CS150 APL: CPS Transformation

Guannan Wei
guannan.wei@tufts.edu

Oct 28, 2025

Tufts University

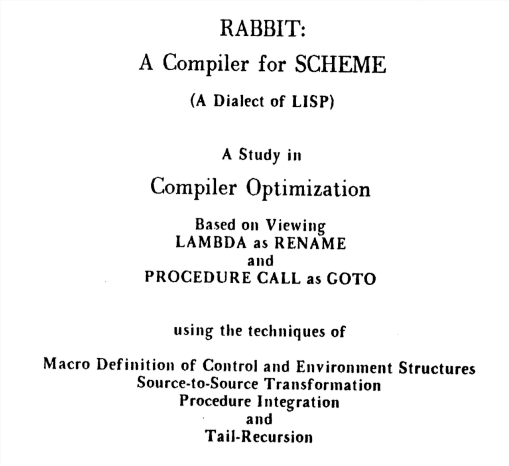
Logistics

- Paper presentation/discussion
  - 30 minutes presentation + 10 minutes discussion
  - Team-up (up to 2) presentation is encouraged
  - Presenter(s) should prepare a list of discussion points/questions
  - Choose your paper early
- 35 minutes mini-lecture
- Project check-in this week

Why compiling to CPS?

- Continuation-passing style: every computation receives a continuation, representing the rest of the computation as a function
- Useful programming technique:
  - Non-standard control flow (e.g., backtracking, coroutines, exceptions, generators)
- Intermediate representation for functional language compilers:
  - Control-flow and evaluation order are made explicit
  - Easy to implement control operators (e.g., call/cc) and effect handlers
  - All function calls are tail calls (constant stack space)

Compiling functional languages



RABBIT:

A Compiler for SCHEME

(A Dialect of LISP)

A Study in

Compiler Optimization

Based on Viewing
LAMBDA as RENAME
and
PROCEDURE CALL as GOTO

using the techniques of

Macro Definition of Control and Environment Structures
Source-to-Source Transformation
Procedure Integration
and
Tail-Recursion

- Rabbit compiler for Scheme (1978)
- Others: SML/NJ, Chicken Scheme, etc.

Today's lecture

- Manual CPS transformation by writing an interpreter in CPS
- Algorithmic CPS transformation by writing a CPS transformer

Defining the source and target languages

Source language: direct-style λ-calculus

$$
\begin{array}{rcll}
n & \in & \mathbb{N} & \\
t & ::= & n \mid x \mid \lambda x.t \mid t_1\, t_2 \mid t_1 + t_2 \mid t_1 - t_2 \mid \cdots & \textbf{terms}
\end{array}
$$

Target language: continuation-passing style

$$
\begin{array}{rcll}
v & ::= & n \mid x \mid \lambda x k.c \mid \lambda_c x.c \mid \mathsf{halt} & \textbf{values} \\
e & ::= & v \mid v_1 + v_2 \mid v_1 - v_2 \mid \cdots & \textbf{simple expressions} \\
c & ::= & \mathsf{let}\ x = e\ \mathsf{in}\ c \mid v_1\ v_2 \mid v_1\ v_2\ v_3 & \textbf{term/computation}
\end{array}
$$

Example

- Translation function from source terms to target terms:

$$[\![\cdot]\!] : \mathsf{Src} \to \mathsf{Tgt}$$

- Example:

$$[\![(\lambda x.x + 1)\ (2 \times 3)]\!] =$$
$$\mathsf{let}\ x_2 = 2 \times 3\ \mathsf{in}\ (\lambda x c_0.\mathsf{let}\ x_1 = x + 1\ \mathsf{in}\ c_0\ x_1)\ x_2\ (\lambda_c w_3.\mathsf{halt}\ w_3)$$

CPS Translation

- Eventually, you will understand CPS translation:

$$
\begin{aligned}
[\![\cdot]\!] &: \mathsf{Src} \to (\mathsf{TgtVal} \to \mathsf{Tgt}) \to \mathsf{Tgt} \\
[\![n]\!]\ k &= k(n) \\
[\![x]\!]\ k &= k(x) \\
[\![\lambda x.t]\!]\ k &= k(\lambda x c.[\![t]\!]\ (\lambda v.c\ v)) \\
[\![t_1\ t_2]\!]\ k &= [\![t_1]\!]\ (\lambda v_1.[\![t_2]\!]\ (\lambda v_2.v_1\ v_2\ (\lambda_c w.k(w)))) \\
[\![t_1 + t_2]\!]\ k &= [\![t_1]\!]\ (\lambda v_1.[\![t_2]\!]\ (\lambda v_2.\mathsf{let}\ x = v_1 + v_2\ \mathsf{in}\ k(x)))
\end{aligned}
$$

Roadmap

- Three languages:
  - Source language (Src)
  - Target language (Tgt)
  - Meta language (used to define the transformation)

1. Start from the direct-style interpreter for Src
2. Manually transform (1) to CPS
3. Refactor (2) get the algorithmic CPS transformer from Src to Tgt

Direct-style interpreter for Src

- Defining the syntax and value domain

```
enum Src:
  case Num(i: Int)
  case Var(x: String)
  case Lam(x: String, e: Src)
  case App(e1: Src, e2: Src)
  case BinOp(op: String, e1: Src, e2: Src)

enum SrcVal:
  case Num(i: Int)
  case Closure(x: String, body: Src, env: Map[String, SrcVal])
```

Direct-style interpreter for Src (simple cases)

```
def interp(e: Src, env: Map[String, SrcVal]): SrcVal = e match
  case Src.Num(i) => SrcVal.Num(i)
  case Src.Var(x) => env(x)
  case Src.Lam(x, body) => SrcVal.Closure(x, body, env)
  case Src.App(e1, e2) => ...
  case Src.BinOp(op, e1, e2) => ...
```

Direct-style interpreter for Src (App and BinOp)

```
def interp(e: Src, env: Map[String, SrcVal]): SrcVal = e match
  case Src.App(e1, e2) =>
    val SrcVal.Closure(x, body, cloEnv) = interp(e1, env)
    val v2 = interp(e2, env)
    interp(body, cloEnv + (x -> v2))
  case Src.BinOp(op, e1, e2) => ...
```

Direct-style interpreter for Src (App and BinOp)

```
def interp(e: Src, env: Map[String, SrcVal]): SrcVal = e match
  case Src.App(e1, e2) =>
    val SrcVal.Closure(x, body, cloEnv) = interp(e1, env)
    val v2 = interp(e2, env)
    interp(body, cloEnv + (x -> v2))
  case Src.BinOp(op, e1, e2) =>
    val SrcVal.Num(v1) = interp(e1, env)
    val SrcVal.Num(v2) = interp(e2, env)
    if (op == "+") SrcVal.Num(v1 + v2)
    else if (op == "-") SrcVal.Num(v1 - v2)
    ...
```

CPS interpreter

- Transforming the interpreter to CPS (manually)
- Direct-style interpreter:

```
def interp(e: Src, env: Map[String, SrcVal]): SrcVal
```

- CPS interpreter:

```
def interpk(e: Src, env: Map[String, SrcVal],
            k: SrcVal => SrcVal): SrcVal
```

CPS interpreter (simple cases)

- Basic idea of CPS: instead of returning a value directly, pass it to the continuation function `k`

```
def interpk(e: Src, env: Map[String, SrcVal],
            k: SrcVal => SrcVal): SrcVal =
  e match
    case Src.Num(i) => k(SrcVal.Num(i))
    case Src.Var(x) => k(env(x))
    case Src.Lam(x, body) => k(SrcVal.Closure(x, body, env))
    case Src.App(e1, e2) => ...
    case Src.BinOp(op, e1, e2) => ...
```

CPS interpreter (App)

- Now we need to **construct** the continuation after evaluating `e1` to value `v1`:

```
def interpk(e: Src, env: Map[String, SrcVal],
            k: SrcVal => SrcVal): SrcVal = e match
  case Src.App(e1, e2) =>
    interpk(e1, env, v1 => // v1 has type SrcVal
      ???
      )
  case Src.BinOp(op, e1, e2) => ...
```

CPS interpreter (App)

- Now we need to **construct** the continuation after evaluating e2 to value v2:

```
def interpk(e: Src, env: Map[String, SrcVal],
            k: SrcVal => SrcVal): SrcVal = e match
  case Src.App(e1, e2) =>
    interpk(e1, env, v1 => // v1 has type SrcVal
      interpk(e2, env, v2 => // v2 has type SrcVal
        ???
      ))
  case Src.BinOp(op, e1, e2) => ...
```

CPS interpreter (App)

- Wtih `v1` and `v2`, we can now perform the application:

```
def interpk(e: Src, env: Map[String, SrcVal],
            k: SrcVal => SrcVal): SrcVal = e match
  case Src.App(e1, e2) =>
    interpk(e1, env, v1 =>
      interpk(e2, env, v2 =>
        v1 match
          case SrcVal.Closure(x, body, cloEnv) =>
            interpk(body, cloEnv + (x -> v2), k) // k is passed along
      ))
  case Src.BinOp(op, e1, e2) => ...
```

CPS interpreter (BinOp)

```
def interpk(e: Src, env: Map[String, SrcVal],
            k: SrcVal => SrcVal): SrcVal = e match
  case Src.BinOp(op, e1, e2) =>
    interpk(e1, env, v1 =>
      interpk(e2, env, v2 =>
        val SrcVal.Num(n1) = v1
        val SrcVal.Num(n2) = v2
        val res =
          if (op == "+") SrcVal.Num(v1 + v2)
          ...
        k(res) // invoking the continuation with result
      ))
```

From CPS interpreter to CPS transformer

- CPS interpreter: all intermediate results are named/passed via continuation functions
- To CPS transformer: instead of evaluating the source term e to a source value, we construct the corresponding target term representing the same computation in CPS

Defining the target language

Target language

$$
\begin{array}{rcll}
v & ::= & n \mid x \mid \lambda x k.c \mid \lambda_c x.c \mid \mathsf{halt} & \textbf{values} \\
e & ::= & v \mid v_1 + v_2 \mid v_1 - v_2 \mid \cdots & \textbf{simple expressions} \\
c & ::= & \mathsf{let}\ x = e\ \mathsf{in}\ c \mid v_1\ v_2 \mid v_1\ v_2\ v_3 & \textbf{term/computation}
\end{array}
$$

Defining the target language in Scala

```
enum TgtVal:
  case Num(i: Int)
  case Var(x: String)
  case Lam(x: String, k: String, e: Tgt)
  case LamK(x: String, e: Tgt)
  case Halt()
enum TgtExp:
  case Val(v: TgtVal)
  case BinOp(op: String, v1: TgtVal, v2: TgtVal)
enum Tgt:
  case Let(x: String, e1: TgtExp, e2: Tgt)
  case App(e1: TgtVal, e2: TgtVal, k: TgtVal)
  case AppK(k: TgtVal, v: TgtVal)
```

CPS transformer

- Transformer has similar structure as the CPS interpreter
- Transformer takes a continuation `k` of type `TgtVal => Tgt` (in the meta language)
- Instead of returning a `SrcVal`, it constructs a `Tgt` term

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.Num(i) => ...
  case Src.Var(x) => ...
  case Src.Lam(x, body) => ...
  case Src.App(e1, e2) => ...
  case Src.BinOp(op, e1, e2) => ...
```

CPS transformer (Num and Var)

- Direct translation for numbers and variables
- Invoking the continuation `k` to finish the rest of translation

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.Num(i) => k(TgtVal.Num(i))
  case Src.Var(x) => k(TgtVal.Var(x))
  ...
```

CPS transformer (Lam)

- Shape: every target lambda takes an extra continuation argument

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.Lam(x, body) =>
    // generate a fresh variable name for the continuation
    val c: String = fresh("c")
    val trBody: Tgt = ???
    val trLam: TgtVal = TgtVal.Lam(x, c, trBody)
    ...
```

CPS transformer (Lam)

- Shape: every target lambda takes an extra continuation argument
- Transform the body recursively, but what should be the continuation?

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.Lam(x, body) =>
    val c: String = fresh("c")
    val trBody: Tgt = trans(body, v => ???)
    val trLam: TgtVal = TgtVal.Lam(x, c, trBody)
    ...
```

CPS transformer (Lam)

- Shape: every target lambda takes an extra continuation argument
- Transform the body recursively, but what should be the continuation?
- Example: $[\![\lambda x.42]\!] = \lambda x c.c(42)$

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.Lam(x, body) =>
    val c: String = fresh("c")
    val trBody: Tgt = trans(body, v => ???)
    val trLam: TgtVal = TgtVal.Lam(x, c, trBody)
    ...
```

CPS transformer (Lam)

- Shape: every target lambda takes an extra continuation argument
- Transform the body recursively; the result v is applied to continuation c
- Example: $[\![\lambda x.42]\!] = \lambda x c.c(42)$

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.Lam(x, body) =>
    val c: String = fresh("c")
    val trBody: Tgt = trans(body, v => Tgt.AppK(TgtVal.Var(c), v))
    val trLam: TgtVal = TgtVal.Lam(x, c, trBody)
    k(trLam)
```

CPS transformer (Lam)

- Shape: every target lambda takes an extra continuation argument
- Transform the body recursively; the result `v` is applied to continuation `c`
- Finally, pass the transformed lambda to the outer continuation `k`

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.Lam(x, body) =>
    val c: String = fresh("c")
    val trBody: Tgt = trans(body, v => Tgt.AppK(TgtVal.Var(c), v))
    val trLam: TgtVal = TgtVal.Lam(x, c, trBody)
    k(trLam)
```

CPS transformer (App)

- Similar to the CPS interpreter, we recursively transform `e1` and `e2`
- We now have the CPS-transformed values `v1` and `v2`; the result of applying `v1` to `v2` should be passed to the continuation `k`, how?

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.App(e1, e2) =>
    trans(e1, v1 =>
      trans(e2, v2 =>
        ???
      ))
```

CPS transformer (App)

- Similar to the CPS interpreter, we recursively transform `e1` and `e2`
- We now have the CPS-transformed values `v1` and `v2`; the result of applying `v1` to `v2` should be passed to the continuation `k`, how?
- Example: $[\![(\lambda x.42)\ v]\!] = (\lambda x c.c(42))\ v\ (\lambda_c w. \ldots))$

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.App(e1, e2) =>
    trans(e1, v1 =>
      trans(e2, v2 =>
        ???
      ))
```

CPS transformer (App)

- Similar to the CPS interpreter, we recursively transform `e1` and `e2`
- We now have the CPS-transformed values `v1` and `v2`; the result of applying `v1` to `v2` should be passed to the continuation `k`
- Construct a new continuation: name the intermediate result `w` with `LamK`, applies the current `k` (meta-function) to that intermediate result

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.App(e1, e2) =>
    trans(e1, v1 =>
      trans(e2, v2 =>
        val w = fresh("w")
        val newK: TgtVal = TgtVal.LamK(w, k(TgtVal.Var(w)))
        Tgt.App(v1, v2, newK)
      ))
```

CPS transformer (BinOp)

- Similar to the CPS interpreter, we recursively transform `e1` and `e2`
- We now have the CPS-transformed values `v1` and `v2`; the result `TgtExp.BinOp(op, v1, v2)` should be passed to the continuation `k`, how?

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.BinOp(op, e1, e2) =>
    trans(e1, v1 =>
      trans(e2, v2 =>
        ???
      ))
```

CPS transformer (BinOp)

- Similar to the CPS interpreter, we recursively transform `e1` and `e2`
- We now have the CPS-transformed values `v1` and `v2`; the result `TgtExp.BinOp(op, v1, v2)` should be passed to the continuation `k`, how?
- In principle, we could have something similar to App:

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.BinOp(op, e1, e2) =>
    trans(e1, v1 =>
      trans(e2, v2 =>
        val x = fresh("x")
        val newK = TgtVal.LamK(x, k(TgtVal.Var(x)))
        Tgt.BinOp(v1, v2, newK) // Wrong: BinOp doesn't take a cont.
      ))
```

CPS transformer (BinOp)

- Similar to the CPS interpreter, we recursively transform `e1` and `e2`
- We now have the CPS-transformed values `v1` and `v2`; the result `TgtExp.BinOp(op, v1, v2)` should be passed to the continuation `k`
- Since `TgtExp.BinOp` is a primitive operation, we construct a `Let` binding to hold the result, and pass that to `k`

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.BinOp(op, e1, e2) =>
    trans(e1, v1 =>
      trans(e2, v2 =>
        val x = fresh("x")
        val rhs = TgtExp.BinOp(op, v1, v2)
        val body = k(TgtVal.Var(x))
        Tgt.Let(x, rhs, body)
      ))
```

CPS transformer

- We have defined:

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = ...
```

- Need to provide the initial continuation:

```
def trans(e: Src): Tgt = trans(e, v => Tgt.AppK(TgtVal.Halt(), v))
```

CPS transformer

- Everything in less than 20 lines of code!

```
def trans(e: Src, k: TgtVal => Tgt): Tgt = e match
  case Src.Num(i) => k(TgtVal.Num(i))
  case Src.Var(x) => k(TgtVal.Var(x))
  case Src.Lam(x, body) =>
    val c = fresh("c")
    k(TgtVal.Lam(x, c, trans(body, v => Tgt.AppK(TgtVal.Var(c), v))))
  case Src.App(e1, e2) =>
    trans(e1, v1 =>
      trans(e2, v2 =>
        val w = fresh("w")
        Tgt.App(v1, v2, TgtVal.LamK(w, k(TgtVal.Var(w))))
      ))
  case Src.BinOp(op, e1, e2) =>
    trans(e1, v1 =>
      trans(e2, v2 =>
        val x = fresh("x")
        Tgt.Let(x, TgtExp.BinOp(op, v1, v2), k(TgtVal.Var(x)))
      ))
```

CPS transformer

A few remarks:

- The transformer itself uses continuations
  - Translation follows the evaluation order of the source term
  - The remaining translation is captured by the continuation `k` at the meta-level
- The transformer is a higher-order
  - Vs. first-order transformation that does not use continuation functions
  - Refunctionalization and defunctionalization

CPS transformer

- Now with a bit more formal notations:

$$
\begin{aligned}
[\![\cdot]\!] &: \mathsf{Src} \to (\mathsf{TgtVal} \to \mathsf{Tgt}) \to \mathsf{Tgt} \\
[\![n]\!]\ k &= k(n) \\
[\![x]\!]\ k &= k(x) \\
[\![\lambda x.t]\!]\ k &= k(\lambda x c.[\![t]\!]\ (\lambda v.c\ v)) \\
[\![t_1\ t_2]\!]\ k &= [\![t_1]\!]\ (\lambda v_1.[\![t_2]\!]\ (\lambda v_2.v_1\ v_2\ (\lambda_c w.k(w)))) \\
[\![t_1 + t_2]\!]\ k &= [\![t_1]\!]\ (\lambda v_1.[\![t_2]\!]\ (\lambda v_2.\mathsf{let}\ x = v_1 + v_2\ \mathsf{in}\ k(x)))
\end{aligned}
$$

CPS transformer

- In-class example:

$$
\begin{aligned}
& [\![(\lambda x.x + 1)\ (2 \times 3)]\!](\lambda v.\mathsf{halt}\ v) \\
= {} & [\![\lambda x.x + 1]\!]\ (\lambda v_1.[\![2 \times 3]\!]\ (\lambda v_2.v_1\ v_2\ (\lambda_c w.(\lambda v.\mathsf{halt}\ v)\ w))) \\
= {} & ?
\end{aligned}
$$

Summary

Summary

- CPS transformation from CPS interpreter
- Use continuation to represent the rest transformation

Next time

- Alternatives to CPS
- Want to learn more about CPS in an actual compiler? Take CS107 in Spring 2026!