The Functional Correspondence Applied: An Implementation of a Semantics Transformer

Maciej Buszka

Instytut Informatyki UWr

22.10.2020

The Functional Correspondence

- A method for deriving abstract machines

The Functional Correspondence

- A method for deriving abstract machines
- Takes a definitional interpreter
- Produces an encoding of abstract machine

The Functional Correspondence

- A method for deriving abstract machines
- Takes a definitional interpreter
- Produces an encoding of abstract machine
- Introduced by Danvy et. al. in *A functional correspondence between evaluators and abstract machines*

The Functional Correspondence

- A method for deriving abstract machines
- Takes a definitional interpreter
- Produces an encoding of 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

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

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

In a Nutshell

- Transformation to ANF
- Control flow analysis

In a Nutshell

- Transformation to ANF
- Control flow analysis
- Transformation to CPS

In a Nutshell

- Transformation to ANF
- Control flow analysis
- Transformation to CPS
- Control flow analysis

In a Nutshell

- Transformation to ANF
- Control flow analysis
- Transformation to CPS
- Control flow analysis
- Defunctionalization

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

Transformation to ANF

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

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

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

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

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

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

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

$$\textit{Pattern} \ni p ::= x \mid b \mid \_ \mid \{r\ p...\} \mid [tp\ x]$$

$$\textit{Term} \ni t ::= x \mid b \mid (\texttt{fun}\ (x...)\ t) \mid (t\ t...) \mid \{r\ t...\}$$
$$\mid (\texttt{let}\ p\ t\ t) \mid (\texttt{match}\ t\ (p\ t)...) \mid (\texttt{error}\ s)$$

Transformation to ANF

Abstract syntax in ANF

$$Com \ni c ::= x \mid b \mid (\text{fun } (x \ldots) \ e) \mid (x \ x \ldots)$$
$$\mid \{r \ x \ldots\} \mid (\text{match } x \ (p \ e)\ldots)$$
$$Anf \ni e ::= c \mid (\text{let } p \ c \ e) \mid (\text{error } s)$$

ANF translation

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

Transformation to ANF

Extending atomic continuation

$$[\cdot]_a \ \cdot \ : \ (Var \to Anf) \to Com \to Anf$$
$$[k]_a \ x = k \ x$$
$$[k]_a \ c = (\text{let } x \ c \ (k \ x))$$

Sequencing multiple terms

$$[\![\cdot]\!]_s \ \cdot \ : \ Term^* \times (Var^* \to Anf) \to Anf$$
$$[\![t \ldots]\!]_s \ k = [\![t \ldots]\!]_s^\epsilon$$

$$[\![\epsilon]\!]_s^{x\cdots} \ k = k \ (x \ldots)$$
$$[\![t \ t_r \ldots]\!]_s^{x_{acc}\cdots} \ k = [\![t]\!] \ [\lambda x. \ [\![t_r \ldots]\!]_s^{x_{acc}\cdots x}]_a$$

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

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)

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

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

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

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.

AAM – Performance Considerations

- Use a single shared store (corresponds to widening)

AAM – Performance Considerations

- Use a single shared store (corresponds to widening)
- Trim environment in closures and continuations

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

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

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

$$[\![x]\!]_c \, k \quad = (k \ x)$$

$$[\![b]\!]_c \, k \quad = (\text{let } x \ b \ (k \ x))$$

$$[\![\{r \ x \ldots\}]\!]_c \, k \quad = (\text{let } y \ \{r \ x \ldots\} \ (k \ y))$$

$$[\![(\text{fun \#:atomic } (x \ldots)e)]\!]_c \, k \quad = (\text{let } y \ (\text{fun } (x \ldots) \ [\![e]\!]_d) \ (k \ y))$$

$$[\![(\text{fun } (x \ldots)e)]\!]_c \, k \quad = (\text{let } y \ (\text{fun } (x \ldots k') \ [\![e]\!]_c \, k') \ (k \ y))$$

$$[\![(f^l \ x \ldots)]\!]_c \, k \quad = \begin{cases} (f \ x \ldots \ k) \\ \quad \text{when } noneAtomic(l) \\ (\text{let } y \ (f \ x \ldots) \ (k \ y)) \\ \quad \text{when } allAtomic(l) \end{cases}$$

$$[\![(\text{match } x \ (p \ e)\ldots)]\!]_c \, k \quad = (\text{match } x \ (p \ [\![e]\!]_c \, k)\ldots)$$

$$[\![(\text{let } x \ c \ e)]\!]_c \, k \quad = \begin{cases} (\text{let } x \ [\![c]\!]_d \ [\![e]\!]_c \, k) \\ \quad \text{when } trivial(c) \\ (\text{let } k' \ (\text{fun } (x) \ [\![e]\!]_c \, k) \ [\![c]\!]_c \, k') \\ \quad \text{otherwise} \end{cases}$$

$$[\![(\text{error } s)]\!]_c \, k \quad = (\text{error } s)$$

Selective Translation to CPS

Translation for terms left in direct style

$$\llbracket x \rrbracket_d = x$$

$$\llbracket b \rrbracket_d = b$$

$$\llbracket \{r\ x \ldots\} \rrbracket_d = \{r\ x \ldots\}$$

$$\llbracket (\texttt{fun \#:atomic}\ (x \ldots)e) \rrbracket_d = (\texttt{fun}\ (x \ldots)\ \llbracket e \rrbracket_d)$$

$$\llbracket (\texttt{fun}\ (x \ldots)e) \rrbracket_d = (\texttt{fun}\ (x \ldots k')\ \llbracket e \rrbracket_c\ k')$$

$$\llbracket (f^l\ x \ldots) \rrbracket_d = \begin{cases} (f\ x \ldots) \\ \quad \text{when } \textit{allAtomic}(l) \\ (\texttt{let}\ k\ (\texttt{fun}\ (y)\ y)\ (f\ x \ldots\ k)) \\ \quad \text{when } \textit{noneAtomic}(l) \end{cases}$$

$$\llbracket (\texttt{match}\ x\ (p\ e) \ldots) \rrbracket_d = (\texttt{match}\ x\ (p\ \llbracket e \rrbracket_d) \ldots)$$

$$\llbracket (\texttt{let}\ x\ (f^l\ y \ldots)\ e) \rrbracket_d = \begin{matrix} (\texttt{let}\ k\ (\texttt{fun}\ (z)\ z) \\ (\texttt{let}\ x\ (f\ y \ldots\ k)) \\ \llbracket e \rrbracket_d) \end{matrix} \quad \text{when } \textit{noneAtomic}(l)$$

$$\llbracket (\texttt{let}\ x\ c\ e) \rrbracket_d = (\texttt{let}\ x\ \llbracket c \rrbracket_d\ \llbracket e \rrbracket_d)$$

$$\llbracket (\texttt{error}\ s) \rrbracket_d = (\texttt{error}\ s)$$

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

Selective Defunctionalization

- Control flow analysis provides all the necessary information
- Trade-offs and gotchas
  - ► Direct vs indirect calls to top-level functions

Selective Defunctionalization

- Control flow analysis provides all the necessary information
- Trade-offs and gotchas
  - Direct vs indirect calls to top-level functions
  - Primitive operations

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

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)

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

$$
\llbracket x \rrbracket = \begin{cases} \{\texttt{Prim}_x\} & \text{when } primOp(x) \\ \{\texttt{Top}_x\} & \text{when } topLevel(x) \wedge defun(x) \\ x & \text{otherwise} \end{cases}
$$

$$
\llbracket b \rrbracket = b
$$

$$
\llbracket \{r\ x\ldots\} \rrbracket = \{\texttt{r}\ \llbracket x \rrbracket \ldots\}
$$

$$
\llbracket (\texttt{fun}\ (x\ldots)e)^l \rrbracket = \begin{cases} \{\texttt{Fun}_l\ fvs(e)\} & \text{when } defun(l) \\ (\texttt{fun}\ (x\ldots)\ \llbracket e \rrbracket) & \text{otherwise} \end{cases}
$$

$$
\llbracket (f^{l'}\ x\ldots)^l \rrbracket = \begin{cases} (f\ \llbracket x \rrbracket \ldots) & \text{when } primOp(f) \vee \\ \quad topLevel(f) \\ (\texttt{apply}_l\ f\ \llbracket x \rrbracket \ldots) & \text{else when } allDefun(l') \\ (f\ \llbracket x \rrbracket \ldots) & \text{when } noneDefun(l') \end{cases}
$$

$$
\llbracket (\texttt{match}\ x\ (p\ e)\ldots) \rrbracket = (\texttt{match}\ x\ (p\ \llbracket e \rrbracket)\ldots)
$$

$$
\llbracket (\texttt{let}\ x\ c\ e) \rrbracket = (\texttt{let}\ x\ \llbracket c \rrbracket\ \llbracket e \rrbracket)
$$

$$
\llbracket (\texttt{error}\ s) \rrbracket = (\texttt{error}\ s)
$$

Selective Defunctionalization

$mkBranch(x\ldots, \qquad\qquad\qquad \delta) = (\{\texttt{Prim}_\delta\}\ (\delta\ x\ldots))$

$mkBranch(x\ldots, (\texttt{def}\ f\ (y\ldots)\ e)) = (\{\texttt{Top}_f\}\ (f\ x\ldots))$

$mkBranch(x\ldots, \quad (\texttt{fun}\ (y\ldots)\ e)^l) = (\{\texttt{Fun}_l\ fvs(e)\}\ [\![e]\!]\,[y \mapsto x\ldots])$

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

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

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

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

Thank You