

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
Department of Electrical Engineering and Computer Science

## Final Examination 2002

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 [29 points]	2
Problem 2: Probabilistic Types [12 points]	3
Problem 3: Strictness Effects [17 points]	4
Problem 4: Nosy FLK! [25 points]	5
Problem 5: Pragmatics [17 points]	7
Appendix A: Standard Semantics of FLK!	9
Appendix B: Typing Rules for SCHEME/XSP	12
Appendix C: Typing Rules for SCHEME/R	14
Appendix D: Type Reconstruction Algorithm for SCHEME/R	15

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 [29 points]

- a. [4 points] Give a desugaring from a FLAVAR! program  $P$  to a FLAVAR! program  $P'$  that has the following property:  $P$  evaluated with call-by-value semantics has the same behavior as  $P'$  when evaluated with call-by-name semantics.
- b. [3 points] Give two domains  $D$  and  $E$  such that the number of set-theoretic functions from  $D$  to  $E$  is infinite, but the number of continuous functions is finite.
- c. [3 points] Consider a new construct `smart-compose` to FL/R. (`smart-compose f g`) has the following behavior: it evaluates to whichever of `(compose f g)` or `(compose g f)` type checks. If both type-check, it chooses one of them arbitrarily. Why could type reconstruction using Appendix D fail even when a type for `smart-compose` is reconstructible?
- d. [2 points] What is a broken heart, and what is it used for?
- e. [2 points] Give the reconstructed type of the following expression.

```
(lambda (f g)
  (f (f g 2)
      (g 2)))
```

- f. [2 points] What is the purpose of closure conversion?
- g. [2 points] Consider the following FLAVAR! expression

```
(let ((x 3)
      (foo (lambda (q) (+ x y))))
  (letrec ((y x)
           (z (let ((x 15)) (lambda (q) (z q)))))
    (let ((x 16)
          (y x))
      (foo z))))
```

- (i) [1 points] What is its value, or what error occurs, under call-by-value static scoping?
  - (ii) [1 points] What is its value, or what error occurs, its value under call-by-value dynamic scoping?
- h. [2 points] Give the value of the following expression, or state what error occurs.

```
(let ((x (lambda (x) (+ x 1))))
  (let ((y (lambda (y) (y x))))
    (let ((y (let ((y y)) y))
          (let ((x (y (lambda (x) x))))
            (x 5)))))
```

- (i) [1 points] When evaluated under call-by-value static scoping.
  - (ii) [1 points] When evaluated under call-by-value dynamic scoping.
- i. [3 points] Louis Reasoner has an application in which at least half of memory is garbage each time the garbage collector is invoked. He concludes that the use of two spaces is wasteful: he can modify the garbage collector to work in place. Explain why Louis's proposal does not work.
- j. [6 points] Give a type reconstruction algorithm rule for "call with current continuation" (`cwcc`). You may find referring to Appendix D helpful.

Here is an example of `cwcc`:

```
(+ (cwcc (lambda (exit) 100))
   (cwcc (lambda (exit) (+ 2 (exit 4)))))
⇒ 104
```

## Problem 2: Probabilistic Types [12 points]

Sensing he won't have time to finish his SCHEME/R killer app before Christmas, Ben Bitdiddle buys a subroutine from Subroutines 'R' Us. While Ben's program was completely deterministic, the subroutine he bought is randomized. The fine print states that the subroutine returns the correct answer  $99\frac{44}{100}\%$  of the time. Ben worries about the correctness of his program, since he uses this subroutine many times. But before he gets to answer his own question, his mind wanders off to another subroutine he purchased.

To deal with this problem, you will create a type/effect system. The new type system has judgments like the following:

$$A \vdash E : T \% P,$$

which is pronounced, "in the type environment  $A$ , expression  $E$  has type  $T$  and probability of correctness at least  $P$ ." The types in this new type system are all inherited from SCHEME/R, with one difference: a function type now contains a probability of correctness. We write this as  $(\rightarrow p (T_1 \dots T_n) T_b)$ , which means that when this procedure is applied to arguments  $a_1 \dots a_n$ , it evaluates to the correct value with probability  $p$ .

SCHEME/R always operates correctly and deterministically; however, it is possible to code randomized algorithms using it. When you use a randomized algorithm, you add a probabilistic type for it (specified by the manufacturer) to the typing environment; then, you use the type/effect system to compute the probability that your whole program operates correctly.

The following type/effect judgments

$$\begin{aligned} \vdash N : \text{int} \% 1 \\ \vdash B : \text{bool} \% 1 \end{aligned}$$

indicate that numeric and boolean literals are guaranteed to be correct. You should assume that if any subexpression evaluates incorrectly, then the whole program evaluates incorrectly: subsequent wrong answers never cancel out previous ones.

- [3 points] Give a type/effect rule for `if`.
- [3 points] Give a type/effect rule for `lambda`.
- [3 points] Give a type/effect rule for application.
- [3 points] Give a subtyping rule for procedures by revising the  $\rightarrow\sqsubseteq$  rule of Appendix B (page 13).

### Problem 3: Strictness Effects [17 points]

As presented in class, effect systems indicate what side effects may result from evaluation of an expression. In a lazy language, a natural extension to this notion is whether a particular argument to a procedure is ever evaluated.

In this problem, you will write type reconstruction rules that determine whether a particular expression evaluates a given argument. There are three possibilities for each argument: the procedure may always evaluate the argument, the procedure may never evaluate the argument, or the procedure may sometimes evaluate the argument; we represent these three values as  $A$  (for always),  $N$  (for never), and  $M$  (for maybe).

Typing judgments have the form

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

where  $S$  maps variables to strictness values. (You may assume that  $S$  always contains an entry for every variable that is ever used in the program, and furthermore that all variables are uniquely named. In other words, you don't have to worry about scoping and variable capture.)

For example, a reasonable judgment in a program with 4 variables  $a, b, c,$  and  $d$  might be

$$A \vdash E : T \& \{a : M, b : A, c : N, d : M\}.$$

and a reasonable typing rule might be

$$\frac{A \vdash E_1 : T \& S_1 \quad A \vdash E_2 : T \& S_2}{A \vdash (\text{strange } E_1 E_2) : T \& S_1 \quad \text{AtoN}(S_2)} \quad [\text{strange}]$$

where  $\&$  is taken elementwise and  $\text{AtoN}$  is defined below.

Procedure types have the following form:  $(\rightarrow \text{Ids } S (T_1 \dots T_n) T)$ . "Ids" is a list of the procedure's parameter names.  $S$  is the strictness of the procedure (over all program variables, as usual).

Your task is to extend the typing rules of Appendix C to also compute a strictness environment.

You may find one or more of the following auxiliary procedures to be helpful. However, you are not permitted to cascade them (nor do you need to!); don't write, for example, " $\text{AtoM}(\text{NtoA}(S))$ ".

$\text{allA}$  returns a new mapping in which every variable is bound to  $A$ .

$\text{allN}$  returns a new mapping in which every variable is bound to  $N$ .

$\text{allM}$  returns a new mapping in which every variable is bound to  $M$ .

$\text{AtoN}$  returns a new mapping in which bindings to  $A$  have been replaced by  $N$ .

$\text{AtoM}$  returns a new mapping in which bindings to  $A$  have been replaced by  $M$ .

$\text{NtoA}$  returns a new mapping in which bindings to  $N$  have been replaced by  $A$ .

$\text{NtoM}$  returns a new mapping in which bindings to  $N$  have been replaced by  $M$ .

$\text{MtoA}$  returns a new mapping in which bindings to  $M$  have been replaced by  $A$ .

$\text{MtoN}$  returns a new mapping in which bindings to  $M$  have been replaced by  $N$ .

For example,

$$\text{MtoN}(\{a : M, b : A, c : N, d : M\}) = \{a : N, b : A, c : N, d : N\}.$$

- [2 points] Draw the domain whose elements are  $\{A, N, M\}$ : draw the elements, and draw a line for each non-transitive  $\sqsubset$  relationship, with the greater element higher on the page. (Hint: No element is greater than  $A$ .)
- [4 points] Give the typing rule for `if`.
- [4 points] Give the typing rule for `lambda`.
- [4 points] Give the typing rule for application.
- [3 points] Give the typing rule for (application of) `+`.

## Problem 4: Nosy FLK! [25 points]

The Department of Homeland Security has adopted FLK! as its official programming language, for two reasons. First, FLK! has no foreign function interface, and the DHS disapproves of anything foreign. Second, and more importantly, FLK! has no “private” declaration. The DHS reasons that programmers who have done nothing wrong should have nothing to hide, and the DHS insists on access to any data at any time for any purpose. Imagine the DHS’s disappointment when they realize that shadowing can make a variable binding inaccessible! For example, consider the following code that you might wish to execute:

```
;; Send a love letter
(let ((letter "Dear sweetheart, ... With love, Snookums"))
  (let ((mail-agent (mail-sending-program letter)))
    (let ((letter "s")) ; The "s" key sends a message
      ;; DHS INSERTS CODE HERE
      (apply-keystroke letter mail-agent))))
```

It is a serious problem that, at the specified point in the program, spy agencies cannot read the text of your letter. Accordingly, DHS hires Ben Bitdiddle to add a new construct, `up-env`, to FLK!; the new language is called “Nosy FLK!”. `up-env` permits access to shadowed variables. It takes a number  $n$  and an identifier, and it looks up the identifier in the  $n$ th parent environment of the current one. For example, consider the following DHS-inserted code:

```
(up-env 0 letter)  equivalent to letter; returns "s"
(up-env 1 letter)  returns "Dear sweetheart ...", which is the value of letter
                   in the parent environment (in which mail-agent is also bound)
(up-env 2 letter)  also returns "Dear sweetheart ...", which is the value of letter
                   in the outermost environment
(up-env 3 letter)  issues an “unbound identifier” error
```

With the work only half-finished, DHS’s hidden microphones overhear Ben Bitdiddle talking about “sexps,” and he is taken away as a suspected pornographer. In order to demonstrate your patriotism, you agree to complete work on this language extension.

- a. [4 points] Consider the following code. (We use syntactic sugar such as `let` for convenience and clarity.)

```
(let ((n 11))
  (letrec ((fact (let ((n 9))
                  (lambda (n)
                    (print (up-env 1 n))
                    (print (up-env 2 n))
                    (if (= n 0)
                        1
                        (* n (fact (- n 1))))))))))
  (let ((n 7))
    (fact 5))))
```

The first two executions of `fact` print a total of 4 values.

- (i) [2 points] Assuming static scoping, what are the first four values printed?  
(ii) [2 points] Assuming dynamic scoping, what are the first four values printed?
- b. [15 points]  
We modify the denotational semantics of Appendix A to support the new `up-env` form by adding the following new domain:

$\text{nEnvironment} = \text{Environment} \times \text{nEnvironment}_{\perp}$

We also have the following constructor and destructor helper functions:

$\text{make-nenv} : \text{Environment} \times \text{nEnvironment}_{\perp} \rightarrow \text{nEnvironment}$

$\text{nenv-env} : \text{nEnvironment} \rightarrow \text{Environment}$

$\text{nenv-parent} : \text{nEnvironment} \rightarrow \text{nEnvironment}_{\perp}$

With these domains in hand, one can extend the denotational semantics to handle `up-env`. Note that your new definitions for  $\mathcal{E}$  will take an `nEnvironment` instead of an `Environment`.

- (i) [3 points] Define  $\text{nlookup}$ , which looks up an identifier:  
 $\text{nlookup} : (\text{nEnvironment} \rightarrow \text{Identifier} \rightarrow \text{Integer}) \rightarrow \text{Binding}$   
You may use  $\text{lookup}$  as a subroutine.
- (ii) [3 points] What is  $\mathcal{E}[[I]]$ ?
- (iii) [3 points] What is  $\mathcal{E}[[\text{up-env } E \ I]]$ ?
- (iv) [3 points] What is  $\mathcal{E}[[\text{proc } I \ E]]$ ?
- (v) [3 points] What is  $\mathcal{E}[[\text{call } E_1 \ E_2]]$ ?

Modify the denotational semantics of Appendix A to support the new `up-env` form. Show any changes that you must make to the denotational semantics; do not mention any domains or rules that remain unchanged.

- c. [6 points] Before being dragged away, Ben began to work on a desugaring from Nosy FLK! to FLK!. Unfortunately, most of Ben's notes were confiscated; all that remains is one auxiliary function  $\text{fresh-I-n}$ , the rule for  $D$  (in terms of a helper desugaring  $\mathcal{D}'$ ), and one rule from the helper desugaring  $\mathcal{D}'$ :

$\text{fresh-I-n} : \text{Identifier} \times \text{Integer} \rightarrow \text{Identifier}$

$\text{fresh-I-n}$  returns the same unique fresh identifier for each invocation with the same arguments.

$$\begin{aligned} \mathcal{D}[[E]] &= \mathcal{D}'[[E]] \ 0 \quad \text{for all } E \\ \mathcal{D}'[[I]] \ n &= I \end{aligned}$$

Furthermore, you find a comment that his preliminary solution is quite constrained: the numeric argument to `up-env` must be a literal number; the number must indicate an environment that explicitly binds the identifier; the desugaring assumes static scoping; and the desugaring fails in the presence of side effects. (You may provide a solution better than Ben's, but you are not required to. You must reuse the two existing rules, however.)

Write the rules for the auxiliary desugaring  $\mathcal{D}'[[E]]n$  that helps to convert Nosy FLK! to FLK!. (Don't introduce any additional helper functions.) Omit any rules that merely implement the structural induction by applying the desugaring to each subexpression, with the same arguments as were used for the full expression. Assume that both Nosy FLK! and FLK! use call-by-value semantics.

## Problem 5: Pragmatics [17 points]

Space-Savers, Inc., is working on reducing the run-time space requirements of compiled programs written in a **functional** subset of SCHEME. The company is particularly interested an expression with the following form:

```
(let ((f (lambda (b) E)))
      (primop + (call f 1) (call f 2)))
```

where  $a$  is the only free variable of  $(\text{lambda } (b) E)$ . (You should assume that  $a$  is bound somewhere in the surrounding context of this example and all subsequent examples.)

The company uses a compiler that is composed of a series of source-to-source transforms. The closure conversion phase of their compiler generates flat closures for all `lambda` expressions. Recall that a flat closure is a tuple containing a code component and the values of all free variables referenced by the code component. Closure conversion produces an expression in which closures are explicitly allocated with the `closure-primop` and the values of free variables are explicitly extracted from closures by the `closure-ref` `primop`:

```
(let ((f (%closure (lambda (.closure.73. b) Eclosed)
                  a)))
      (primop + (call-closure f 1) (call-closure f 2)))
```

Here,  $E_{closed}$  is a version of  $E$  in which every reference to  $a$  has been replaced by `(%closure-ref .closure.73. 1)`. (Of course, closure conversion also affects calls, lambdas, and other variable references within  $E_{closed}$ .) The `call-closure` construct applies the code component of a closure to the specified arguments, supplying the closure itself as an implicit first argument.

New employee Bud Lojack, recalling a recent 6.821 problem set on partial evaluation, suggests inlining and reducing the calls to `f` in the original example to yield:

```
(primop + [1/b]E [2/b]E)
```

Bud says, “Before my nifty transformation, the program had to allocate a closure for `f`. But the transformed program doesn’t even have a definition for `f`, so no closure for `f` is ever allocated! So my transformation reduces the space used by the program.”

- a. [5 points] Name one way in which Bud’s transformation can *increase* the space used by the program.

Seasoned hacker Chloe Jour has an idea for reducing the space used by the program. She transforms the sample expression into:

```
(let ((f' (lambda (a b) E)))
      (primop + (call f' a 1) (call f' a 2)))
```

Chloe has transformed the function `f` into the function `f'`, which has the same body as `f`, but takes the free variable  $a$  of `f` as an explicit argument. She has changed each `call` of `f` into a `call` of `f'`, adding  $a$  as an extra argument to the `call`. After closure conversion, this expression becomes:

```
(let ((f' (%closure (lambda (.closure.37. a b) Eclosed'))))
      (primop + (call-closure f' a 1) (call-closure f' a 2)))
```

She says, “Notice that the variable  $a$  is not free in `f'`, so the closure associated with `f'` is smaller than the closure associated with `f`. Therefore my transformation reduces the amount of space *allocated for closures* during execution of the program.”

Bud is excited by Chloe’s suggestion. He decides to transform every subexpression of his program of the form:

$(\text{let } ((I_{fun} \ (\text{lambda } (I_1 \ \dots \ I_n) \ E_{fun})))$   
 $\ E_{let})$

into

$(\text{let } ((I_{fun}' \ (\text{lambda } (J_1 \ \dots \ J_k \ I_1 \ \dots \ I_n) \ E_{fun})))$   
 $\ E_{let}')$

where:

- $J_1 \ \dots \ J_k$  are the free variables of  $(\text{lambda } (I_1 \ \dots \ I_n) \ E_{fun})$ , and
- $E_{let}'$  has been obtained from  $E_{let}$  by transforming each occurrence  $(\text{call } I_{fun} \ E_1 \ \dots \ E_n)$  into  $(\text{call } I_{fun}' \ J_1 \ \dots \ J_k \ E_1 \ \dots \ E_n)$ .

“Of course,” Bud notes, “it may be necessary to rename the bound variables of  $E_{let}$  in order to avoid capture of the variables  $J_1 \ \dots \ J_k$ .”

He shows his idea to Chloe, who has two objections (even after Bud fixes the potential name capture problem). “First,” she says, “this transformation is not semantics-preserving in general.”

- b. [6 points] Give an expression  $E_{let}$  such that Bud’s transformation does not preserve the semantics of the following program. (Recall that the language is functional.)

$(\text{let } ((f \ (\text{lambda } (b) \ (\text{primop } + \ a \ b))))$   
 $\ E_{let})$

“Second,” says Chloe, “even when your transformation *is* semantics-preserving, it may actually *increase* the total amount of memory allocated for closures during the evaluation of a program.”

- c. [6 points] Give an expression  $E_{let}$  such that:

- The semantics of the following expression is preserved by Bud’s transformation:

$(\text{let } ((f \ (\text{lambda } (b) \ (\text{primop } + \ a \ b))))$   
 $\ E_{let})$

- AND the transformed expression allocates more *space for closures* than the original expression.

(Recall that the language is functional.)

# Appendix A: Standard Semantics of FLK!

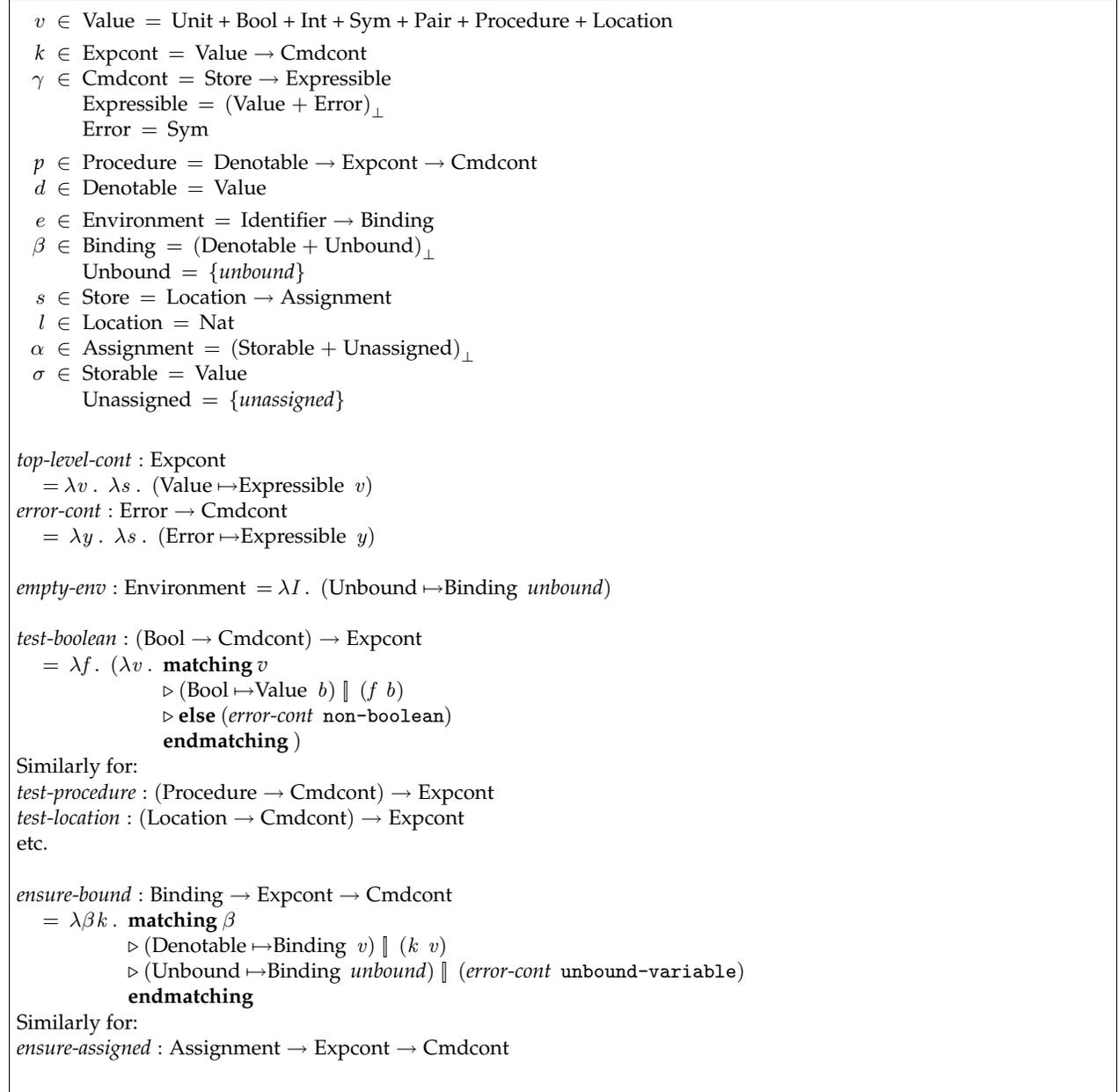


Figure 1: Semantic algebras for standard semantics of strict CBV FLK!.

```

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 = λs . (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!.

```

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

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

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

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

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

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

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

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

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

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

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

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

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

 $\text{extract-value} : \text{Expressible} \rightarrow \text{Binding}$ 
 $= \lambda a . \mathbf{matching} \ a$ 
   $\triangleright (\text{Value} \mapsto \text{Expressible } v) \parallel (\text{Denotable} \mapsto \text{Binding } v)$ 
   $\triangleright \mathbf{else} \ \perp_{\text{Binding}}$ 
   $\mathbf{endmatching}$ 

```

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

# Appendix B: Typing Rules for SCHEME/XSP

## SCHEME/X Rules

$\vdash N : \text{int}$	[int]
$\vdash B : \text{bool}$	[bool]
$\vdash S : \text{string}$	[string]
$\vdash (\text{symbol } I) : \text{sym}$	[sym]
$A[I:T] \vdash I : T$	[var]
$\frac{\forall i (A \vdash E_i : T_i)}{A \vdash (\text{begin } E_1 \dots E_n) : T_n}$	[begin]
$\frac{A \vdash E : T}{A \vdash (\text{the } T E) : T}$	[the]
$\frac{A \vdash E_1 : \text{bool} ; A \vdash E_2 : T ; A \vdash E_3 : T}{A \vdash (\text{if } E_1 E_2 E_3) : T}$	[if]
$\frac{A[I_1:T_1, \dots, I_n:T_n] \vdash E_B : T_B}{A \vdash (\text{lambda } ((I_1 T_1) \dots (I_n T_n)) E_B) : (-> (T_1 \dots T_n) T_B)}$	[\lambda]
$\frac{A \vdash E_P : (-> (T_1 \dots T_n) T_B) \quad \forall i (A \vdash E_i : T_i)}{A \vdash (E_P E_1 \dots E_n) : T_B}$	[call]
$\frac{\forall i (A \vdash E_i : T_i) \quad A[I_1:T_1, \dots, I_n:T_n] \vdash E_B : T_B}{A \vdash (\text{let } ((I_1 E_1) \dots (I_n E_n)) E_B) : T_B}$	[let]
$\frac{A' = A[I_1:T_1, \dots, I_n:T_n] \quad \forall i (A' \vdash E_i : T_i) \quad A' \vdash E_B : T_B}{A \vdash (\text{letrec } ((I_1 T_1 E_1) \dots (I_n T_1 E_n)) E_B) : T_B}$	[letrec]
$\frac{A \vdash (\forall i [T_i/I_i]) E_{\text{body}} : T_{\text{body}}}{A \vdash (\text{tlet } ((I_1 T_1) \dots (I_n T_n)) E_{\text{body}}) : T_{\text{body}}}$	[tlet]
$\frac{\forall i (A \vdash E_i : T_i)}{A \vdash (\text{record } (I_1 E_1) \dots (I_n E_n)) : (\text{recordof } (I_1 T_1) \dots (I_n T_n))}$	[record]
$\frac{A \vdash E : (\text{recordof } \dots (I T) \dots)}{A \vdash (\text{select } I E) : T}$	[select]
$\frac{A \vdash E : T_E ; T = (\text{oneof } \dots (I T_E) \dots)}{A \vdash (\text{one } T I E) : T}$	[one]
$\frac{A \vdash E_{\text{disc}} : (\text{oneof } (I_1 T_1) \dots (I_n T_n)) \quad \forall i. \exists j. ((I_i = I_{\text{tag}_j}) \wedge (A[I_{\text{val}_j}:T_i] \vdash E_j : T))}{A \vdash (\text{tagcase } E_{\text{disc}} (I_{\text{tag}_1} I_{\text{val}_1} E_1) \dots (I_{\text{tag}_n} I_{\text{val}_n} E_n)) : T}$	[tagcase1]
$\frac{A \vdash E_{\text{disc}} : (\text{oneof } (I_1 T_1) \dots (I_n T_n)) \quad \forall i   (\exists j. (I_i = I_{\text{tag}_j})) . A[I_{\text{val}_j}:T_i] \vdash E_j : T \quad A \vdash E_{\text{default}} : T}{A \vdash (\text{tagcase } E_{\text{disc}} (I_{\text{tag}_1} I_{\text{val}_1} E_1) \dots (I_{\text{tag}_n} I_{\text{val}_n} E_n) (\text{else } E_{\text{default}})) : T}$	[tagcase2]

## Rules Introduced by SCHEME/XS to Handle Subtyping

$T \sqsubseteq T$	[reflexive- $\sqsubseteq$ ]
$\frac{T_1 \sqsubseteq T_2 ; T_2 \sqsubseteq T_3}{T_1 \sqsubseteq T_3}$	[transitive- $\sqsubseteq$ ]
$\frac{\begin{array}{c} (T_1 \sqsubseteq T_2) \\ (T_2 \sqsubseteq T_1) \end{array}}{T_1 \equiv T_2}$	[ $\equiv$ ]
$\frac{\forall i \exists j ((I_i = J_j) \wedge (S_j \sqsubseteq T_i))}{(\text{recordof } (J_1 S_1) \dots (J_m S_m)) \sqsubseteq (\text{recordof } (I_1 T_1) \dots (I_n T_n))}$	[recordof- $\sqsubseteq$ ]
$\frac{\forall j \exists i ((J_j = I_i) \wedge (S_j \sqsubseteq S_i))}{(\text{oneof } (J_1 S_1) \dots (J_m S_m)) \sqsubseteq (\text{oneof } (I_1 T_1) \dots (I_n T_n))}$	[oneof- $\sqsubseteq$ ]
$\frac{\forall i (T_i \sqsubseteq S_i) ; S_{\text{body}} \sqsubseteq T_{\text{body}}}{(-> (S_1 \dots S_n) S_{\text{body}}) \sqsubseteq (-> (T_1 \dots T_n) T_{\text{body}})}$	[->- $\sqsubseteq$ ]
$\frac{\forall T ([T/I_1]T_1 \sqsubseteq [T/I_2]T_2)}{(\text{recof } I_1 T_1) \sqsubseteq (\text{recof } I_2 T_2)}$	[recof- $\sqsubseteq$ ]
$\frac{\begin{array}{c} A \vdash E_{\text{rator}} : (-> (T_1 \dots T_n) T_{\text{body}}) \\ \forall i ((A \vdash E_i : S_i) \wedge (S_i \sqsubseteq T_i)) \end{array}}{A \vdash (E_{\text{rator}} E_1 \dots E_n) : T_{\text{body}}}$	[call-inclusion]
$\frac{\begin{array}{c} A \vdash E : S \\ S \sqsubseteq T \end{array}}{A \vdash (\text{the } T E) : T}$	[the-inclusion]

## Rules Introduced by SCHEME/XSP to Handle Polymorphism

$\frac{\begin{array}{c} A \vdash E : T ; \\ \forall i (I_i \notin (\text{FTV } (\text{Free-Ids}[E])A)) \end{array}}{A \vdash (\text{plambda } (I_1 \dots I_n) E) : (\text{poly } (I_1 \dots I_n) T)}$	[p $\lambda$ ]
$\frac{A \vdash E : (\text{poly } (I_1 \dots I_n) T_E)}{A \vdash (\text{proj } E T_1 \dots T_n) : (\forall i [T_i/I_i]) T_E}$	[project]
$\frac{(\forall i [I_i/J_i]) S \sqsubseteq T, \forall i (I_i \notin \text{Free-Ids}[S])}{(\text{poly } (J_1 \dots J_n) S) \sqsubseteq (\text{poly } (I_1 \dots I_n) T)}$	[poly- $\sqsubseteq$ ]

### recof Equivalence

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

# Appendix C: Typing Rules for SCHEME/R

$\vdash \#u : \text{unit}$	[unit]
$\vdash B : \text{bool}$	[bool]
$\vdash N : \text{int}$	[int]
$\vdash (\text{symbol } I) : \text{sym}$	[symbol]
$[\dots, I : T, \dots] \vdash I : T$	[var]
$[\dots, I : (\text{generic } (I_1 \dots I_n) T_{body}), \dots] \vdash I : (\forall i [T_i/I_i])T_{body}$	[genvar]
$\frac{A \vdash E_{test} : \text{bool} ; A \vdash E_{con} : T ; A \vdash E_{alt} : T}{A \vdash (\text{if } E_{test} E_{con} E_{alt}) : T}$	[if]
$\frac{A[I_1 : T_1, \dots, I_n : T_n] \vdash E_{body} : T_{body}}{A \vdash (\text{lambda } (I_1 \dots I_n) E_{body}) : (-> (T_1 \dots T_n) T_{body})}$	[\lambda]
$\frac{A \vdash E_{rator} : (-> (T_1 \dots T_n) T_{body}) \quad \forall i . (A \vdash E_i : T_i)}{A \vdash (E_{rator} E_1 \dots E_n) : T_{body}}$	[apply]
$\frac{\forall i . (A \vdash E_i : T_i) \quad A[I_1 : \text{Gen}(T_1, A), \dots, I_n : \text{Gen}(T_n, A)] \vdash E_{body} : T_{body}}{A \vdash (\text{let } ((I_1 E_1) \dots (I_n E_n)) E_{body}) : T_{body}}$	[let]
$\frac{\forall i . (A[I_1 : T_1, \dots, I_n : T_n] \vdash E_i : T_i) \quad A[I_1 : \text{Gen}(T_1, A), \dots, I_n : \text{Gen}(T_n, A)] \vdash E_{body} : T_{body}}{A \vdash (\text{letrec } ((I_1 E_1) \dots (I_n E_n)) E_{body}) : T_{body}}$	[letrec]
$\frac{\forall i . (A \vdash E_i : T_i)}{A \vdash (\text{record } (I_1 E_1) \dots (I_n E_n)) : (\text{recordof } (I_1 T_1) \dots (I_n T_n))}$	[record]
$\frac{A \vdash E_r : (\text{recordof } (I_1 T_1) \dots (I_n T_n)) \quad A[I_1 : T_1, \dots, I_n : T_n] \vdash E_b : T}{A \vdash (\text{with } (I_1 \dots I_n) E_r E_b) : T}$	[with]
$\frac{A \vdash (\text{letrec } ((I_1 E_1) \dots (I_n E_n)) E_{body}) : T}{A \vdash (\text{program } (\text{define } I_1 E_1) \dots (\text{define } I_n E_n) E_{body}) : T}$	[program]

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

# Appendix D: Type Reconstruction Algorithm for SCHEME/R

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

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

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

$$R[\mathbf{symbol } I] A S = \langle \mathbf{sym}, S \rangle$$

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

$$R[I] A[I : (\mathbf{generic } (I_1 \dots I_n) T)] S = \langle T[?v_i/I_i], S \rangle \quad (?v_i \text{ are new})$$

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

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

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

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

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

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

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

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

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