

The Functional Correspondence Applied: An Implementation of a Semantics Transformer

Maciej Buszka

Instytut Informatyki UWr

23.07.2020

Outline

- 1 Introduction
- 2 The Functional Correspondence
 - Introduction
 - CPS
 - Defunctionalization
- 3 The Semantics Transformer
 - Control-flow Analysis
 - A-normal Form
 - Selective Translation to CPS
 - Selective Defunctionalization
- 4 Conclusion

Problem statement

- Abstract Machines

Problem statement

- Abstract Machines
 - ▶ Precisely describe operational properties of a program

Problem statement

- Abstract Machines
 - ▶ Precisely describe operational properties of a program
 - ▶ May serve as an implementation basis

Problem statement

- Abstract Machines
 - ▶ Precisely describe operational properties of a program
 - ▶ May serve as an implementation basis
 - ▶ Hard to create from scratch
- High-level semantics

Problem statement

- Abstract Machines
 - ▶ Precisely describe operational properties of a program
 - ▶ May serve as an implementation basis
 - ▶ Hard to create from scratch
- High-level semantics
 - ▶ Denotational and big-step operational

Problem statement

- Abstract Machines
 - ▶ Precisely describe operational properties of a program
 - ▶ May serve as an implementation basis
 - ▶ Hard to create from scratch
- High-level semantics
 - ▶ Denotational and big-step operational
 - ▶ Concise and abstract

Problem statement

- Abstract Machines
 - ▶ Precisely describe operational properties of a program
 - ▶ May serve as an implementation basis
 - ▶ Hard to create from scratch
- High-level semantics
 - ▶ Denotational and big-step operational
 - ▶ Concise and abstract
 - ▶ May be understood intuitively
- Obtaining an abstract machine corresponding to a given high-level semantics

Problem statement

- Abstract Machines
 - ▶ Precisely describe operational properties of a program
 - ▶ May serve as an implementation basis
 - ▶ Hard to create from scratch
- High-level semantics
 - ▶ Denotational and big-step operational
 - ▶ Concise and abstract
 - ▶ May be understood intuitively
- Obtaining an abstract machine corresponding to a given high-level semantics
 - ▶ Requires proof of correctness

Problem statement

- Abstract Machines
 - ▶ Precisely describe operational properties of a program
 - ▶ May serve as an implementation basis
 - ▶ Hard to create from scratch
- High-level semantics
 - ▶ Denotational and big-step operational
 - ▶ Concise and abstract
 - ▶ May be understood intuitively
- Obtaining an abstract machine corresponding to a given high-level semantics
 - ▶ Requires proof of correctness
 - ▶ Time consuming and non-trivial

Goals

- An algorithm deriving an abstract machine
 - ▶ Takes an interpreter in a functional language
 - ▶ Produces an encoding of an abstract machine
 - ▶ Fully automatic
- A practical tool which implements this algorithm
 - ▶ Gives control over the shape of the result
 - ▶ Generates readable machines
 - ▶ Allows for testing the interpreters

The Functional Correspondence

- A manual method of deriving abstract machines
- Begin with an evaluator
- Finish with an abstract machine
- Based on two source-to-source transformations
 - ▶ Translation to continuation-passing style (CPS)
 - ▶ Defunctionalization
- Successfully applied to a multitude of diverse evaluators

Running example: call-by-name λ -calculus

```
(def eval (expr env)
  (match expr
    ([Integer n] ((env n)))
    ({App f x}
      ((eval f env) (fun () (eval x env)))))
    ({Abs body}
      (fun (x) (eval body (cons x env))))))
```

Translation to CPS

- Goal: expose control-flow of an interpreter
- Classify functions into trivial and serious ones
 - ▶ Serious functions may only be called in tail position
 - ▶ Trivial functions may be called anywhere
- Pass additional argument – the continuation
 - ▶ Specifies what should be done after the function finishes
 - ▶ Allows to express interesting programs while retaining tail-call property

Interpreter in CPS

```
(def eval (expr env cont)
  (match expr
    ([Integer n] ((env n) cont))
    ({App f x}
     (eval f env
           (fun (var3)
              (var3 (fun (cont1) (eval x env cont1)) cont))))))
    ({Abs body}
     (cont (fun (x cont2) (eval body (cons x env) cont2)))))))
```


Defunctionalization

- Goal: produce first-order program
- For each function space
 - ▶ Transform anonymous function definitions into records holding the free variables
 - ▶ Generate top-level function which matches the records and evaluates the bodies
 - ▶ Transform applications of functions in the space into a call to the top-level function

Resulting Machine

```
(def eval (expr env cont)
  (match expr
    ([Integer n] (force (env n) cont))
    ({App f x} (eval f env {App1 cont env x}))
    ({Abs body} (continue cont {Clo body env}))))

(def force (fn cont1)
  (match fn ({Thunk env x} (eval x env cont1)))))

(def apply (fn1 x cont2)
  (match fn1 ({Clo body env} (eval body (cons x env) cont2)))))

(def continue (fn2 var3)
  (match fn2
    ({App1 cont env x} (apply var3 {Thunk env x} cont))
    ({Halt} var3)))
```

Practical Considerations

- Interpreters embedded in *Racket* source files
 - ▶ Natural, functional meta-language with pattern matching
 - ▶ May be executed in REPL, loaded from other modules etc.
 - ▶ Can be tested using *Racket*'s framework
 - ▶ Tests may use full power of the language and macros
- Machine generated using only the source file
 - ▶ No need to write additional configuration, flags, etc.
 - ▶ Shape of the result may be modified by annotating source program
- The tool can pretty print intermediate results
- Names are generated mostly deterministically

Control-flow Analysis

- For each expression in a program, find the over-approximation of the set of functions it may evaluate to
- Exactly matches requirements of defunctionalization
- Textbook approaches
 - ▶ Constraint systems
 - ▶ Annotated type systems
 - ▶ Subjectively hard to adapt to my requirements

Abstracting Abstract Machines

- Derive an analysis from abstract machine
 - ▶ Mechanical, principled process
 - ▶ Easy to adapt various language features
- Results of analysis fit the functional correspondence well
- Reasonable running time on small (100 lines) interpreters even with very naive implementation
- The analysis specified in the thesis' algorithm works on terms in A-normal form

Translation to A-normal Form

- Intermediate representation of programs
- Every expression has only variables as subterms
- Works by let-binding intermediate results
- Simplifies analysis and subsequent transformations

Selective Translation to CPS

- Extension of standard CPS translation
 - ▶ Allow to specify which functions should be left in direct style
 - ▶ Functions in direct style may call CPS ones and vice versa
- Uses control-flow analysis to guide transformation of applications
- Beneficial in practice – machine is not cluttered with control flow of helper functions

CPS Annotations

```
(def cons #:atomic (val env)
  (fun #:atomic (n)
    (match n
      (o val)
      (_ (env (- n 1))))))

(eval term (fun #:atomic (n) (error "empty env")))
```


Selective Defunctionalization

- Extends defunctionalization with option to leave selected function spaces untouched
- Uses control-flow analysis
 - ▶ Generation of apply functions
 - ▶ Guide transformation of applications
 - ▶ Pass references to top-level functions as records where necessary
- Beneficial in practice – pieces of machine may be left abstract

Defunctionalization Annotations

```
(def cons #:atomic (val env)
  (fun #:atomic #:no-defun (n)
    (match n
      (o val)
      (_ (env (- n 1))))))

(eval term (fun #:atomic #:no-defun (n) (error "empty env")))
```

An evaluator for call-by-name λ -calculus

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
 - ▶ Modification of transformation via annotations
 - ▶ Tested on a selection of interpreters
- Further work
 - ▶ Formalization in *Coq*
 - ▶ Transformation of other encodings of semantic formats
 - ▶ Different backends: *C*, \LaTeX
 - ▶ Nondeterministic languages