)} : T_B \ \ @\ \ S_1 \cup\ \ldots\ \cup S_n \cup S_B - \{I_1\ \ \ldots\ \ I_n\}} \quad [let]$$

b.

$$\frac{A[I_1 {:} T_1\texttt{,}\ \ \ldots\texttt{,}\ \ I_n {:} T_n\texttt{,}\ \ I'_1 : T'_1\texttt{,}\ \ \ldots\texttt{,}\ \ I'_m : T'_m] \vdash E_B : T_B \ \ @\ \ S \quad S \subset \{I_1\ \ \ldots\ \ I_n,\ \ I'_1\ \ \ldots\ \ I'_m\}}{\begin{array}{c}A \vdash \texttt{(lambda (((}I_1\ \ T_1\texttt{)}\ \ldots\ \texttt{(}I_n\ \ T_n\texttt{))}\ \texttt{((}I'_1\ \ T'_1\texttt{)}\ \ldots\ \texttt{(}I'_m\ \ T'_m\texttt{)))}\ E_B\texttt{)} : \\ \texttt{(-> ((}T_1\ \ldots\ T_n\texttt{)}\ \texttt{((}I'_1\ \ T'_1\texttt{)}\ \ldots\ \texttt{(}I'_m\ \ T'_m\texttt{)))}\ T_B\texttt{)}\ \ @\ \ \{\}\end{array}} \quad [\lambda]$$

c.

$$\frac{\begin{array}{c}A \vdash E_P : \texttt{(-> ((}T_1\ \ldots\ T_n\texttt{)}\ \texttt{((}I'_1\ \ T'_1\texttt{)}\ \ldots\ \texttt{(}I'_m\ \ T'_m\texttt{)))}\ T_B\texttt{)}\ \ @\ \ S_P \\ \forall i\ .\ (A \vdash E_i : T'_i\ \ @\ \ S_i)\ \wedge\ \forall j\ .\ (A[I'_j] = T'_j)\end{array}}{A \vdash \texttt{(}E_P\ \ E_1\ \ \ldots\ \ E_n\texttt{)} : T_B\ \ @\ \ S_1 \cup \ldots \cup S_n \cup S_P \cup \{I'_1\ \ \ldots\ \ I'_m\}} \quad [call]$$

d. The [call] rule guarantees that all dynamic variables needed in the procedure are bound. The expression $(A[I'_j] = T'_j)$ will produce a type error if any I'_j is not bound. In addition, the [λ] rule guarantees that every dynamic variable used in the body of a procedure is properly declared.

Problem 3: Type Reconstruction

a.

$$\frac{\forall i\ .\ (A[I_1 {:} \texttt{(commof}\ \ T'_1\texttt{)},\ \ \ldots I_n {:} \texttt{(commof}\ \ T'_n\texttt{)}] \vdash E_i : T_i)}{A \vdash \texttt{(go}\ \ \texttt{(}I_1\ \ \ldots I_n\texttt{)}\ \ E_1\ \ \ldots E_n\texttt{)} : T_1} \quad [go]$$

b. $R[\![\texttt{(talk!}\ I\ \ E\texttt{)}]\!]\ A\ S =$
$$\begin{array}{l}\textbf{let}\ \langle T,\ S_1\rangle = R[\![I]\!]\ A\ S \\ \textbf{in}\ \ \textbf{let}\ \langle T',\ S_2\rangle = R[\![E]\!]\ A\ S \\ \quad\ \ \textbf{in}\ \ \textbf{let}\ S_3 = U(T,\ (\mathit{commof}\ T'),\ S_2) \\ \quad\quad\quad \textbf{in}\ \ \langle \mathit{unit},\ S_3\rangle\end{array}$$

c. $R[\![\texttt{(listen}\ I\texttt{)}]\!]\ A\ S =$
$$\begin{array}{l}\textbf{let}\ \langle T,\ S_1\rangle = R[\![I]\!]\ A\ S \\ \textbf{in}\ \ \textbf{let}\ S_2 = U((\mathit{commof}\ ?t),\ T,\ S_1) \\ \quad\ \ \textbf{in}\ \ \langle ?t,\ S_2\rangle\end{array}$$

d. $R[\![$(go (I_1 ... I_n) E_1 ... E_m)$]\!]\ A\ S =$ **let** $A_1 = A[I_1 : (commof\ ?t_1) \ldots I_n : (commof\ ?t_n)]$
in **let** $\langle T_1,\ S_1\rangle = R[\![E_1]\!]\ A_1\ S$
in ...
let $\langle T_m,\ S_m\rangle = R[\![E_m]\!]\ A_1\ S_{m-1}$
in $\langle T_1,\ S_m\rangle$

where $?t_i \ldots ?t_n$ are fresh.

Problem 4: Pragmatics

a. (i) $\mathcal{MCPS}[\![$(LABEL I E)$]\!]$
= $[\lambda m$. (LET ((I $[meta\text{-}cont \rightarrow exp\ \ m]$))
$[\mathcal{MCPS}[\![E]\!]$ $[\lambda v$. (CALL I v)$]]$)$]$

I is lexically bound to $[meta\text{-}cont \rightarrow exp\ \ m]$. In the last line, we could have put m instead of $[\lambda v$. (CALL I v) but this would lead to an exponential increase in the code size.

(ii) $\mathcal{MCPS}[\![$(JUMP 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) $]]]]]$

Very similar to the rule for CALL. However, this time we totally ignore m as required by the semantics of jump.

b. If we can explictly free memory, then it would be possible to free a block of memory orginally containing data of type T, then allocating it to data containing T′, thus resulting in a type error when an expression gets a T′ instead of a T.