The Functional Correspondence Applied: An Implementation of a Semantics Transformer

Maciej Buszka

Instytut Informatyki UWr

18.11.2020

The Functional Correspondence

and an introduction to the meta-language

The Functional Correspondence

- A method for deriving abstract machines
- Takes a definitional interpreter
- Produces an encoding of an abstract machine
- Introduced by Danvy et al. in *A functional correspondence between evaluators and abstract machines*
- Based on two source-to-source transformations
  - Translation to continuation-passing style (CPS)
  - Defunctionalization

Meta-language by Example: CBV λ-calculus

```
(def-data Term
  String              ;; variable
  {Abs String Term}   ;; lambda abstraction
  {App Term Term})    ;; application

;; Environment is encoded as a partial function

(def extend (env x v) ... ) ;; extends env at x with value v

(def eval (env term)
  (match term
    ([String x]   (env x))
    ({Abs x body} (fun (v) (eval (extend env x v) body)))
    ({App fn arg} ((eval env fn) (eval env arg)))))
```

Transformation Steps We Aim For

(Also a reminder of how functional correspondence works)

After CPS Translation

```
(def eval (env term cont)
  (match term
    ([String x] (cont (env x)))
    ({Abs x body}
      (cont (fun (v cont)
        (eval (extend env x v) body cont))))
    ({App fn arg}
      (eval env fn (fun (fn)
        (eval env arg (fun (arg) (fn arg cont))))))))
```

After Defunctionalization

```
(def eval (env term cont)
  (match term
    ([String x]   (continue cont (env x)))
    ({Abs x body} (continue cont {Fun body env x}))
    ({App fn arg} (eval env fn {App1 arg cont env}))))

(def apply (fn v cont)
  (match fn ({Fun body env x}
    (eval (extend env x v) body cont))))

(def continue (cont val)
  (match cont
    ({App1 arg cont env} (eval env arg {App2 cont val}))
    ({App2 cont fn}      (apply fn val cont))
    ({Halt}              val)))
```

The Result

- An abstract machine encoded as mutually tail-recursive functions
- Uses abstract environment (encoded as a function)
  - `extend` is still higer-order
  - `extend` and functions representing environment remain in direct style

Desired Features

- Metalanguage expressivity
  - Mutually recursive, anonymous and higher-order functions
  - Records and pattern matching
  - Dynamic (strong) typing
- Automation
  - Function space detection
  - CPS and defunctionalization
  - Sensible name generation
- User's control over transformation
  - Naming of records for defunctionalized lambdas, apply functions
  - Disabling defunctionalization and/or cps transformation of chosen functions

Why Control is Needed

```
(def extend (env y v cont1) (continue cont1 {Ext env v y}))
(def lookup (fn1 x cont2) ... (continue2 ...))

(def eval (env term cont3)
  (match term
    ([String x] (lookup env x cont3))
    ({Abs x body} (continue2 cont3 {Fun body env x}))
    ({App fn arg} (eval env fn {App1 arg cont3 env}))))

(def continue2 (fn4 var3)
  (match fn4
    ({App2 cont3 var2} (apply var2 var3 cont3))
    ({App1 arg cont3 env} (eval env arg {App2 cont3 var3}))
    ({Halt } var3)))

(def apply (fn2 v cont4) ... (extend ...))
```

Automating the Functional Correspondence

(A particular approach)

In a Nutshell

- Transformation to ANF
- Control flow analysis
- Transformation to CPS
- Control flow analysis
- Defunctionalization
- Inlining administrative let-bindings

Transformation to ANF

- Assumes left-to-right call-by-value semantics
- Let-bind every intermediate result
- Eases program analysis and further transformations
- Preserves tail calls

Abstract syntax of the meta-language

$$x, y, z \in \mathit{Var} \qquad r \in \mathit{StructName}$$

$$s \in \mathit{String} \qquad b \in \mathit{Int} \cup \mathit{Boolean} \cup \mathit{String}$$

$$\mathit{Tp} \ni \mathit{tp} ::= \texttt{String} \mid \texttt{Integer} \mid \texttt{Boolean}$$

$$\mathit{Pattern} \ni p ::= x \mid b \mid \_ \mid \texttt{\{}r\ p \ldots\texttt{\}} \mid \texttt{[}\mathit{tp}\ x\texttt{]}$$

$$\mathit{Term} \ni t ::= x \mid b \mid \texttt{(fun (}x \ldots\texttt{)}\ t\texttt{)} \mid \texttt{(}t\ t \ldots\texttt{)} \mid \texttt{\{}r\ t \ldots\texttt{\}}$$

$$\mid \texttt{(let}\ p\ t\ t\texttt{)} \mid \texttt{(match}\ t\ \texttt{(}p\ t\texttt{)}\ldots\texttt{)} \mid \texttt{(error}\ s\texttt{)}$$

Transformation to ANF

Abstract syntax in ANF

$$\begin{aligned} \mathit{Com} \ni c \;\; &::= x \mid b \mid (\texttt{fun}\ (x \ldots)\ e) \mid (x\ x \ldots) \\ &\mid \{r\ x \ldots\} \mid (\texttt{match}\ x\ (p\ e) \ldots) \\ \mathit{Anf} \ni e \;\; &::= c \mid (\texttt{let}\ p\ c\ e) \mid (\texttt{error}\ s) \end{aligned}$$

ANF translation

$$\begin{aligned} [\![\cdot]\!]\ \cdot \;\; &: \mathit{Term} \times (\mathit{Com} \to \mathit{Anf}) \to \mathit{Anf} \\ [\![a]\!]\ k &= k\, a \\ [\![(\texttt{fun}\ (x \ldots) t)]\!]\ k &= k\, (\texttt{fun}\ (x \ldots)\ [\![t]\!]\, \mathit{id}) \\ [\![(t_f\ t_{arg} \ldots)]\!]\ k &= [\![t_f]\!]\, [\lambda x_f.\, [\![t_{arg} \ldots]\!]_s\, \lambda(x_{arg} \ldots).k\, (x_f\ x_{arg} \ldots)]_a \\ [\![(\texttt{let}\ x\ t_1\ t_2)]\!]\ k &= [\![t_1]\!]\, \lambda c_1.(\texttt{let}\ x\ c_1\ [\![t_2]\!]\, k) \\ [\![\{r\ t \ldots\}]\!]\ k &= [\![t \ldots]\!]_s\, \lambda(x \ldots).k\, \{r\ x \ldots\} \\ [\![(\texttt{match}\ t\ (p\ t_b))]\!]\ k &= [\![t]\!]\, [\lambda x.k\, (\texttt{match}\ x\ (p\ [\![t_b]\!]\, \mathit{id})]_a \\ [\![(\texttt{error}\ s)]\!]\ \_ &= (\texttt{error}\ s) \end{aligned}$$

Transformation to ANF

Extending atomic continuation

$$
\begin{aligned}
[\cdot]_a \; \cdot \; &: (\mathit{Var} \to \mathit{Anf}) \to \mathit{Com} \to \mathit{Anf} \\
[k]_a \; x &= k\, x \\
[k]_a \; c &= (\texttt{let} \;\; x \;\; c \;\; (k\, x))
\end{aligned}
$$

Sequencing multiple terms

$$
\begin{aligned}
[\![\cdot]\!]_s \; \cdot \; &: \; \mathit{Term}^* \times (\mathit{Var}^* \to \mathit{Anf}) \to \mathit{Anf} \\
[\![t \ldots]\!]_s \; k &= [\![t \ldots]\!]_s^{\epsilon}
\end{aligned}
$$

$$
\begin{aligned}
[\![\epsilon]\!]_s^{x \ldots} \; k &= k\,(x \ldots) \\
[\![t\; t_r \ldots]\!]_s^{x_{acc} \ldots} \; k &= [\![t]\!] \, [\lambda x. \, [\![t_r \ldots]\!]_s^{x_{acc} \ldots x}]_a
\end{aligned}
$$

After ANF Transformation

```
(def eval (env term)
  (match term
    ([String x] (env x))
    ({Abs x body}
      (fun (v)
        (let var1 (extend env x v))
        (eval var1 body)))
    ({App fn arg}
      (let var2 (eval env fn))
      (let var3 (eval env arg))
      (var2 var3))))
```

Control Flow Analysis

- For each expression in a program, find the over-approximation of the set of functions it may evaluate to
- Detects function spaces for defunctionalization
- Enables selective cps translation
- Textbook approaches:
  - Constraint systems
  - Annotated type systems

Abstracting Abstract Machines

- Begin with a CESK* machine extended with patern matching (Control Environment Store Kontinuation pointer)
- Because the program is in ANF:
  - There are only two types of continuations
  - Every proper subexpression will allocate a value in the store
- Provide finite approximations for base types
- Let the store map to sets of value approximations
- Compute the set of configurations reachable from the initial one
- Now for every variable in function position the store contains a set of values to which it may evaluate
- Methodology introduced by Van Horn et al. in *Abstracting Abstract Machines*

AAM – Performance Considerations

- Use a single shared store (corresponds to widening)
- Trim environment in closures and continuations
- Sufficiently fast even with naive fixed-point iteration

Selective Translation to CPS

- Allows the user to specify which functions should be left in direct style
- Functions in direct style may call CPS ones and vice versa
- Quite simple since the program is already in ANF
- Designed not to duplicate code
- Produces a program in ANF

CPS Annotations

```
(def init #:atomic (x) (error "empty environment"))

(def extend #:atomic (env y v)
  (fun #:atomic (x) (match (eq? x y)
    (#t v)
    (#f (env x)))))
```

Selective Translation to CPS

Translation to CPS

$$
\begin{aligned}
[\![x]\!]_c\, k &= (k\ \ x) \\
[\![b]\!]_c\, k &= (\texttt{let}\ x\ b\ (k\ \ x)) \\
[\![\{r\, x \ldots\}]\!]_c\, k &= (\texttt{let}\ y\ \{r\, x \ldots\}\ (k\ \ y)) \\
[\![(\texttt{fun}\ \texttt{\#:atomic}\ (x \ldots) e)]\!]_c\, k &= (\texttt{let}\ y\ (\texttt{fun}\ (x \ldots)\ [\![e]\!]_d)\ (k\ \ y)) \\
[\![(\texttt{fun}\ (x \ldots) e)]\!]_c\, k &= (\texttt{let}\ y\ (\texttt{fun}\ (x \ldots k')\ [\![e]\!]_c\, k')\ (k\ \ y)) \\
[\![(f^l\ x \ldots)]\!]_c\, k &= \begin{cases} (f\ \ x \ldots\ \ k) \\ \quad \text{when } \mathit{noneAtomic}(l) \\ (\texttt{let}\ y\ (f\ \ x \ldots)\ (k\ \ y)) \\ \quad \text{when } \mathit{allAtomic}(l) \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e) \ldots)]\!]_c\, k &= (\texttt{match}\ \ x\ \ (p\ \ [\![e]\!]_c\, k) \ldots) \\
[\![(\texttt{let}\ x\ c\ e)]\!]_c\, k &= \begin{cases} (\texttt{let}\ \ x\ \ [\![c]\!]_d\ \ [\![e]\!]_c\, k) \\ \quad \text{when } \mathit{trivial}(c) \\ (\texttt{let}\ \ k'\ (\texttt{fun}\ (x)\ [\![e]\!]_c\, k)\ [\![c]\!]_c\, k') \\ \quad \text{otherwise} \end{cases} \\
[\![(\texttt{error}\ s)]\!]_c\, k &= (\texttt{error}\ \ s)
\end{aligned}
$$

Selective Translation to CPS

Translation for terms left in direct style

$$
\begin{aligned}
[\![x]\!]_d &= x \\
[\![b]\!]_d &= b \\
[\![\{r\ x \ldots\}]\!]_d &= \{r\ x \ldots\} \\
[\![(\texttt{fun}\ \texttt{\#:atomic}\ (x \ldots) e)]\!]_d &= (\texttt{fun}\ (x \ldots)\ [\![e]\!]_d) \\
[\![(\texttt{fun}\ (x \ldots) e)]\!]_d &= (\texttt{fun}\ (x \ldots k')\ [\![e]\!]_c\ k') \\
[\![(f^l\ x \ldots)]\!]_d &= \begin{cases} (f\ \ x \ldots) \\ \quad \text{when } \mathit{allAtomic}(l) \\ (\texttt{let}\ \ k\ (\texttt{fun}\ (y)\ y)\ (f\ \ x \ldots\ \ k)) \\ \quad \text{when } \mathit{noneAtomic}(l) \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e) \ldots)]\!]_d &= (\texttt{match}\ \ x\ (p\ \ [\![e]\!]_d) \ldots) \\
[\![(\texttt{let}\ x\ (f^l\ y \ldots)\ e)]\!]_d &= \begin{array}{l} (\texttt{let}\ \ k\ (\texttt{fun}\ (z)\ z) \\ \quad (\texttt{let}\ \ x\ (f\ \ y \ldots\ \ k)) \\ \qquad [\![e]\!]_d) \end{array} \quad \text{when } \mathit{noneAtomic}(l) \\
[\![(\texttt{let}\ x\ c\ e)]\!]_d &= (\texttt{let}\ \ x\ \ [\![c]\!]_d\ \ [\![e]\!]_d) \\
[\![(\texttt{error}\ s)]\!]_d &= (\texttt{error}\ \ s)
\end{aligned}
$$

After CPS Translation

```
(def eval (env term cont)
  (match term
    ([String x]
      (let val (env x))
      (cont val))
    ({Abs x body}
      (let val1
        (fun (v cont1)
          (let var1 (extend env x v))
          (eval var1 body cont1)))
      (cont val1))
    ({App fn arg}
      (eval env fn (fun (var2)
        (eval env arg (fun (var3)
          (var2 var3 cont)))))))))
```

Selective Defunctionalization

- Control flow analysis provides all the necessary information
- Trade-offs and gotchas
  - Direct vs indirect calls to top-level functions
  - Primitive operations
- Record name generation
  - For anonymous functions: let the user decide
  - For continuations: use the constructor of current branch (heuristic)
- Result is no longer in ANF

Defunctionalization Annotations

```
(def init #:no-defun (x) (error "empty environment"))

(def extend (env y v)
  (fun #no-defun (x) (match (eq? x y)
    (#t v)
    (#f (env x)))))

(def eval (env term)
  (match term
    ([String x] (env x))
    ({Abs x body} (fun #:name Fun #:apply apply (v) (eval (extend e
    ({App fn arg} ((eval env fn) (eval env arg)))))
```

Selective Defunctionalization

$$
\begin{aligned}
[\![x]\!] &= \begin{cases} \{\texttt{Prim}_x\} & \text{when } \mathit{primOp}(x) \\ \{\texttt{Top}_x\} & \text{when } \mathit{topLevel}(x) \wedge \mathit{defun}(x) \\ x & \text{otherwise} \end{cases} \\
[\![b]\!] &= b \\
[\![\{r\ x \ldots\}]\!] &= \{\texttt{r}\ [\![x]\!] \ldots\} \\
[\![(\texttt{fun}\ (x \ldots) e)^l]\!] &= \begin{cases} \{\texttt{Fun}_l\ \ \mathit{fvs}(e)\} & \text{when } \mathit{defun}(l) \\ (\texttt{fun}\ (x \ldots)\ [\![e]\!]) & \text{otherwise} \end{cases} \\
[\![(f^{l'}\ x \ldots)^l]\!] &= \begin{cases} (f\ [\![x]\!] \ldots) & \text{when } \mathit{primOp}(f) \vee \\ \quad \mathit{topLevel}(f) & \\ (\texttt{apply}_l\ f\ [\![x]\!] \ldots) & \text{else when } \mathit{allDefun}(l') \\ (f\ [\![x]\!] \ldots) & \text{when } \mathit{noneDefun}(l') \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e) \ldots)]\!] &= (\texttt{match}\ x\ (p\ [\![e]\!]) \ldots) \\
[\![(\texttt{let}\ x\ c\ e)]\!] &= (\texttt{let}\ x\ [\![c]\!]\ [\![e]\!]) \\
[\![(\texttt{error}\ s)]\!] &= (\texttt{error}\ s)
\end{aligned}
$$

Selective Defunctionalization

Apply function generation

$$mkBranch(x\ldots, \qquad \delta) = (\{\mathtt{Prim}_\delta\}\ (\delta\ x\ldots))$$
$$mkBranch(x\ldots, (\mathtt{def}\ f\ (y\ldots)\ \mathtt{e})) = (\{\mathtt{Top}_f\}\ (f\ x\ldots))$$
$$mkBranch(x\ldots, (\mathtt{fun}\ (y\ldots)\ e)^l) = (\{\mathtt{Fun}_l\ fvs(e)\}\ [\![e]\!]\,[y \mapsto x\ldots])$$

$$mkApply(l, fn\ldots) = \begin{array}{l} (\mathtt{def}\ \mathtt{apply}_l\ (f\ x\ldots) \\ \quad (\mathtt{match}\ f \\ \qquad mkBranch(x\ldots, fn)\ldots)) \end{array}$$

After Defunctionalization

```
(def eval (env term cont)
  (match term
    ([String x]
      (let val (env x))
      (continue1 cont val))
    ({Abs x body}
      (let val1 {Fun body env x})
      (continue1 cont val1))
    ({App fn arg} (eval env fn {App1 arg cont env}))))
(def apply (fn1 v cont1)
  (match fn1
    ({Fun body env x}
      (let var1 (extend env x v))
      (eval var1 body cont1))))
(def continue1 (fn2 var3)
  (match fn2
    ({App2 cont var2} (apply var2 var3 cont))
    ({App1 arg cont env} (eval env arg {App2 cont var3}))
    ({Halt } var3)))
```

Finishing Touches

- Inline let-bindings generated by transformation
  Only if the variable is used exactly once
- Future work: sugar single branch matches as let bindings

The Result

```
(def eval (env term cont)
  (match term
    ([String x] (continue1 cont (env x)))
    ({Abs x body} (continue1 cont {Fun body env x}))
    ({App fn arg} (eval env fn {App1 arg cont env}))))

(def apply (fn1 v cont1)
  (match fn1 ({Fun body env x} (eval (extend env x v) body cont1))))

(def continue1 (fn2 var3)
  (match fn2
    ({App2 cont var2} (apply var2 var3 cont))
    ({App1 arg cont env} (eval env arg {App2 cont var3}))
    ({Halt } var3)))
```

Demo: Actually Using the Tool

call-by-value normalization by evaluation

Case studies

Language	Interpreter style	Lang. Features	Result
call-by-value λ-calculus	denotational	·	CEK machine
	denotational	integers with add	CEK with add
	denotational, recursion via environment	integers, recursive let-bindings	similar to Reynold's first-order interpreter
	denotational with conts.	shift and reset	two layers of conts.
	denotational, monadic	exceptions with handlers	explicit stack unwinding
	denotational, CPS		pointer to exception handler
	normalization by evaluation	·	strong CEK machine
call-by-name λ-calculus	big-step	·	Krivine machine
call-by-need λ-calculus	big-step (state passing)	memoization	lazy Krivine machine
simple imperative	big-step (state passing)	conditionals, while, assignment	·
micro-Prolog	CPS	backtracking, cut operator	logic engine

Conclusion

- Algorithm
  - Fully automatic transformation
  - Works with interpreters expressed in a higher-order language
  - Allows for fine-grained control over the resulting machine
- Implementation
  - Interpreters embedded in *Racket* source files
  - Users influence transformation using annotations
  - Tested on a selection of interpreters
- Current work: *Coq* formalization
  - First version: encodings of natural semantics
  - Define transformation on a deeply embedded language
  - Leverage *MetaCoq* library to transform to and from shallow embeddings

- Reynolds: *Definitional Interpreters for Higher-Order Programming Languages*
- Ager, Biernacki, Danvy and Midtgaard: *A functional correspondence between evaluators and abstract machines*
- Van Horn and Might: *Abstracting abstract machines*