Continuation Passing Style (CPS) Conversion

CIS400 (Compiler Construction)
Kris Micinski, Fall 2021

* These slides use examples written by Thomas Gilray, Andrew Appel, and Matthew Might

In continuation passing style, control is **completely explicit**, as all calls are invocations of continuations

```
(f (g 1))
```

- First call g, get its result
- Next, call f on that result
- Then, halt

In continuation passing style, control is **completely explicit**, as all calls are invocations of continuations

```
(f (g 1))
```

- First call g, get its result
- Next, call f on that result
- Then, halt

```
(define halt (lambda (x k) x))

(g 1 (lambda (g-r k0)
       (f g-r halt)))
```

In continuation passing style, control is **completely explicit**, as all calls are invocations of continuations

```
(f (g 1))
```

- First call g, get its result
- Next, call f on that result
- Then, halt

```
(define halt (lambda (x k) x))
(g 1 (lambda (g-r k0)
       (f g-r halt)))
```

- Every function (λ) gets an explicit “continuation” argument
- “halt” continuation gives the final result
- (Real implementation would represent halt specially)

Like programming exclusively with callbacks, whatever the lambda does must end by calling some continuation

```
((λ (x) x) (λ (y) y))
```

```
((λ (x $k0) ($k0 x))
 (λ (y $k1) ($k1 y))
 (λ ($return $k2) (halt $return)))
```

Final “return” value

Look at this example carefully: every “normal” return point (in the direct-style code) is (implicitly) an invocation of the current continuation, made explicit in CPS!

One consequence of this is that every lambda makes a choice: which in-scope continuation does it invoke?

```
(lambda (x k0)
  (x … kN)) ;; calling argument

          (lambda (x k0)
            (k0 … kN)) ;; calling continuation
```

Different continuations represent different return/join points within the program

No matter what I invoke, I must pass some continuation (or special “halt”/done continuation to end)

Some examples to think about along the way…

```
(x y)
(x (x y))
((lambda (x) x) y)
(call/cc (lambda (k) x))
(call/cc (lambda (k) (k x)))
```

No matter what I invoke, I must pass some continuation
(or special “halt”/done continuation to end)

Naïve CPS transformation

Two functions:

M : atom → atomic-cps-expr

```
(define (M expr) …)
```

converts atomic value to atomic CPS value (i.e., adds continuations to lambdas)

T : expression × syntactic-continuation → cps-expression

```
(define (T expr k) …)
```

Takes a term and a syntactic continuation (i.e., a variable representing a continuation within scope in the program) and applies k to the CPS-converted version of expr

converts atomic value to atomic CPS value (i.e., adds continuations to lambdas)

```
(define (M expr)
  (match expr
    [`(lambda (,var) ,expr)
      ; =>
      (define $k (gensym '$k))
     `(lambda (,var ,$k) ,(T expr $k))]
    [(? symbol?) expr]
    [(? number? n) expr]))
```

Takes a term and a syntactic continuation (i.e., a variable representing a continuation within scope in the program) and applies **k** to the CPS-converted version of **expr**

```
(define (T expr cc)
  (match expr
    [`(λ ,x ,_) `(,cc ,(M expr))]
    [(? symbol?) `(,cc ,(M expr))]
    [(? number?) `(,cc ,(M expr))]
    [`(,e0 ,e1)
      (define $e0res (gensym '$e0res))
      (define $e1res (gensym '$e1res))
      (T e0 `(λ (,$e0res)
                ,(T e1 `(λ (,$e1res)
                           (,$e0res ,$e1res ,cc)))))]))
```

Why does this transformation work?

```
(define (M expr)
  (match expr
    [`(lambda (,var) ,expr)
      ; =>
      (define $k (gensym '$k))
     `(lambda (,var ,$k) ,(T expr $k))]
    [(? symbol?) expr]
    [(? number? n) expr]))
```

T assumes its input is in direct-style. Atoms are implicitly return points, so invoke the current continuation (**cc**) on them

To perform a direct-style call, first compute e0, passing a continuation that takes its result, then calls e1, then performs the call and invokes the current continuation.

```
(define (T expr cc)
  (match expr
    [`(λ ,x ,_) `(,cc ,(M expr))]
    [(? symbol?) `(,cc ,(M expr))]
    [(? number?) `(,cc ,(M expr))]
    [`(,e0 ,e1)
      (define $e0res (gensym '$e0res))
      (define $e1res (gensym '$e1res))
      (T e0 `(λ (,$e0res)
                ,(T e1 `(λ (,$e1res)
                           (,$e0res ,$e1res ,cc)))))]))
```

Why is the transformation naive…?

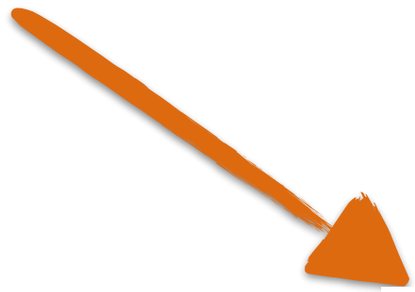
Because it is wasteful: things that are already atoms shouldn't get their own lambda, but in this transformation they do…

The transformation of `(g a)` results in…

```
(T '(g a) 'halt)
```

It does this…

We would rather it do this…



```
(g a halt)
```

```
((λ ($f1445)
  ((λ ($e1446)
    ($f1445 $e1446 halt)) a)) g)
```

The higher-order CPS transform

To fix our naive transformation, we can instead pass a **Racket** function *to construct the syntax of the continuation* instead of an explicit syntactic continuation.

This will allow us to interpose a check to see if the value is an atom

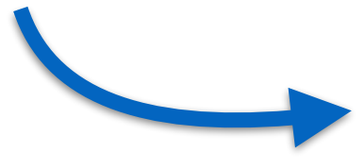
The ANF conversion algorithm uses this same trick!

```
;; Now k is a *Racket* function
(define (T expr k) …)

;; For example, to convert (g a)…
(T '(g a) (λ (ans) `(halt ,ans)))
```

```
(define (T expr k)
  (match expr
    [`(λ . ,_)          (k (M expr))]
    [ (? symbol?)       (k (M expr))]
    [`(,f ,e)
      (define $rv (gensym '$rv))
      (define cont `(λ (,$rv) ,(k $rv)))
      (T f (λ ($f)
             (T e (λ ($e)
(define (M expr)          `(,$f ,$e ,cont)))))))]))
  (match expr
    [`(λ (,var) ,expr)
      (define $k (gensym '$k))
     `(λ (,var ,$k) ,(T expr (λ (rv) `(,$k ,rv))))]
    [(? symbol?) expr]))


(T '(g a) (λ (ans) `(halt ,ans)))
```



```
(g a (λ ($rv1) (halt $rv1)))
```

Our CPS transformation is made a lot easier if we assume the input is in ANF: that way, we know precisely when we have to form continuations—at all non-tail forms.

```
(define (T expr cc)
  (match expr
[`(λ ,x ,_) `(,cc ,(M expr))]
[(? symbol?) `(,cc ,(M expr))]
[(? number?) `(,cc ,(M expr))]
[`(let ([,x ,e0]) ,e1)
 (T e0 `(λ (,(gensym 'k) ,x) ,(T e1 cc))
[`(,a0 ,a1) (,(M a0) ,(M a1) ,cc)])))]))
```

Solution for Project 3

Similar to the algorithm presented so far, we will demonstrate a correct CPS-transformation for ANF (and assignment)-converted Scheme expressions

First is **`T-ae`**, which transforms an ANF atom to a CPS atom

```
(define (T-ae e)
    (match e
      [(? symbol? x) x]
      [`',dat `',dat]
      [`(lambda (,xs ...) ,ebody)
       (define kvar (gensym 'k))
       `(lambda (,kvar ,@xs) ,(T ebody kvar))]
      [`(lambda ,x ,ebody)
       (define argvar (gensym x))
       (define kvar (gensym 'k))
       `(lambda ,argvar (let ([,kvar (prim car ,argvar)])
                          (let ([,x (prim cdr ,argvar)])
                            ,(T ebody kvar))))]))
```

Solution for Project 3

Similar to the algorithm presented so far, we will demonstrate a correct CPS-transformation for ANF (and assignment)-converted Scheme expressions

First is **`T-ae`**, which transforms an ANF atom to a CPS atom

Note: we put the k **first** in this implementation

```
(define (T-ae e)
  (match e
    [(? symbol? x) x]
    [`',dat `',dat]
    [`(lambda (,xs ...) ,ebody)
     (define kvar (gensym 'k))
     `(lambda (,kvar ,@xs) ,(T ebody kvar))]
    [`(lambda ,x ,ebody)
     (define argvar (gensym x))
     (define kvar (gensym 'k))
     `(lambda ,argvar (let ([,kvar (prim car ,argvar)])
                        (let ([,x (prim cdr ,argvar)])
                          ,(T ebody kvar))))]))
```

Stackless Compilation

- One consequence of our choice to use CPS as our IR is that we will achieve “stackless” compilation
  - We don’t use the C stack for call/return matching, no need to worry about overflows, etc..
- Many ways to implement stackless compilation
  - Mixing monolithic (i.e., C) and segmented stacks is a challenging performance issue.
- Upshot: our language will be not-horribly-slow, but we will support call/cc by compiling to CPS, which will naturally introduce (a) unnecessary function calls and (b) more indirect jumps (more expensive, hard on icache)
- To ameliorate this problem, we also allow representing several things as straight-line blocks of lets
  - This allows for straight-line blocks of prims, assignments of datums, copying, etc… (which are the common case!)

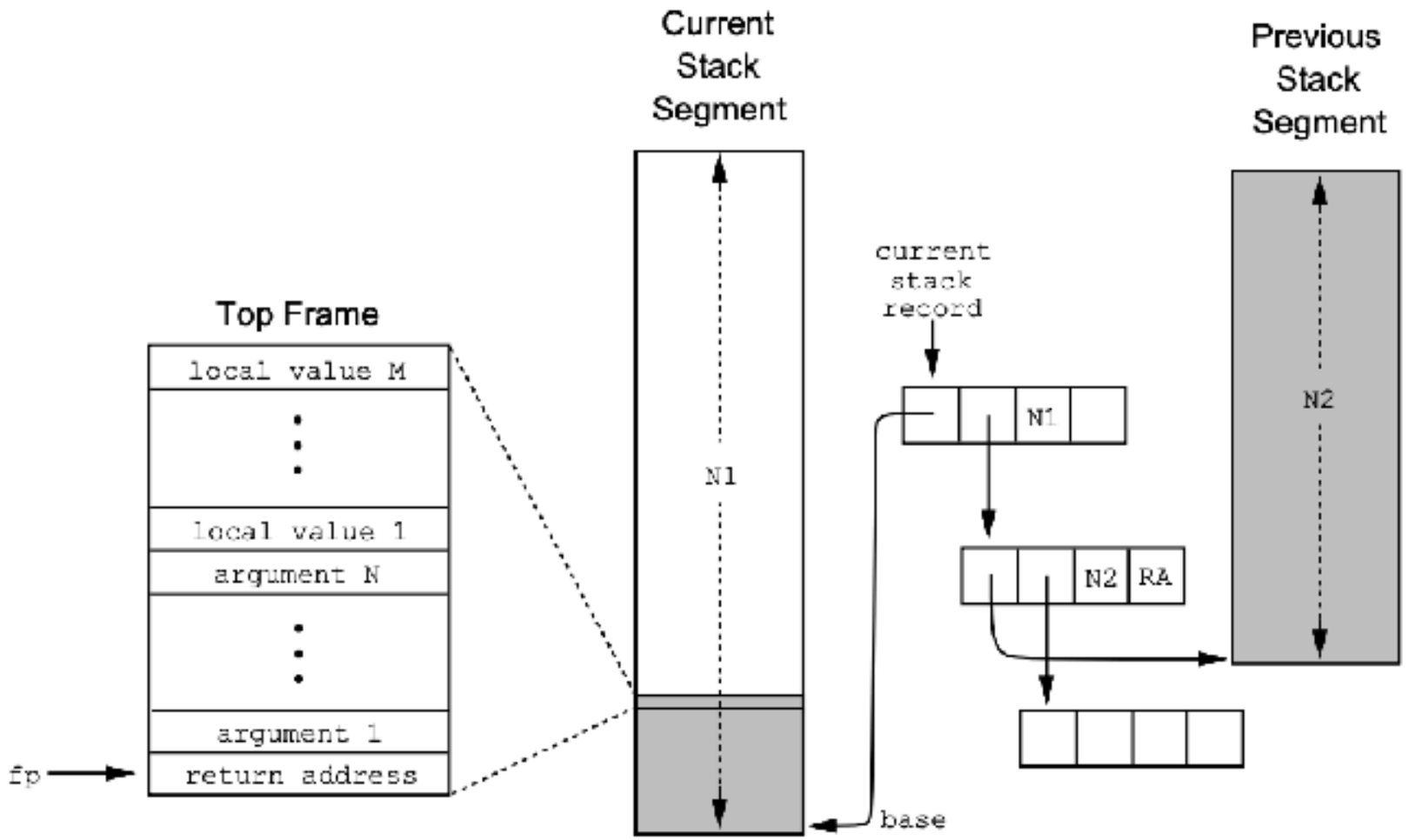


Figure 3. The segmented stack model is a simple generalization of the traditional stack model. By implementing the control stack as a linked list of stack segments, continuation operations are bounded by the size of the top segment instead of the size of the entire control stack.

Representing Control in the Presence of First-Class Continuations *

Robert Hieb, R. Kent Dybvig, Carl Bruggeman
Indiana University
Computer Science Department
Lindley Hall 101
Bloomington IN 47405

Abstract

Languages such as Scheme and Smalltalk that provide continuations as first-class data objects present a challenge to efficient implementation. Allocating activation

efficient linkage on calls and returns. On modern architectures with hierarchical memory, stacks also help maintain locality of memory operations.

Traditional stack management techniques are inadequate for some modern languages, however. If a lan-

Our output language…

```
; e ::= (let ([x (apply-prim op ae)]) e)
;      | (let ([x (prim op ae ...)]) e)
;      | (let ([x (lambda x e)]) e)
;      | (let ([x (lambda (x ...) e)]) e)
;      | (let ([x (quote dat)]) e)
;      | (apply ae ae)
;      | (ae ae ...)
;      | (if ae e e)
; ae ::= (lambda (x ...) e)
;      | (lambda x e)
;      | x
;      | (quote dat)
```

Note that lets are still “valid” as long as their arguments are all things we can atomically evaluate (such as prims)—this simplifies the implementation, and nested lets correspond to “basic blocks” of atomic operations

Next, we define `(T e cae)`

```
(define (T e cae)
    (match e
      ; When we hit an atomic expression, we would usually return
      ; but in CPS, all we can do is call the continuation.
      ; Note: invoke cae with 0: *all* lambdas, even continuations,
      ; must accept arguments, so we use 0 as a "bogus" argument
      [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
      [_ 'todo]))
```

Next, let-bindings..

```
(define (T e cae)
    (match e
      [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
      ; bind all possible RHS of lets to a continuation-extending call
      ; Why is it ok to throw away (not use) _k here?
      [`(let ([,x ,e]) ,ebody)
       (T e `(lambda (,(gensym '_k) ,x) ,(T ebody cae)))]
      [_ 'todo]))
```

Next, let-bindings..

```
(define (T e cae)
    (match e
      [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
      ; This allows us to not over-transform some specific terms…
      [`(let ([,x (prim ,op ,aes ...)]) ,ebody)
       `(let ([,x (prim ,op ,@(map T-ae aes))]) ,(T ebody cae))]
      [`(let ([,x (apply-prim ,op ,aex)]) ,ebody)
       `(let ([,x (apply-prim ,op ,(T-ae aex))]) ,(T ebody cae))]
      [`(let ([,x `(lambda ,xa ,ef)]) ,ebody)
       `(let ([,x (lambda ,xa ,ef)]) ,(T ebody cae))]
      [`(let ([,x ',dat]) ,ebody)
       `(let ([,x ',dat]) ,(T ebody cae))]
      ; bind all possible RHS of lets to a continuation-extending call
      ; Why is it ok to throw away (not use) _k here?
      [`(let ([,x ,e]) ,ebody)
       (T e `(lambda (,(gensym '_k) ,x) ,(T ebody cae)))]
      [_ 'todo]))
```

Prims desugar to lets

```
(define (T e cae)
    (match e
      [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
      …
      [`(let ([,x ,e]) ,ebody)
       (T e `(lambda (,(gensym '_k) ,x) ,(T ebody cae)))]
      ; prim must be lifted into a let binding to become valid CPS. This
      ; will help during the final transformation into the target language.
      [`(prim ,op ,aes ...)
       (define bndvar (gensym 'cpsprim))
       `(let ([,bndvar (prim ,op ,@(map T-ae aes))]) ,(T bndvar cae))]
      ; same is true for apply-prim
      [`(apply-prim ,op ,aex)
       (define bndvar (gensym 'cpsaprim))
       `(let ([,bndvar (apply-prim ,op ,(T-ae aex))]) ,(T bndvar cae))]
      [_ 'todo]))
```

Still “valid” CPS in our output language

call/cc

```
(define (T e cae)
    (match e
      [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
      …
      ; Here we truly define call-with-current-continuation.
      ; We are finally *calling* the function with the current continuation
      ; as the argument! Of course, if the continuation is never explicitly
      ; a continuation is needed to 'return' with, so we use cae there too.
      [`(call/cc ,aef)
       `(,(T-ae aef) ,cae ,cae)]
      [_ 'todo]))
```

If

```
(define (T e cae)
    (match e
      [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
      …
; If is a simple enough transformation, we just call the correct T
; function as needed and move along.
[`(if ,ec ,aet ,aef)
 `(if ,(T-ae ec) ,(T aet cae) ,(T aef cae))]
[_ 'todo]))
```

Apply

```
(define (T e cae)
    (match e
      [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
      …
      ; apply is allowed in the output language.
      ; but we must add the continuation to the given list.
      [`(apply ,aef ,aex)
       (define argvar (gensym 'cpsargs))
       `(let ([,argvar (prim cons ,cae ,(T-ae aex))])
          (apply ,(T-ae aef) ,argvar))]
      [_ 'todo]))
```

Finally, application

```
(define (T e cae)
    (match e
      [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
      …
      ; When calling a function, we need to specify the continuation cae
      [`(,aef ,aes ...) `(,(T-ae aef) ,cae ,@(map T-ae aes))]))
```

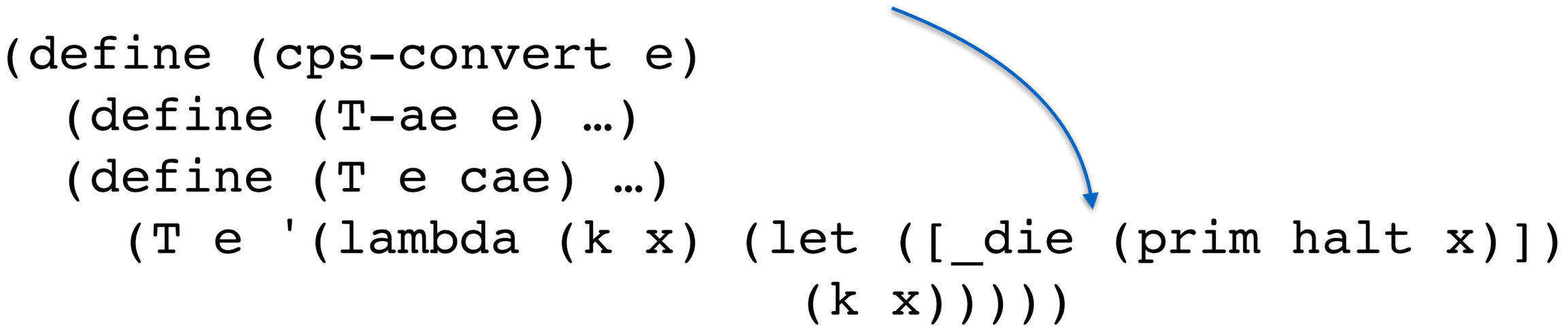
Here's the whole thing `(T e cae)`

```
(define (T e cae)
  (match e
    ; When we hit an atomic expression, we would usually return
    ; but in CPS, all we can do is call the continuation.
    [(or `(lambda ,_ ,_) (? symbol?) `',_) `(,cae '0 ,(T-ae e))]
    ;;; These patterns are moderately unnecessary, but save some parsing later
    ; We can catch let bindings that will be legal in CPS and not over-transform them.
    [`(let ([,x (prim ,op ,aes ...)]) ,ebody)
     `(let ([,x (prim ,op ,@(map T-ae aes))]) ,(T ebody cae))]
    [`(let ([,x (apply-prim ,op ,aex)]) ,ebody)
     `(let ([,x (apply-prim ,op ,(T-ae aex))]) ,(T ebody cae))]
    [`(let ([,x `(lambda ,xa ,ef)]) ,ebody)
     `(let ([,x (lambda ,xa ,ef)]) ,(T ebody cae))]
    [`(let ([,x ',dat]) ,ebody)
     `(let ([,x ',dat]) ,(T ebody cae))]
    ;;; end 'unnecessary' let patterns.
    ; bind all possible RHS of lets to a continuation-extending call
    [`(let ([,x ,e]) ,ebody)
     (T e `(lambda (,(gensym '_k) ,x) ,(T ebody cae)))]
    ; prim must be lifted into a let binding to become valid CPS. This
    ; will help during the final transformation into the target language.
    [`(prim ,op ,aes ...)
     (define bndvar (gensym 'cpsprim))
     `(let ([,bndvar (prim ,op ,@(map T-ae aes))]) ,(T bndvar cae))]
    ; same is true for apply-prim
    [`(apply-prim ,op ,aex)
     (define bndvar (gensym 'cpsaprim))
     `(let ([,bndvar (apply-prim ,op ,(T-ae aex))]) ,(T bndvar cae))]
    ; here we truly define call-with-current-continuation.
    ; we are finally *calling* the function with the current continuation
    ; as the argument! Of course, if the continuation is never explicitly
    ; a continuation is needed to 'return' with. So use the current continuation there as well.
    [`(call/cc ,aef)
     `(,(T-ae aef) ,cae ,cae)]
    ; If is a simple enough transformation, we just call the correct T
    ; function as needed and move along.
    [`(if ,ec ,aet ,aef)
     `(if ,(T-ae ec) ,(T aet cae) ,(T aef cae))]
    ; apply is allowed in the output language.
    ; but we must add the continuation to the given list.
    [`(apply ,aef ,aex)
     (define argvar (gensym 'cpsargs))
     `(let ([,argvar (prim cons ,cae ,(T-ae aex))])
        (apply ,(T-ae aef) ,argvar))]
    ; When calling a function, we need to give the continuation now
    [`(,aef ,aes ...) `(,(T-ae aef) ,cae ,@(map T-ae aes))]))
```

`(cps-convert e)`

Invokes (T e cae) with a continuation that will get the final return value to return as the program result (remember, it will also have a continuation, as will **all** functions in our language)

We “kick off” e with an initial continuation that applies the prim halt to x, which we will handle (in later stages) specially to bind to an exit routine in our language.



```
(define (cps-convert e)
  (define (T-ae e) …)
  (define (T e cae) …)
    (T e '(lambda (k x) (let ([_die (prim halt x)])
                          (k x)))))
```