THE UNIVERSITY OF HONG KONG
DEPARTMENT OF
COMPUTER SCIENCE

# COMPILER INFRASTRUCTURE FOR EFFICIENT IMPLEMENTATION OF FUNCTIONAL LANGUAGES ON JVM

TOMAS TAUBER

PhD Probation Talk; December 4, 2014

# DISCLAIMER

- Mainly based on "**From System F to Java Efficiently!**" (currently under a *double-blinded review process for ACM SIGPLAN PLDI'15*)
- The title + abstract were rephrased and "obfuscated":
  - 2nd order / polymorphic lambda calculus = **System F**
  - "compiler infrastructure…" = **FCore**
  - "new representation of closures" = **IFO**

# ACKNOWLEDGEMENT

- A big thanks to all of you who helped in this work:
  - Xuan (Jeremy) Bi
  - Zhiyuan (George) Shi
  - Dr. Bruno C. d. S. Oliveira
  - Weixin Zhang
  - Our summer interns: Huang Li & Zhenrui (Jerry) Zhang

# OUTLINE

- *Motivation & Background*
- **System F, Closure F, Imperative Functional Objects (IFOs)** + main formalization rules
- **Tail-Call Elimination (TCE)** using **IFOs**
- *Implementation Overview & Evaluation*
- *Related Work, Future Plans, Conclusion with Q&A*

# FUNCTIONAL PROGRAMMING I

- "The Future is Parallel, and the Future of Parallel is **Declarative**" — Simon Peyton Jones

- **Declarative** example 1:

```sql
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
    FROM suppliers
    RIGHT OUTER JOIN orders
    ON suppliers.supplier_id = orders.supplier_id
    WHERE orders.order_status = 'INCOMPLETE'
    ORDER BY orders.order_date DESC;
```

# FUNCTIONAL PROGRAMMING II

- **Declarative** example II:

$$y = x*x$$

y fx 1

| | A | B | C | D |
|---|---|---|---|---|
| 1 | x | y | | |
| 2 | 1 | 1 | | |
| 3 | 2 | 4 | | |
| 4 | 3 | 9 | | |
| 5 | 4 | 16 | | |
| 6 | 5 | 25 | | |

```
Option Explicit
Sub sq()
 Dim n As Integer
 Dim j As Integer
 Dim z As Variant

 'Count size of array
   n = Range("x").Count

 'Do loop
   For j = 1 To n
     z = Range("x").Cells(j).Value
     Range("y").Cells(j).Value = z * z
   Next j
End Sub
```

# FUNCTIONAL PROGRAMMING III

**FP** is **declarative** and **general**:

```
quicksort :: Ord a => [a] -> [a]
quicksort []      = []
quicksort (p:xs) =
(quicksort lesser)
 ++ [p] ++ (quicksort greater)
    where
        lesser  =
          filter (< p) xs
        greater =
          filter (>= p) xs
```

```
// To sort array a[] of size n: qsort(a,0,n-1)

void qsort(int a[], int lo, int hi)
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    a[hi] = a[l];
    a[l] = p;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

# INDUSTRIAL FP TRENDS

**New languages**

F#

Scala

...

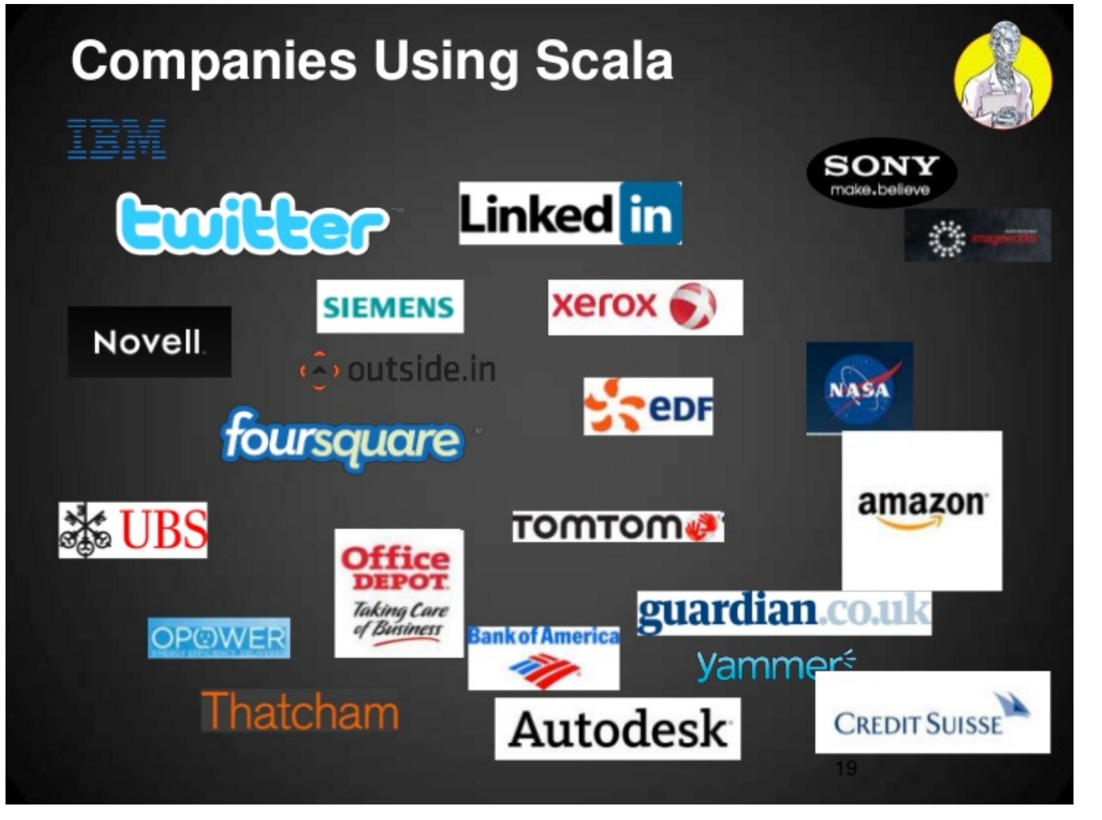
**New features**

C11

C++11

C++14

...



# PARADIGM COMPARISON

## Object-Oriented (imperative)

- Data and the operations upon it are tightly coupled
- Objects hide their implementation of operations from other objects via their interfaces
- The central model for abstraction is the data itself
- The central activity is composing new objects and extending existing objects by adding new methods to them

★ **Imperative algorithms with loops, mutable variables, …**

## Functional (declarative)

- Data is only loosely coupled to functions
- Functions hide their implementation, and the language's abstractions speak to functions and the way they are combined or expressed
- The central model for abstraction is the function, not the data structure.
- The central activity is writing new functions

★ **Declarative algorithms with recursion, pattern matching, first class functions, …**

# JAVA VIRTUAL MACHINE

- Stack-based VM originally designed for Java (OO)
- *Attractive target* for language implementors:
  - **Mature**
  - **Multi-platform**
  - **Many libraries and tools**

# FP ON JVM

- **High-profile JVM languages are functional**:
  - *Scala*, *Clojure*, *Xtend*, …
- *Java 8* has "first-class functions" and lazy streams
- Why are functional programmers still unhappy?

# ENCODING OF FUNCTIONS

- "Functions as (static) **methods**":
  - Straightforward and efficient, but **not general:** not "first-class", no higher-order functions, no partial applications / currying, …
- "Functions as Objects" (**FAO**):

```
interface FAO { Object apply(Object arg);}
```

  - **Inefficient** and avoided, unless necessary

# ENCODING OF FUNCTIONS

- "Functions as (static) **methods**":

```
map :: (a -> b) -> [a] -> [b]
```

  - Straightforward and efficient, but **not general:** not "first-class", no higher-order functions, no partial applications / currying, …

- "Functions as Objects" (**FAO**):

```
interface FAO { Object apply(Object arg);}
```

  - **Inefficient** and avoided, unless necessary

# ENCODING OF FUNCTIONS

- "Functions as (static) **methods**":

```
map :: (a -> b) -> [a] -> [b]
```

  - Straightforward and efficient, but **not general:** not "first-class", no higher-order functions, no partial applications / currying, …

- "Functions as Objects" (**FAO**):

```
interface FAO { Object apply(Object arg);}
```

  - **Inefficient** and avoided, unless necessary

# ENCODING OF FUNCTIONS

- "Functions as (static) **methods**":

```
map :: (a -> b) -> [a] -> [b]
```

  - Straightforward and efficient, but **not general:** not "first-class", no higher-order functions, no partial applications / currying, …

```
compare :: Ord a => a -> a -> Ordering
```

- "Functions as Objects" (**FAO**):

```
interface FAO { Object apply(Object arg);}
```

  - **Inefficient** and avoided, unless necessary

# ENCODING OF FUNCTIONS

- "Functions as (static) **methods**":

```
map :: (a -> b) -> [a] -> [b]
```

  - Straightforward and efficient, but **not general:** not "first-class", no higher-order functions, no partial applications / currying, …

```
compare :: Ord a => a -> a -> Ordering
```

- "Functions as Objects" (**FAO**):

```
interface FAO { Object apply(Object arg);}
```

  - **Inefficient** and avoided, unless necessary

# ENCODING OF FUNCTIONS

- "Functions as (static) **methods**":

```
map :: (a -> b) -> [a] -> [b]
```

  - Straightforward and efficient, but **not general:** not "first-class", no higher-order functions, no partial applications / currying, ...

```
compare :: Ord a => a -> a -> Ordering
let compareWithHundred x = compare 100 x
```

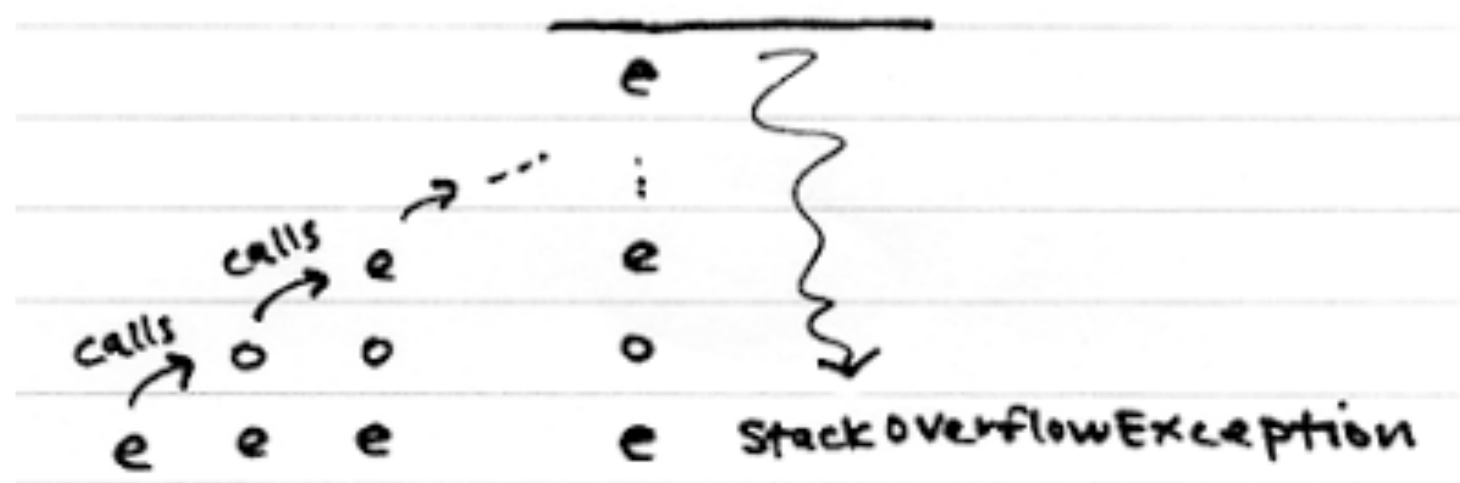
- "Functions as Objects" (**FAO**):

```
interface FAO { Object apply(Object arg);}
```

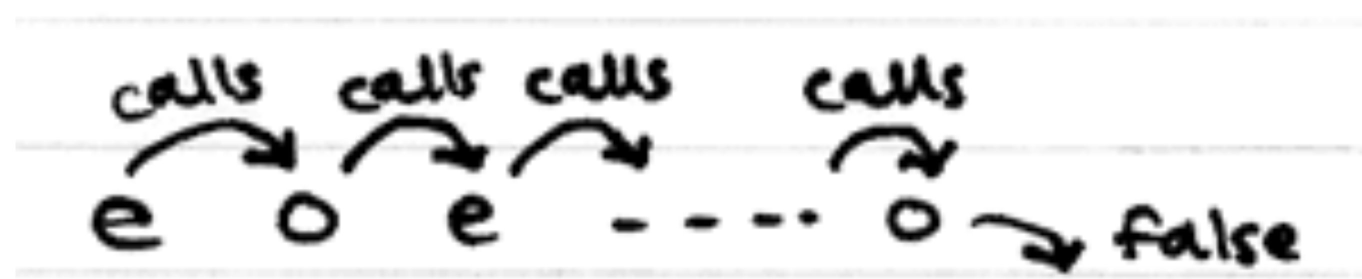
  - **Inefficient** and avoided, unless necessary

# TAIL-CALL ELIMINATION: OVERVIEW

*Default JVM methods*:




*What we want*:




Important for:

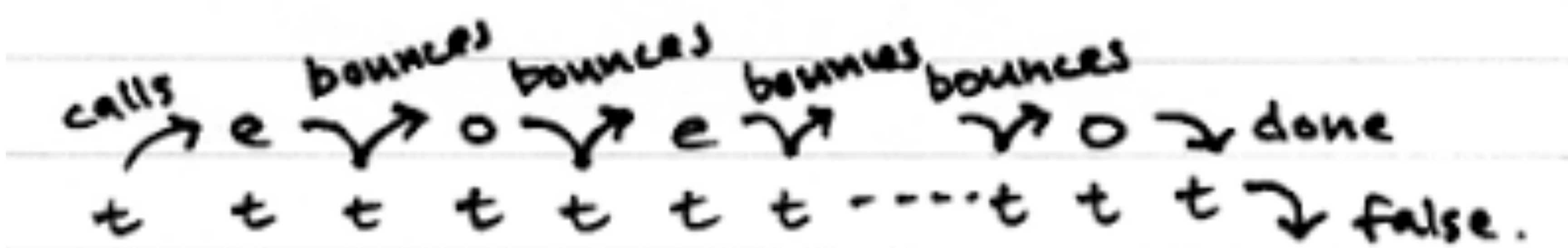
1) **correctness**

2) **time optimization**

3) **space optimization**

# TAIL-CALL ELIMINATION: JVM APPROACHES

- TCE is fairly straightforward in C/Assembly, challenging in JVM due to its design (no control over stack, limited jumps…)
- "*Put everything in one method*": impractical
- *Self-recursion* (@tailrec in Scala): not general




- *Trampolines*: general, but not time and space efficient

# PROBLEM STATEMENT

- Due to the current encoding approaches and inefficient TCE, *programmers may* ***need to avoid idiomatic FP*** *on JVM.*

- With an alternative encoding, *programmers can* ***use idiomatic FP on JVM seamlessly*** *without compromises*:

    - Uniform function representation

    - Low time and memory performance overhead

# SIMPLEST FUNCTIONAL LANGUAGES

- Introduced in 1930s by Alonzo Church
- (Untyped) **Lambda Calculus** (basis for LISP):

$$
\begin{array}{rcll}
e & ::= & x & \text{Variables} \\
 & | & \lambda x.e & \text{Functions} \\
 & | & e\ e & \text{Function applications}
\end{array}
$$

- *Equivalent to Turing Machines* (yet much simpler)

# EXTENSIONS

- **Simply Typed Lambda Calculus** (Church, 1940):

| Terms | | | |
|---|---|---|---|
| t | ::= | $x$ | Variable |
| | \| | $t\ t$ | Function application |
| | \| | $\lambda x : \tau . t$ | Lambda abstraction |
| **Types** | | | |
| $\tau$ | ::= | $T$ | Primitive type |
| | \| | $\tau \to \tau$ | Function |

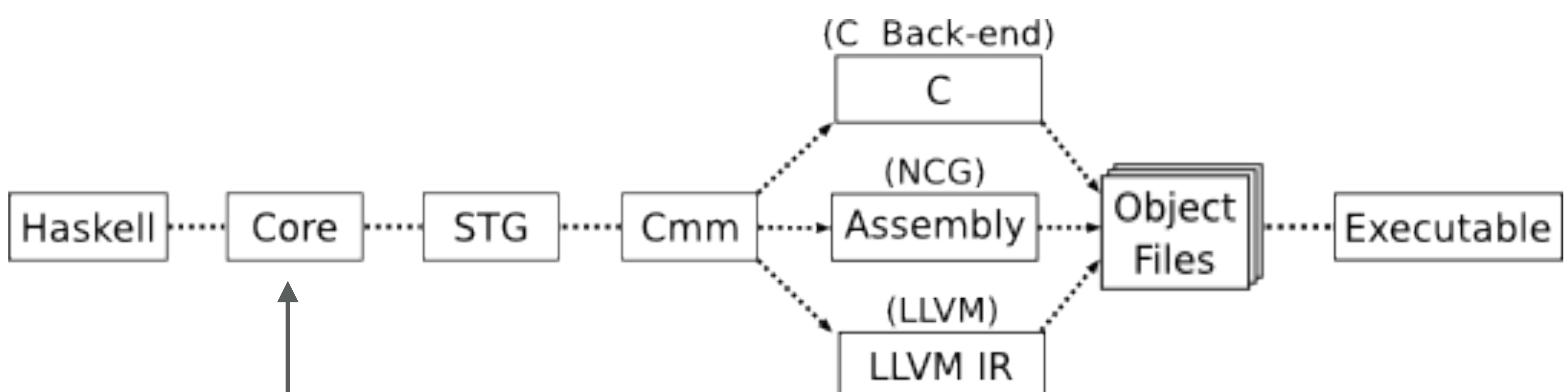
- Polymorphic Lambda Calculus AKA **System F** (Girard in 1972 and Reynolds in 1974):

**Types** $\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha . \tau$

**Expressions** $e ::= x \mid \lambda(x : \tau).e \mid e_1\ e_2 \mid \Lambda \alpha . e \mid e\ \tau$

# SYSTEM F

- Simple, yet expressive enough
- Compilers of statically typed functional languages use it as their intermediate form, for example:




**This is based on System F!**

# HOW TO TRANSLATE IT TO JAVA?

- No time to show the complete translation (full details in the paper)

- Function encoding (**IFO**):

```
abstract class Function {
   Object arg, res;
   abstract void apply();
}
```

**Example**

$$id \equiv (\lambda x : Int).\, x$$

$$\downarrow$$

```
Function id = new Function() {
  void apply() { res = arg; }
};
```

# TWO MORE EXAMPLES

$$const \equiv \lambda A\ (x : A)\ (y : A).x$$

```
Function constant = new Function() {
  void apply() {
    res = new Function() {
       void apply() {res = constant.arg;}
    };
  }
};
```

$$three \equiv const\ 3$$

```
constant.arg = 3;
constant.apply();
```

# TRANSLATION NOTE 1

- We first translate System F to Closure F:

$$const \equiv \Lambda A.(\lambda x : A).(\lambda y : A).x$$

$$\downarrow$$

$$const \equiv \lambda A\ (x : A)\ (y : A).x$$

---

| | |
|---|---|
| **Types** | $\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\tau$ |
| **Expressions** | $e ::= x \mid \lambda(x : \tau).e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ \tau$ |

$$\downarrow$$

| | |
|---|---|
| **Types** | $T ::= \alpha \mid \forall\Delta.T$ |
| **Expressions** | $E ::= x \mid \lambda\Delta.E \mid E\ E \mid E\ T$ |
| **Binders** | $\Delta ::= \epsilon \mid \Delta(x : T) \mid \Delta\alpha$ |

# TRANSLATION NOTE II

- Type-directed translation from Closure F to Java is defined using inductive rules, e.g.:

$$\boxed{\Gamma \vdash E : T \leadsto J \textbf{ in } S}$$

$$(\texttt{CJ-App})\quad \frac{\begin{array}{c}\Gamma \vdash E_1 : \forall(x : T_2)\Delta.T_1 \leadsto J_1 \textbf{ in } S_1 \\ \Gamma \vdash E_2 : T_2 \leadsto J_2 \textbf{ in } S_2 \qquad \Delta; T_1 \Downarrow T_3 \\ f,\ x_f\ fresh\end{array}}{\Gamma \vdash E_1\ E_2 : T_3 \leadsto x_f \textbf{ in } S_1 \uplus S_2 \uplus S_3}$$

```
S3 := {
  Function f = J1;
  f.arg = J2;
  f.apply();
  ⟨T3⟩ xf = (⟨T3⟩) f.res;}
```

Translation Environments:

$$\Gamma ::= \epsilon \mid \Gamma\,(x_1 : T \mapsto x_2) \mid \Gamma\alpha$$

# TAIL-CALL ELIMINATION: EXAMPLE 1

$$fact \equiv \lambda(n : Int)(acc : Int).\textbf{if } (n = 0) \textbf{ then } acc \textbf{ else } fact\ (n - 1)\ (n * acc)$$

```
// naive tail recursive factorial
class Fact extends Function
    {
      Function fact = this;
      public void apply ()
      {
        class FactI extends Function
        {
          public void apply ()
          {
            if (arg == 0) //n
            {
              res = fact.arg;
            }
            else
            {
              fact.arg = fact.arg*arg;
              fact.apply();
              Function facti = fact.res;
              facti.arg = arg - 1;
              facti.apply();
              res = facti.res;
            }
          }
        }
        res = new Fact();
      }
    }
```

```
// tail call elimination
class Fact extends Function
    {
      Function fact = this;
      {
        class FactI extends Function
        {
          public void apply ()
          {
            if (arg == 0) //n
            {
              res = fact.arg; //acc
            }
            else
            {
              fact.arg = arg*fact.arg;
              Function facti = fact.res;
              facti.arg = arg - 1;
              Next.next = facti;
            }
          }
        }
        res = new Fact();
      }
      public void apply ()
      {
      }
    }
```

# TAIL-CALL ELIMINATION: EXAMPLE 2

$$even \equiv \lambda(n : Int).\ \textbf{if}\ (n = 0)\ \textbf{then}\ true\ \textbf{else}\ odd(n-1)$$
$$odd \equiv \lambda(n : Int).\ \textbf{if}\ (n = 0)\ \textbf{then}\ false\ \textbf{else}\ even(n-1)$$

```
//Naive even and odd
Function even = new Function() {
 void apply() {
  Integer n = (Integer) arg;
  if (n == 0)
   res = true;
  else {
   odd.arg = n-1;
   odd.apply();
   res = odd.res;
  }}
};
Function odd = new Function() {
 void apply() {
  Integer n = (Integer) arg;
  if (n == 0)
   res = false;
  else {
   even.arg = n-1;
   even.apply();
   res = even.res;
  }}
};
```

```
// tail call elimination
Function teven = new Function() {
 void apply() {
  Integer n = (Integer) arg;
  if (n == 0)
   res = true;
  else {
   todd.arg = n-1;
   // tail call
   Next.next = todd;
  }}
};
Function todd = new Function() {
 void apply() {
  Integer n = (Integer) arg;
  if (n == 0)
   res = false;
  else {
   teven.arg = n-1;
   // tail call
   Next.next = teven;
  }}
};
```

# TAIL-CALL ELIMINATION: TRANSLATION I

- We detect tail calls based on the tail call context:

$$E ::= x \mid \lambda\Delta.E \mid E\,E \mid E\,T$$

- And modify the CJ-App rule:

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : \forall(x : T_2)\Delta.T_1 \rightsquigarrow J_1 \;\mathbf{in}\; S_1 \\ \Gamma \vdash E_2 : T_2 \rightsquigarrow J_2 \;\mathbf{in}\; S_2 \qquad \Delta; T_1 \Downarrow T_3 \\ x_f,\; f,\; c\; fresh\end{array}}{\Gamma \vdash E_1\,E_2 : T_3 \rightsquigarrow x_f \;\mathbf{in}\; S_1 \uplus S_2 \uplus S_3}$$

# TAIL-CALL ELIMINATION: TRANSLATION II

**TC**

$S_3$ := {
```
    Function f = J1;
    f.arg = J2;
    Next.next = f;
}
```

**Non-TC**

$S_3$ := {
```
    Function f = J1;
    f.arg = J2;
    Next.next = f;
    Function c;
    Object x_f;
    do {
      c = Next.next;
      Next.next = null;
      c.apply();
    } while (Next.next != null);
    x_f = c.res;
}
```

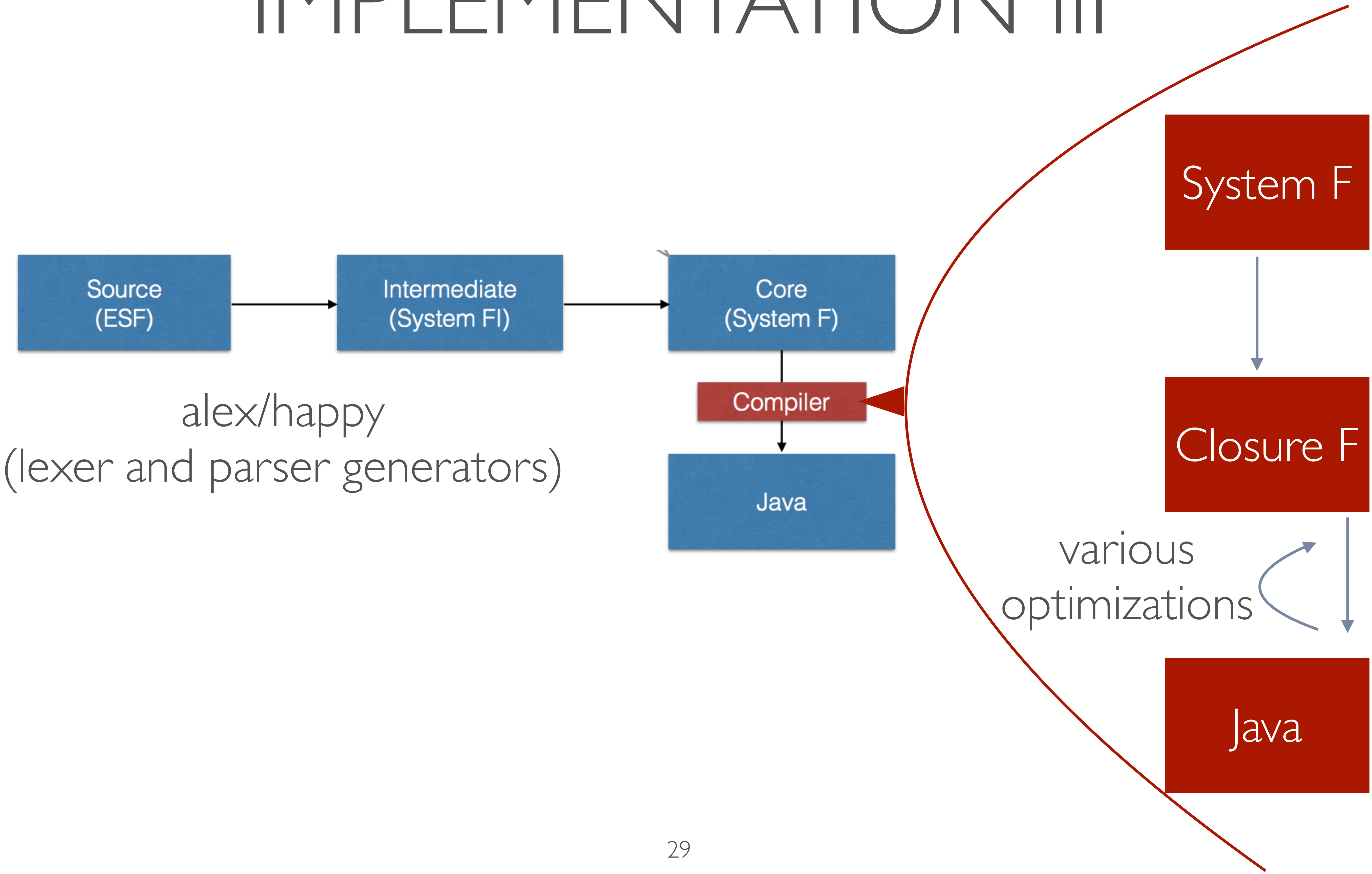
# IMPLEMENTATION I

- **Haskell**: translation rules resemble the code in it
- **System F extended** with fixpoints, conditionals, primitive types and operations, tuples, let bindings, and basic Java interoperability
- Other **standard optimizations**: multi-argument functions, inlining of definitions, partial evaluation

# IMPLEMENTATION II

- Test suite with 118 tests
- Example programs
- Used by all my colleagues here doing PL research
- Open-source soon!

# IMPLEMENTATION III

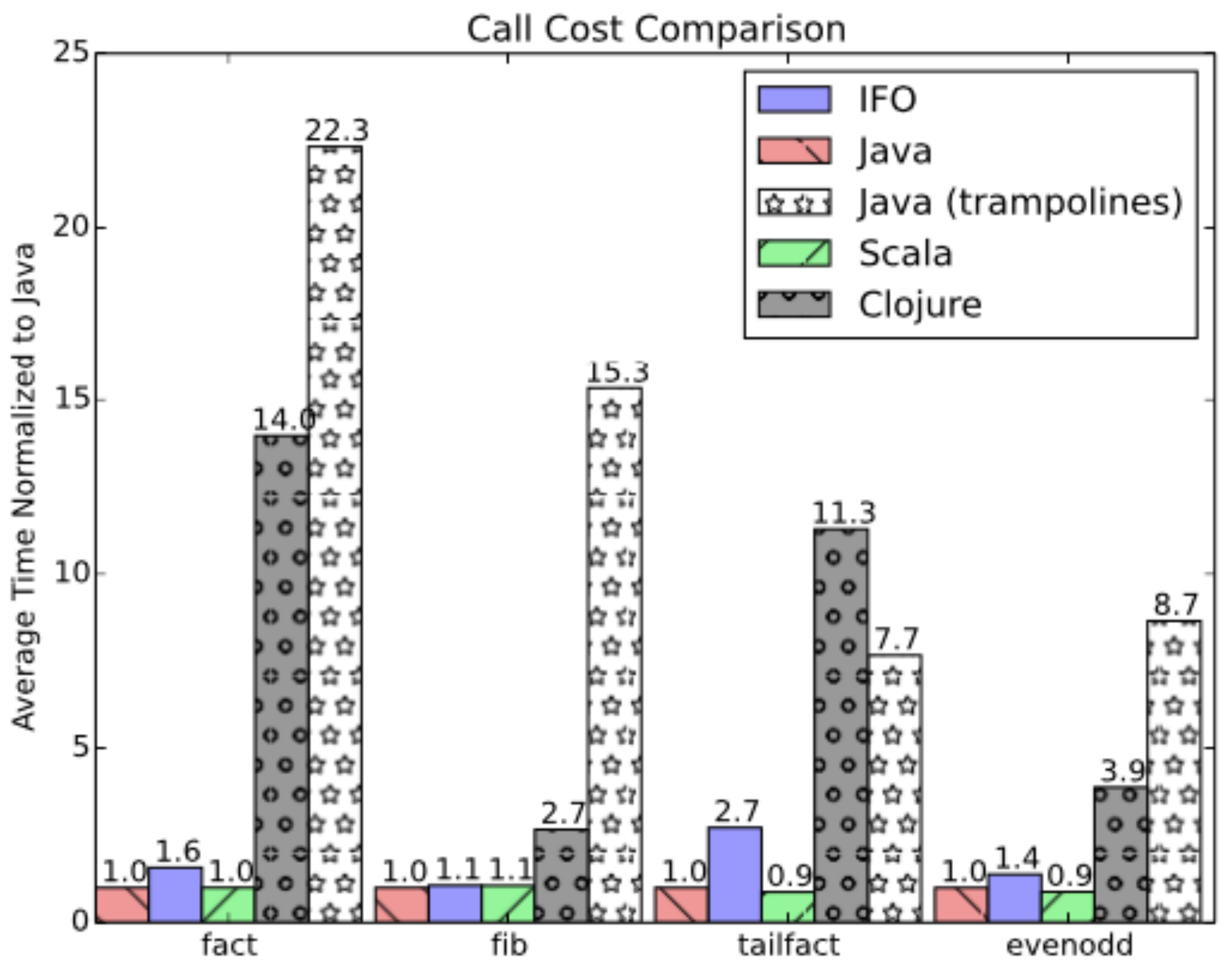


# EVALUATION

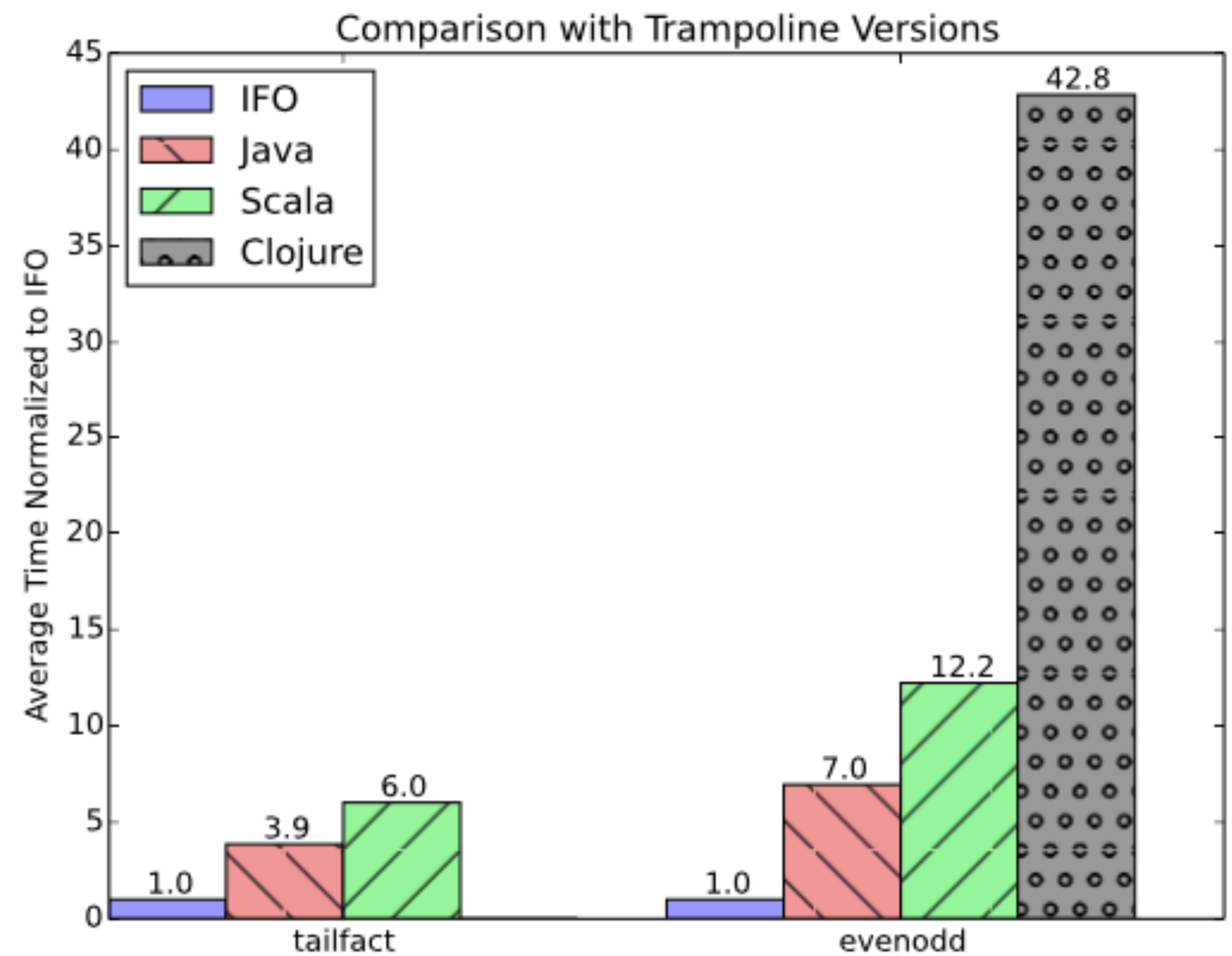
- **Uniform representation of functions + TCE**: does it have the claimed low overhead in time and memory?
- Two parts:
  1. **Micro-benchmarks**: general recursion, self tail-recursion, and mutual recursion
  2. **Applications**: encoding of Finite State Automata, and CPS-transformed (naive) 0-1 Knapsack Problem solution

# EVALUATION: MICROBENCHMARKS

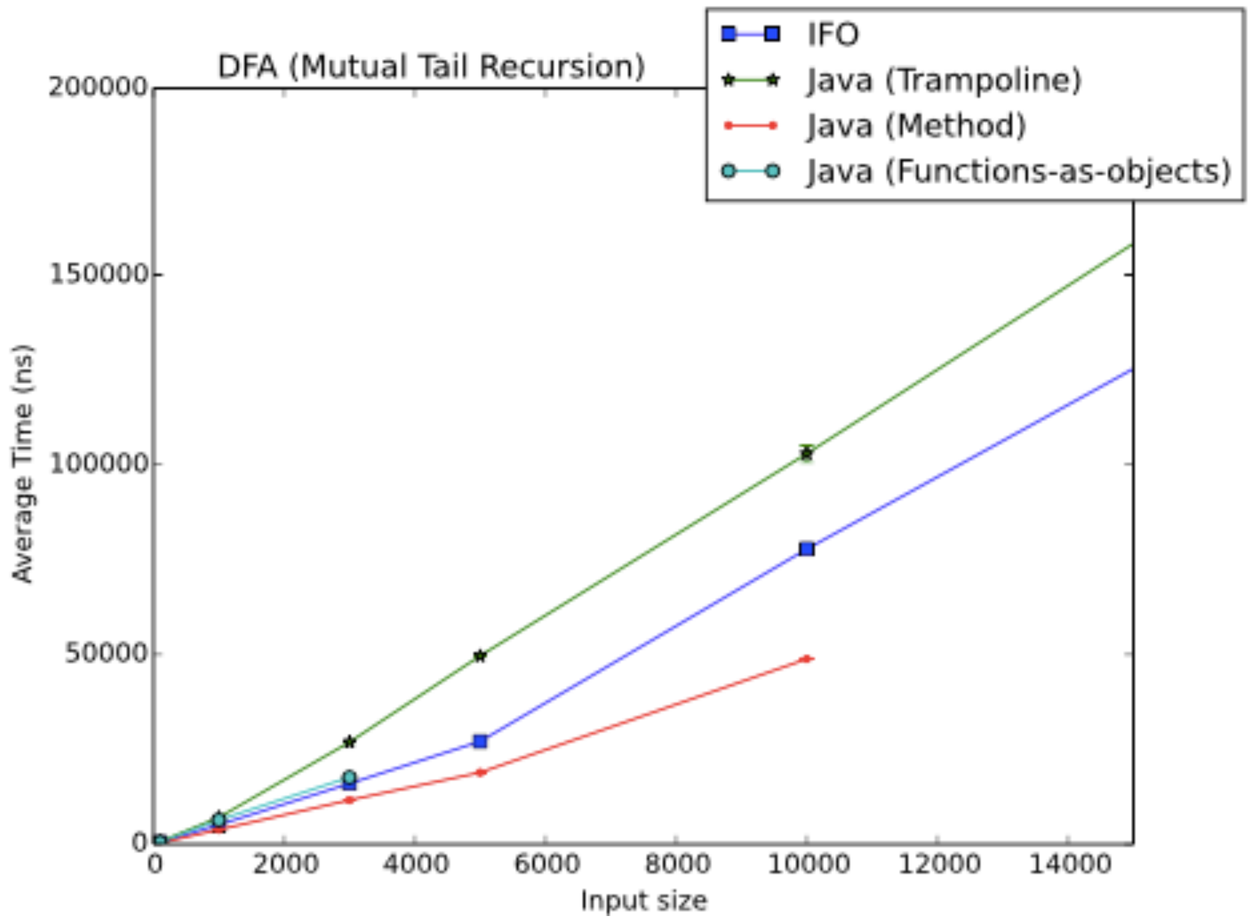
Low value (methods work)

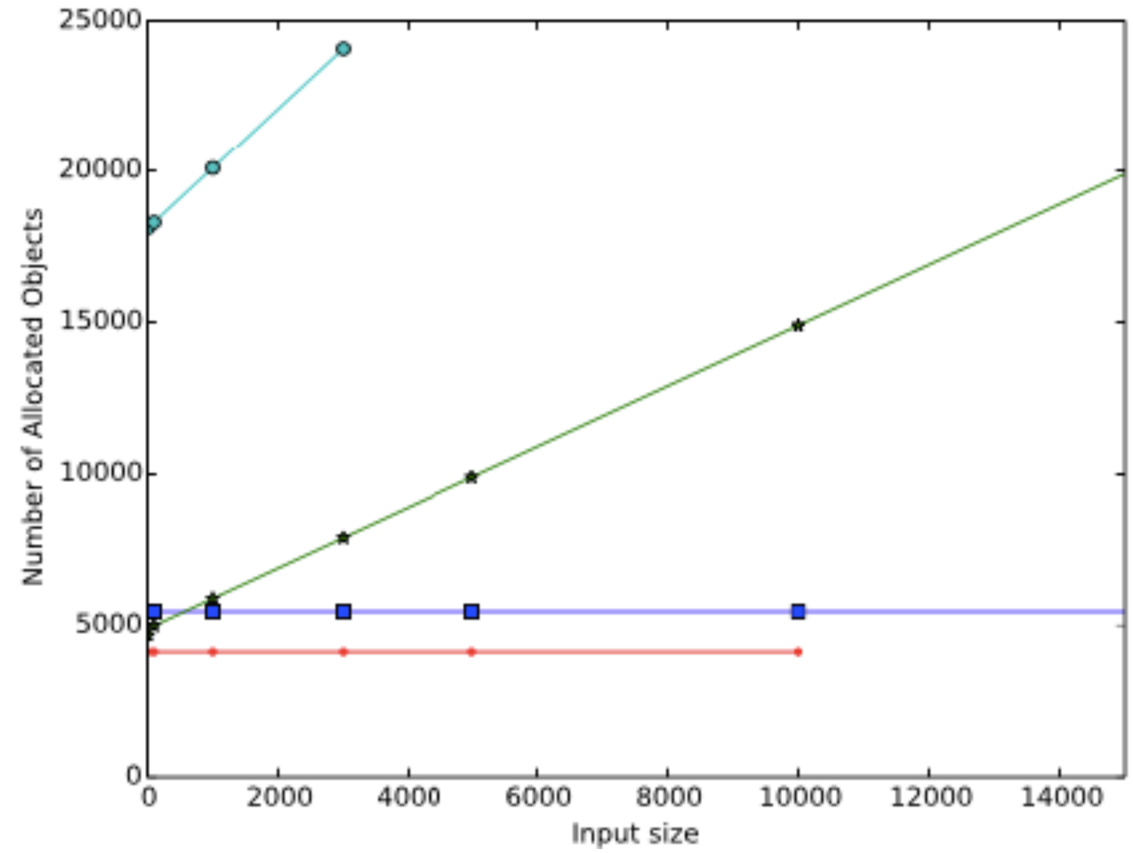



High value (trampolines needed)



# EVALUATION: DFA & CPS




CPS: method-based and FAO threw exception after input > 10

| ***Input length (time units)*** | 10 ($\mu s$) | 20 ($\mu s$) | 40 ($ms$) | 50 ($ms$) | 75 ($ms$) |
|---|---|---|---|---|---|
| **Java (Trampoline-based)** | $17.10 \pm 0.26$ | $320.42 \pm 5.88$ | $14.84 \pm 0.27$ | $62.35 \pm 1.06$ | $1044.61 \pm 12.99$ |
| **IFO** | $11.06 \pm 0.55$ | $289.19 \pm 6.52$ | $12.70 \pm 0.26$ | $48.47 \pm 1.08$ | $805.06 \pm 7.06$ |
| *Relative speedup* | 54.6% | 10.8% | 16.9% | 28.6% | 29.8% |

# RELATED WORK

- "*Transposing F to C#*" (A. Kennedy and D. Syme)
- Guy Steele's pioneering work on Scheme
- Various ML and Haskell-to-Java compilers (MLj, Jaskell, …)
- JVM modifications

# FUTURE WORK

- (**Now**) **to mid-2015**: additional work on the compiler backend (new optimizations, formalization, …)
- **Mid-2015 to 2016**: bootstrapping compiler
- **(~Now) to mid-2016**: mixin-based module system combined with formalized package management / build system (extending the current compiler)
- **Mid-2016 to 2017**: writing up thesis

# SUMMARY I

Uniform representation of function (no need for 2)

+

tail calls:

| | *correctness* | *space usage* | *time performance* | *supported in JVM* |
|---|---|---|---|---|
| proper tail calls | yes | none | fast | no |
| methods | no | stack | normal | yes |
| trampolines | yes | heap* | slow | yes |
| **our approach** | **yes** | **~none*** | **normal** | **yes** |

# SUMMARY II

- With IFOs, *programmers can* ***use idiomatic FP on JVM seamlessly*** *without compromises.*
- Two main contributions:
    - *Formalized compilation technique* that can be used and adapted by others (e.g. for Python VM)
    - *Its evaluated implementation* that can be used by functional language designers to target JVM

# BACKUP SLIDES

$0 := \lambda f.\lambda x.x$

$1 := \lambda f.\lambda x.f\ x$

$2 := \lambda f.\lambda x.f\ (f\ x)$

$3 := \lambda f.\lambda x.f\ (f\ (f\ x))$

$\mathrm{SUCC} := \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$

$\mathrm{PLUS} := \lambda m.\lambda n.m\ \mathrm{SUCC}\ n$

$\mathrm{MULT} := \lambda m.\lambda n.m\ (\mathrm{PLUS}\ n)\ 0$

$\mathrm{AND} := \lambda p.\lambda q.p\ q\ p$

$\mathrm{OR} := \lambda p.\lambda q.p\ p\ q$

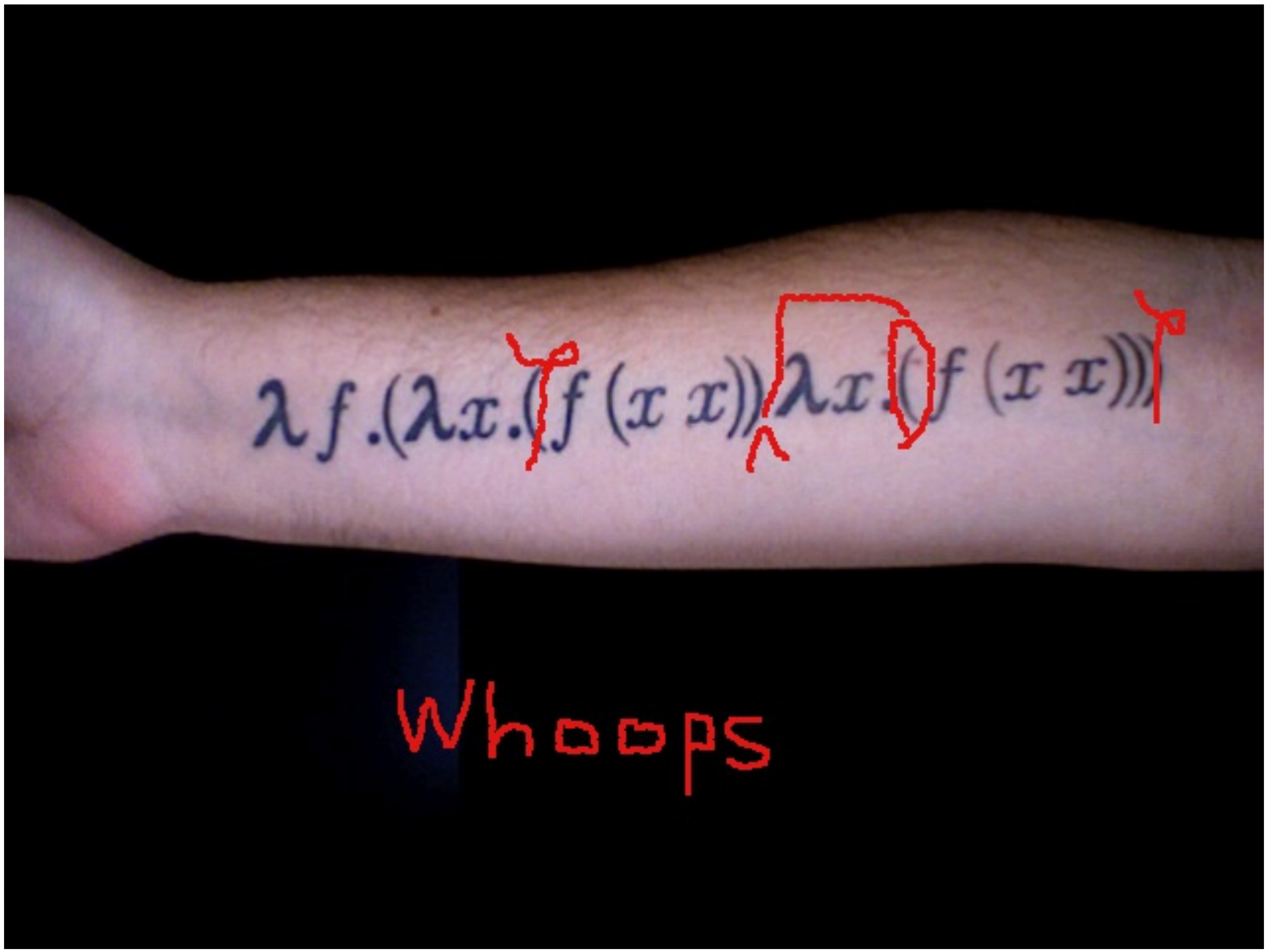
$\mathrm{NOT} := \lambda p.\lambda a.\lambda b.p\ b\ a$

$\mathrm{IFTHENELSE} := \lambda p.\lambda a.\lambda b.p\ a\ b$

$\mathrm{TRUE} := \lambda x.\lambda y.x$

$\mathrm{FALSE} := \lambda x.\lambda y.y$

# FIX-POINT COMBINATOR



# SYSTEM F -> CLOSURE F: DEFINITIONS

**Types** $\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau$

**Expressions** $e ::= x \mid \lambda(x:\tau).e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ \tau$

**Types** $T ::= \alpha \mid \forall\Delta.T$

**Expressions** $E ::= x \mid \lambda\Delta.E \mid E\ E \mid E\ T$

**Binders** $\Delta ::= \epsilon \mid \Delta(x:T) \mid \Delta\alpha$

# SYSTEM F -> CLOSURE F: EXAMPLES

$$const \equiv \Lambda A.(\lambda x : A).(\lambda y : A).x$$

$$\downarrow$$

$$const \equiv \lambda A\ (x : A)\ (y : A).x$$

$$const\ (\forall B.B \to B)\ (\Lambda B.(\lambda z : B).z)$$

$$\downarrow$$

$$const\ (\forall B\ (x : B).B)\ (\lambda B\ (z : B).z)$$

# SYSTEM F -> CLOSURE F: EXPRESSIONS

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![e_1 \;\; e_2]\!] &= [\![e_1]\!] \; [\![e_2]\!] \\
[\![e \; \tau]\!] &= [\![e]\!] \; |\tau| \\
[\![e]\!] &= [\![e]\!](\epsilon)
\end{aligned}
$$

$$
\begin{aligned}
[\![\lambda x : \tau_1 . e]\!](\Delta) &= [\![e]\!](\Delta \; (x : |\tau_1|)) \\
[\![\Lambda \alpha . e]\!](\Delta) &= [\![e]\!](\Delta \; \alpha) \\
[\![e]\!](\Delta) &= \lambda \Delta . [\![e]\!]
\end{aligned}
$$

# SYSTEM F -> CLOSURE F: TYPES

$$
\begin{aligned}
|\alpha| &= \alpha \\
|\tau| &= |\tau|(\epsilon) \\
\\
|\tau_1 \;\to\; \tau_2|(\Delta) &= |\tau_2|\;(\Delta\;(x : |\tau_1|))\;\textbf{where}\;x\;\textit{fresh} \\
|\forall\alpha.\tau|(\Delta) &= |\tau|\;(\Delta\;\alpha) \\
\\
|\tau|(\epsilon) &= \tau \\
|\tau|(\Delta) &= \forall\Delta.\tau
\end{aligned}
$$

# CLOSURE F -> JAVA: PRELIMINARY

- **IFO**:

```
abstract class Function {
  Object arg, res;
  abstract void apply();
}
```

- No formalization of Java

- Erasure semantics of System F / Closure F

- **Translation environments**:

$$\Gamma ::= \epsilon \mid \Gamma\,(x_1 : T \mapsto x_2) \mid \Gamma\alpha$$

# CLOSURE F -> JAVA: RULES 1A

$$\boxed{\Gamma \vdash E : T \leadsto J \text{ in } S}$$

(CJ-Var)
$$\frac{(x_1 : T \mapsto x_2) \in \Delta}{\Gamma \vdash x_1 : T \leadsto x_2 \text{ in } \{\}}$$

(CJ-Abs)
$$\frac{\Gamma; \Delta \vdash E : T \leadsto J \text{ in } S}{\Gamma \vdash \lambda\Delta.E : \forall\Delta.T \leadsto J \text{ in } S}$$

(CJ-TApp)
$$\frac{\Gamma \vdash E : \forall\alpha\Delta.T_2 \leadsto J \text{ in } S \qquad \Delta; T_2 \Downarrow T_3}{\Gamma \vdash E\,T_1 : T_3[T_1/\alpha] \leadsto J \text{ in } S}$$

# CLOSURE F -> JAVA: RULES 1B

$$\boxed{\Gamma \vdash E : T \leadsto J \textbf{ in } S}$$

(CJ-App)

$$\frac{\begin{array}{c}\Gamma \vdash E_1 : \forall (x : T_2)\Delta.T_1 \leadsto J_1 \textbf{ in } S_1 \\ \Gamma \vdash E_2 : T_2 \leadsto J_2 \textbf{ in } S_2 \qquad \Delta; T_1 \Downarrow T_3 \\ f,\ x_f\ fresh\end{array}}{\Gamma \vdash E_1\ E_2 : T_3 \leadsto x_f \textbf{ in } S_1 \uplus S_2 \uplus S_3}$$

$S_3$ := {
  Function f = $J_1$;
  f.arg = $J_2$;
  f.apply();
  $\langle T_3 \rangle$ $x_f$ = ($\langle T_3 \rangle$) f.res;}

# CLOSURE F -> JAVA: RULES IIA

$$\boxed{\Gamma; \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S}$$

(CJD-Empty) $$\frac{\Gamma \vdash E : T \rightsquigarrow J \textbf{ in } S}{\Gamma; \epsilon \vdash E : T \rightsquigarrow J \textbf{ in } S}$$

(CJD-Bind2) $$\frac{\Gamma\ \alpha; \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S}{\Gamma; \alpha\ \Delta \vdash E : T \rightsquigarrow J \textbf{ in } S}$$

# CLOSURE F -> JAVA: RULES 11B

$$\boxed{\Gamma; \Delta \vdash E : T \leadsto J \textbf{ in } S}$$

$$\texttt{(CJD-Bind1)} \quad \frac{\Gamma\,(y : T_1 \mapsto x_2); \Delta \vdash E : T \leadsto J \textbf{ in } S \qquad FC,\ x_1,\ x_2,\ f\ fresh}{\Gamma; (y : T_1)\ \Delta \vdash E : T \leadsto f \textbf{ in } S'}$$

```
S' := {
  class FC extends Function {
    Function x1 = this;
    void apply() {
      ⟨T1⟩ x2 = (⟨T1⟩) x1.arg;
      S;
      res = J;
    }
  };
  Function f = new FC();}
```

# CLOSURE F -> JAVA: RULES III

$$\boxed{\Delta; T_1 \Downarrow T_2}$$

$(\texttt{D-Empty})$ $$\overline{\epsilon; T \Downarrow T}$$

$(\texttt{D-NonEmpty})$ $$\overline{\Delta; T \Downarrow \forall \Delta . T}$$

Translation of Closure F types to Java types:

$$\begin{aligned} \langle \alpha \rangle &= \texttt{Object} \\ \langle \forall \Delta . T \rangle &= \texttt{Function} \end{aligned}$$

# DFA

$c(ad)^*r$ $\longrightarrow$

```
automaton init
  init : c -> more
  more : a -> more
         d -> more
         r -> end
  end  :
```

# CPS

```
let bestValue weightOf valueOf desiredWeight (items:_[]) =
   let weightOf i = weightOf items.[i-1]
   let valueOf i = valueOf items.[i-1]
   let rec bestValue = function
      | 0, _ | _, 0 -> 0.
      | i, w when weightOf i > w -> bestValue (i-1, w)
      | i, w ->
         let withoutItem = bestValue (i-1, w)
         let withItem = bestValue (i-1, w - weightOf i) + valueOf i
         max withoutItem withItem
   bestValue (items.Length, desiredWeight)
```

Method-based at 10: 9.68 ± 0.20 μs

FAO at 10: 13.79 ± 0.22 μs

we fixed the total weight to 10 and generated input values and weights lists of different sizes: we generated weights as consecutive sequences 1 to 5 and values as i × weight[i].

```
let bestValue' weightOf valueOf desiredWeight (items:_[]) =
   let weightOf i = weightOf items.[i-1]
   let valueOf i = valueOf items.[i-1]
   let rec bestValue k = function
      | 0, _ | _, 0 -> k 0.
      | i, w when weightOf i > w -> bestValue k (i-1, w)
      | i, w ->
         bestValue (fun withoutItem ->
            bestValue (fun withItem ->
                  let withItem = withItem + valueOf i
                  k(max withoutItem withItem)
               ) (i-1, w - weightOf i)
            ) (i-1, w)
   bestValue id (items.Length, desiredWeight)
```