Compiler Construction 2012/2013
Functional Programming Languages

Peter Thiemann

February 6, 2017

Functional Programming Languages

- Based on the mathematical notion of function
- Equational reasoning: $f(a) = f(a)$
- Pure/impure functional programming languages
- Characteristic feature:
  higher-order functions with nested lexical scope
  see also: delegates, anonymous classes, ...
- Well known functional programming languages
  - OCaml (F#, Standard ML),
  - Haskell,
  - Lisp (Scheme, Racket, Clojure)

Outline

Three Flavors of FP

FunJava

- MiniJava with higher-order functions
- Side effects permitted, cf. Scheme, ML
- Impure, higher-order functional language

PureFunJava

- FunJava without side effects
- Pure, higher-order functional language

LazyFunJava

- PureFunJava with lazy evaluation
- Nonstrict, pure functional, cf. Haskell

FunJava, the Language

MiniJava + function types

$$
\begin{array}{lcl}
\mathit{ClassDecl} & ::= & \texttt{type}\ \mathit{id} = \mathit{TypeExp}; \\
\mathit{TypeExp} & ::= & \mathit{TypeExp} \texttt{ -> } \mathit{TypeExp} \\
 & | & (\mathit{TypeList}) \texttt{ -> } \mathit{TypeExp} \\
 & | & (\mathit{TypeExp}) \\
 & | & \mathit{Type} \\
\mathit{TypeList} & ::= & \mathit{TypeExp}\ \mathit{TypeRest}^{*} \\
 & | & \varepsilon \\
\mathit{TypeRest} & ::= & ,\ \mathit{TypeExp}
\end{array}
$$

FunJava, the Language

MiniJava + function types

ClassDecl	$::=$	`type` *id* $=$ *TypeExp*;
TypeExp	$::=$	*TypeExp* `->` *TypeExp*
	$\mid$	(*TypeList*) `->` *TypeExp*
	$\mid$	(*TypeExp*)
	$\mid$	*Type*
TypeList	$::=$	*TypeExp* *TypeRest**
	$\mid$	ε
TypeRest	$::=$	, *TypeExp*

Examples

```
type constT = int -> int -> int
type arithT = (int, int) -> int
type runariT = arithT -> int -> int -> int
type thunkT = () -> int
```

FunJava, the Language

MiniJava + function calls

$$\begin{array}{rcl} \mathit{Exp} & ::= & \mathit{Exp}(\mathit{ExpList}) \\ & | & \mathit{Exp}.\mathit{id} \end{array}$$

- If `v` is an object with method `int m (int[])`, then `v.m` evaluates to a function of type `(int[]) -> int`.
- Evaluating `v.m` does **not** invoke the method.

FunJava, the Language

Expressions and Statements

$$
\begin{array}{lcl}
\mathit{MethodDecl} & ::= & \texttt{public}\ \mathit{Type}\ \mathit{id}(\mathit{FormalList})\ \mathit{Compound} \\
\mathit{Compound} & ::= & \{\mathit{VarDecl}^*\ \mathit{MethodDecl}^*\ \mathit{Statement}^* \\
 & & \qquad \texttt{return}\ \mathit{Exp};\} \\
\mathit{Exp} & ::= & \mathit{Compound} \\
 & | & \texttt{if}\ (\mathit{Exp})\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp}
\end{array}
$$

- Variables and functions/methods can be declared at the beginning of each block. (Nested functions)
- `return` produces the result for the next enclosing block. `{ return 3;} + { return 4;}` yields `7`.
- The `if` statement is replaced by an `if` expression.

FunJava Example Program

```
type intf = int -> int
class C {
  public intf add (n: int) {
    public int h (int m) { return m+n; }
    return h;
  }
  public intf twice (f: intf) {
    public int g (int x) { return f (f (x)); }
    return g;
  }
  public int test () {
    intf addFive = add (5);
    intf addSeven = add (7);
    int twenty = addFive (15);
    int twentyTwo = addSeven (15);
    intf addTen = twice (addFive);
    int seventeen = twice (add (5)) (7);
    intf addTwentyFour = twice (twice (add (6)));
    return addTwentyFour (seventeen);
  }
}
```

Outline

Representation of Function Values

Function pointer

Without nested functions (C)

- function pointers
  Function value = address of function's code
- In the IR:

```
MOVE (TEMP (t_ff), NAME (L_function))
CALL (TEMP (t_ff), ... parameters ...)
```

Representation of Function Values

- Function pointer insufficient for nested functions `h` and `g`:
  - where does `n` come from?
  - where does `f` come from?
- Solution: represent function value by a closure

Closure

- Object with one method and several instance variables
- Closure = code address + environment
- Environment = vector of values of free variables

Activation Records

- Function (`add`) may return a locally defined function (`h`)

⇒ This function `h` may refer to parameters and local variables of the enclosing function `add` (in particular, `n`)

⇒ Parameters and local variables cannot be allocated on the stack, but must be put in an activation record on the heap.

- Activation record holds a static link to the last activation record of the next enclosing function.

⇒ Need to create a closure object for `h` that contains the free variable `n`.

Closure for h

Translation to Java

```
interface Intf_Closure {
  public int apply(int n);
}
public Intf_Closure add (int n) {
  static class H implements Intf_Closure {
    int n;
    H(int n) {
      this.n = n;
    }
    int apply(int m) {
      return m+n;
    }
  }
  return new H(n);
}
```

Outline

Immutable Variables

- Equational reasoning not sound for FunJava

⇒ Consider a restricted language PureFunJava

⇒ PureFunJava prohibits side effects
  - No assignments to variables
    (exception: variable initialization)
  - No assignments to fields of records
    (exception: initialization in the constructor)
  - No calls to side-effecting external functions like `println`

- Programs in functional style produce new objects (partial copies) instead of changing existing ones.

Special Constructor Syntax

Syntax changes for PureFunJava

$$
\begin{array}{lcl}
\textit{ClassDecl} & ::= & \texttt{class}\ \textit{id} \\
 & & \quad \{\ \textit{VarDecl}^*\ \ \textit{MethodDecl}^*\ \ \textit{Constructor}\ \} \\
\textit{Constructor} & ::= & \texttt{public}\ \textit{id}\ (\textit{FormalList})\ \{\ \textit{Init}^*\ \} \\
\textit{Init} & ::= & \texttt{this.}\textit{id} = \textit{id}
\end{array}
$$

Continuation-Based I/O

- How to do I/O if side effects are disallowed?
- Answer: Enforce proper sequencing by using function calls
- I/O visible to type checker: `ans` type

Interface for functional I/O

```
type ans // special built-in type
type intConsumer = int -> ans
type cont = () -> ans

interface ContIO {
  public ans readByte (intConsumer c);
  public ans putByte (int i, cont c);
  public ans exit ();
}
```

Language Changes

- Remove `System.out.println`
- Add functional I/O types and operations
- Remove assignment and while loops
- Each block is limited to one statement following the declarations

PureFunJava, Example Program

```
public ans getInt (intConsumer done) {
  public ans nextDigit (int accum) {
    public ans eatChar (int dgt) {
      return if (isDigit (dgt))
                 nextDigit (accum*10+dgt-48)
               else done (accum);
    }
    return ContIO.readByte (eatChar);
  }
  return nextDigit (0);
}
```

Optimization of PureFunJava

- PureFunJava is a proper subset of FunJava
- All existing optimizations apply
- Computing the control flow graph is more demanding
- Additionally, optimization can exploit equational reasoning

Exploiting Equational Reasoning

Example Program

```
class G {
  int a; int b;
  public G (int a, int b) {
    this.a = a;
    this.b = b;
  }
}

int a1 = 5;
int b1 = 7;
G r = new G (a1, b1);

int x = f (r); // no change of r possible

int y = r.a + r.b; // must be equivalent to
int y = a1 + b1;
```

Outline

Inline Expansion

Definition: Inline Expansion (Inlining)

- Replace a function call by its definition
- Substitute actual parameter expressions for formal parameters

- Essential optimization for FP
  - many short functions
  - specializes higher-order functions
  - enabled by purity
- Further optimization enabled after inline expansion

Avoiding Variable Capture

Program with hole in scope

```
int x = 5
int g (int y) {
  return y+x;
}
int f (int x) {
  return g (1)+ x;
}
void main () { ... f(2)+x ... }
```

Avoiding Variable Capture

Program with hole in scope

```
int x = 5
int g (int y) {
  return y+x;
}
int f (int x) {
  return g (1)+ x;
}
void main () { ... f(2)+x ... }
```

Naive inlining of g into f (WRONG)

```
int f (int x) {
  return { return 1+x; } + x;
}
```

Avoiding Variable Capture

α-Conversion — Renaming of Bound Variables

First rename local variable

```
int g (int y) {
  return y+x;
}
int f (int a) {      // renamed x -> a
  return g (1)+ a;
}
```

Avoiding Variable Capture

α-Conversion — Renaming of Bound Variables

First rename local variable

```
int g (int y) {
  return y+x;
}
int f (int a) {      // renamed x -> a
  return g (1)+ a;
}
```

Then substitute g into f

```
int f (int a) {
  return { return 1+x; } + a;
}
```

Avoiding Variable Capture

α-Conversion — Renaming of Bound Variables

First rename local variable

```
int g (int y) {
  return y+x;
}
int f (int a) {      // renamed x -> a
  return g (1)+ a;
}
```

Then substitute g into f

```
int f (int a) {
  return { return 1+x; } + a;
}
```

Alternative

Rename all local variables so that each variable is bound at most once in the program.

Inline Expansion Algorithm

If actual parameters are variables . . .

Let $f(T_1\ a_1, \ldots, T_n\ a_n)B$ be in scope
Let $f(i_1, \ldots, i_n)$ be a call with i_j variables
Replace the call with
$B[a_1 \mapsto i_1, \ldots, a_n \mapsto i_n]$

Inline Expansion Algorithm

If actual parameters are variables . . .

Let $f(T_1\ a_1, \ldots, T_n\ a_n)B$ be in scope
Let $f(i_1, \ldots, i_n)$ be a call with i_j variables
Replace the call with
$B[a_1 \mapsto i_1, \ldots, a_n \mapsto i_n]$

If actual parameters are expressions . . .

Let $f(T_1\ a_1, \ldots, T_n\ a_n)B$ be in scope
Let $f(e_1, \ldots, e_n)$ be a call with e_j non-trivial expressions
Rewrite the call to
$\{T_1\ i_1 = e_1; \ldots T_n\ i_n = e_n;$ `return` $B[a_1 \mapsto i_1, \ldots, a_n \mapsto i_n]\}$
where i_j are fresh variables

Comments on Inline Expansion Algorithm

Why introduce fresh variables?

- Let `int double (j) { return j+j; }`
- Consider expanding the call `double (g (x))` ignoring that the actual argument is a non-trivial expression
- Result: `g (x) + g (x)`
  - Computation is repeated (expensive)
  - If impure, then side effect of `g(x)` is repeated and each call may yield a different result
  - (no problem if `g` is side effect-free)
- Introducing fresh variables avoids these problems:
  `{ i = g (x); return i+i; }`

Comments on Inline Expansion Algorithm

Why introduce fresh variables?

- Let `int double (j) { return j+j; }`
- Consider expanding the call `double (g (x))` ignoring that the actual argument is a non-trivial expression
- Result: `g (x) + g (x)`
  - Computation is repeated (expensive)
  - If impure, then side effect of `g(x)` is repeated and each call may yield a different result
  - (no problem if `g` is side effect-free)
- Introducing fresh variables avoids these problems:

  `{ i = g (x); return i+i; }`
- Remarks
  - Order of aux. definitions must match evaluation order
  - An implementation would handle each argument separately
  - Dead function elimination possible after inlining

Inlining Recursive Functions

Some Example Code

```
class list {int head; int tail;} // constructor omitted
type observeInt = (int, cont) -> ans

public ans doList (observeInt f, list l, cont c) {
  return
    if (l===null)
      c ();
    else {
      public ans doRest () {
        return doList (f, l.tail, c);
      }
      return f (l.head, doRest);
    };
}
public ans printTable (list l, cont c) {
  return doList (printDouble, l, c);
}
```

Inlining Recursive Functions

Inlining `doList` into `printTable` does not yield the desired result:

```
public ans printTableDL (list l, cont c) {
  return
    if (l===null)
      c ();
    else {
      public ans doRest () {
        return doList (printDouble, l.tail, c);
      }
      return printDouble (l.head, doRest);
    };
}
```

Inlining Recursive Functions

Inlining `doList` into `printTable` does not yield the desired result:

```
public ans printTableDL (list l, cont c) {
  return
    if (l===null)
      c ();
    else {
      public ans doRest () {
        return doList (printDouble, l.tail, c);
      }
      return printDouble (l.head, doRest);
    };
}
```

- Only the first element is processed directly with `printDouble`, the remaining are still processed with the generic `doList`

Inlining Recursive Functions

Loop-Preheader Transformation

Given recursive function $T\ f(a_1, \ldots, a_n)B$
Transform to

```
$T$   $f(a'_1, \ldots, a'_n)\{$
        $T\ f'(a_1, \ldots, a_n)B[f \mapsto f']$
        return $f'(a'_1, \ldots, a'_n)$;
$\}$
```

Inlining Recursive Functions

Loop-Preheader Transformation

Given recursive function $T\ f(a_1, \ldots, a_n)B$
Transform to

$$\begin{array}{ll} T & f(a'_1, \ldots, a'_n)\{ \\ & T\ f'(a_1, \ldots, a_n)B[f \mapsto f'] \\ & \texttt{return}\ f'(a'_1, \ldots, a'_n); \\ \} & \end{array}$$

- Inlining copies specialized local function f' into the target

Inlining Recursive Functions
Loop-Preheader Transformation Applied

```
public ans doList (observeInt fX, list lX, cont cX) {
  public ans doListX (observeInt f, list l, cont c) {
    return
      if (l===null)
        c ();
      else {
        public ans doRest () {
          return doListX (f, l.tail, c);
        }
        return f (l.head, doRest);
      };
  }
  return doListX (fX, lX, cX);
}
```

Inlining Recursive Functions

Loop-Preheader Transformation Applied

```
public ans doList (observeInt fX, list lX, cont cX) {
  public ans doListX (observeInt f, list l, cont c) {
    return
      if (l===null)
        c ();
      else {
        public ans doRest () {
          return doListX (f, l.tail, c);
        }
        return f (l.head, doRest);
      };
  }
  return doListX (fX, lX, cX);
}
```

- Observation: arguments `f` and `c` are loop invariant
- Replace by outer parameters

Inlining Recursive Functions
Hoisting Loop-Invariant Arguments

```
public ans doList (observeInt f, list lX, cont c) {
  public ans doListX (list l) {
    return
      if (l===null)
        c ();
      else {
        public ans doRest () {
          return doListX (l.tail);
        }
        return f (l.head, doRest);
      };
  }
  return doListX (lX);
}
```

Inlining Recursive Functions

Inlining of `doList` into `printTable` continued

```
public ans printTable (list lX, cont c) {
  public ans doListX (list l) {
    return
      if (l===null)
        c ();
      else {
        public ans doRest () {
          return doListX (l.tail);
        }
        return printDouble (l.head, doRest);
      };
  }
  return doListX (lX);
}
```

- `printDouble` is called directly and can be inlined!

Inlining Recursive Functions

Cascaded Inlining

```
public ans printTable (list lX, cont c) {
 public ans doListX (list l) {
  return
   if (l===null)
    c ();
   else {
    public ans doRest () {
     return doListX (l.tail);
    }
    return {
     int i = l.head;
     public ans again() {return putInt (i+i, doRest);}
     return putInt (i, again);
    };
   };
 }
 return doListX (lX);
}
```

Avoiding Code Explosion

- Inline expansion copies function bodies

$\Rightarrow$ The program text becomes bigger

$\Rightarrow$ Expansion may not terminate

- Controlling inlining
  1. Expand very frequently executed call sites
     determine frequency by static estimation or execution profiling
  2. Expand functions with very small bodies
  3. Expand functions called only once
     rely on dead function elimination

Outline

Closure Conversion

- Closure = code address + environment
- One representation of closures: objects
- Closure conversion transforms the program so that no function appears to access free variables
- Approach: represent a function value of type `t1 -> t2` by an object implementing the interface

  ```
  interface I_t1_t2 {
    public t2 exec (t1 x);
  }
  ```
- There is a separate implementation class for each function, as the free variables differ

Closure Conversion

Example

```
class doRest implements I_list_answer {
  doListX dlx; list l;
  public ans exec () { return dlx.exec (l.tail); }
}
class again implements I_void_answer {
  doListX dlx; list l; int i;
  public ans exec () {return putInt (i+i, new doRest (dlx, l));}
}
class doListX implements I_list_answer {
  cont c;
  public ans exec (list l) {
    return
      if (l===null) c.exec ();
      else {
        return { int i = l.head;
                 return putInt (i, new again (this, l, i)); };
      };
  }
class printTable implements I_list_cont_answer {
  public exec (list lX, cont c) {
    return new doListX (c).exec (lX);
  }
}
```

Outline

Tail Recursion

- Functional programs have no loops (e.g., no while, for, repeat)
- Efficient (iterative) recursion through tail recursion
- A function is tail recursive if each recursive function call is a tail call
- Tail calls defined by contexts:

$$\begin{array}{lcl} B & = & \{t_1\ x_1 = e_1; \ldots t_n\ x_n = e_n;\ \texttt{return}\ B'\} \\ B' & = & \square \mid B \mid \texttt{if}(e)\ B'\ \texttt{else}\ B' \end{array}$$

- A call to g is a tail call if it occurs in a function definition as follows
  $t\ f(a_1, \ldots, a_n)B[g(e_1, \ldots, e_m)]$

Implementation of Tail Calls

Example

```
int g (int y) { int x = h(y); return f(x); }
```

- `h(y)` is not a tail call
- `f(x)` is a tail call
- Tail calls can be implemented more efficiently by a jump instead of a call
- Calling sequence for tail call:
  1. Move actual parameters into argument registers
  2. Restore callee-save registers
  3. Pop stack frame of the calling function (if it has one)
  4. Jump to the callee

Effects of Tail Calls

- In `printTable`, all calls are tail calls

$\Rightarrow$ Can all be implemented with jumps

- The generated code is very similar to the code generated for the equivalent imperative program (with a while loop)
- Difference: activation block on the heap vs. on the stack
- Amendment
  - By compile-time escape analysis: objects that do not escape can be stack-allocated
  - By extremely cheap heap allocation and garbage collection

Outline

Lazy Evaluation

- β-reduction: important law in equational reasoning
- Reminder β-reduction: if $f(x) = B$, then $f(e) = B[x \mapsto e]$
- PureFunJava violates this law

Unsound β-Reduction in PureFunJava

```
{
  int loop (int z) {
    return
      if (z>0) 42
      else loop (z));
  }
  int f (int x) {
    return if (y>8) x
           else -y;
  }
  return f (loop (y));
}
```

```
{
  int loop (int z) {
    return
      if (z>0) 42
      else loop (z));
  }
  int f (int x) {
    return if (y>8) x
           else -y;
  }
  return if (y>8) loop (y)
           else -y;
}
```

- For $y = 0$, code on left loops, but code on right terminates

Remedy: LazyJava With Call-By-Name Evaluation

- LazyJava
  - same syntax as PureFunJava
  - but with lazy evaluation:
    expressions are only evaluated if and when their value is demanded by execution of the program
- First step: call-by-name evaluation
  - Transform each expression to a thunk
  - Thunk: parameterless procedure that yields the value of the expression when invoked
  - Advantage: evaluation only when needed
  - Disadvantage: evaluation can be repeated arbitrarily often

Introducing Thunks

Original Program (lookup in binary tree)

```
class Tree {
  String key;
  int binding;
  Tree left;
  Tree right;
}
public int look (Tree t, String k) {
  int c = t.key.compareTo(k);
  if (c < 0) return look (t.left, k);
  else if (c > 0) return look (t.right, k);
  else return t.binding;
}
```

Introducing Thunks

Transformed Program (lookup in binary tree)

```
type th_int = () -> int;
type th_tree = () -> Tree;
type th_string = () -> String;

class Tree {
  th_String key;
  th_int binding;
  th_tree left;
  th_tree right;
}
public th_int look (th_tree t, th_String k) {
  th_int c = t ().key ().compareTo(k);
  if (c () < 0) return look (t ().left, k);
  else if (c () > 0) return look (t ().right, k);
  else return t ().binding;
}
```

Call-By-Need Evaluation

- Second step: call-by-need evaluation

= Call-by-name evaluation with caching of result

- First invocation of a thunk stores result in memo slot of the thunk's closure
- Further invocations return the value from the memo slot
- (exploits / requires purity)

Call-By-Need Transformation

Example

Recall

```
int y;
f (loop (y))
```

is transformed to

```
th_int y;
f.exec (new intThunk () {
  public int eval () {
    return loop.exec (y);
  };
})
```

With supportive definitions (requiring assignment)

```
abstract class intThunk {
  int memo; boolean done = false;
  abstract public int eval();
  public int exec () {
    if (!done) {
      memo = this.eval();
      done = true;
    }
    return memo;
  }
}
```

Example Evaluation of a Lazy Program

```
{
  int fact (int i) {
    return if (i==0) 1 else i * fact (i-1);
  }
  Tree t0 = new Tree ("",0,null,null);
  Tree t1 = t0.enter ("-one", fact (-1));
  Tree t2 = t1.enter ("three", fact (3));
  return putInt (t2.look ("three", exit));
}
```

- Fortunately, `fact (-1)` is never evaluated!

Optimization

- All the standard optimizations apply
- Additional optimization opportunities due to equational reasoning
  - Invariant hoisting
  - Dead-code removal
  - Deforestation

Invariant Hoisting

```
type intfun = int -> int

intfun f (int i) {
  public int g (int j) {
    return h (i) * j;
  }
  return g;
}
```

```
type intfun = int -> int

intfun f (int i) {
  int hi = h (i);
  public int g (int j) {
    return hi * j;
  }
  return g;
}
```

- In lazy functional language, left can be transformed into right
- Incorrect in strict language: `h(i)` may not terminate or yield different results on each call

Dead-Code Removal

```
int f (int i) {
  int d = g (x);
  return i+2;
}
```

- d is dead after its definition
- The LFL compiler removes this definition
- Incorrect in strict language!

Common modularization in FP

```
class intList {int head, intList tail;}
type intfun = int -> int;
type int2fun = (int,int) -> int;

public int sumSq (intfun inc, int2fun mul, int2fun add) {
  public intList range (int i, int j) {
    return if (i>j) then null
           else new intList (i, range (inc (i), j));
  }
  public intList squares (intList l) {
    return if (l==null) null
           else new intList (mul (l.head, l.head), squares (l.tail));
  }
  public int sum (int accum, intList l) {
    return if (l==null) accum
           else sum (add (accum, l.head), l.tail);
  }
  return sum (0, squares (range (1,100)));
}
```

Result of Deforestation

```
public int sumSq (intfun inc, int2fun mul, int2fun add)
  public int f (int accum, int i, int j) {
    return if (i>j) accum
           else f (add (accum, mul (i,i)), inc (i));
  }
  return f (0,1,100);
}
```

- Deforestation removes intermediate data structures
- Rearranges the order of function calls
- Only legal in a pure FL

Strictness Analysis

- A function is strict in an argument, if this argument is always needed to produce the result of the function.
- Put formally:
  A function $f(x_1, \ldots, x_n)$ is strict in x_i if whenever the expression a fails to terminate, then the function call $f(b_1, \ldots, b_{i-1}, a, b_{i+1}, \ldots, b_n)$ fails to termiante.
- If the compiler knows that a function is strict, then it need not allocate a thunk for the argument, but it can evaluate it right away.
- Program analysis can approximate strictness

Examples: Strictness

```
int f (int x, int y) { return x + x + y; }

int g (int x, int y) { return if (x>0) y else x; }

Tree h (String x, int y) {
  return new Tree (x, y, null, null);
}

int j (int x) { return j(0); }
```

- `f` strict in `x` and `y`
- `g` strict in `x` not in `y`
- `h` not strict
- `j` strict in `x`

Using Strictness Information

- Lookup in a tree is strict in the tree and in the key
- But the binding information as well as the fields in the tree are not strict

```
th_String look (Tree t, key k) {
  return if (k < t.key.eval())
           look (t.left.eval (), k)
         else if (k > t.key.eval())
           look (t.right.eval (), k)
         else
           t.binding;
}
```

Strictness Analysis

- Exact strictness information is not computable
- Conservative approximation needed
- Domain: $b \in \{0, 1\}$
  - 1 (true) evaluation may terminate
  - 0 (false) evaluation does not terminate (definitely)
- Result is set H containing pairs $(f, \vec{b})$
- f strict in x_i if $(f, (1, \ldots, 1, 0, 1, \ldots, 1)) \notin H$

Strictness Analysis

For First-Order Functions

$$
\begin{array}{lcl}
M(c,\sigma) & = & 1 \\
M(x,\sigma) & = & x \in \sigma \\
M(E_1 + E_2,\sigma) & = & M(E_1,\sigma) \wedge M(E_2,\sigma) \\
M(\texttt{new}(E_1,\ldots),\sigma) & = & 1 \\
M(\texttt{if}\ E_1\ E_2\ E_3,\sigma) & = & M(E_1,\sigma) \wedge (M(E_2,\sigma) \vee M(E_3,\sigma)) \\
M(f(E_1,\ldots),\sigma) & = & (f,(M(E_1,\sigma),\ldots)) \in H
\end{array}
$$

Strictness Analysis

Fixpoint Iteration

```
$H \leftarrow \{\}$
**repeat**
    $done \leftarrow$ true
    **for** each function $f(x_1, \ldots, x_n) = B$ **do**
        **for** each sequence $(b_1, \ldots, b_n) \in \{0, 1\}^n$ **do**
            **if** $(f, (b_1, \ldots, b_n)) \notin H$ **then**
                $\sigma \leftarrow \{x_i \mid b_i = 1\}$
                **if** $M(B, \sigma)$ **then**
                    $done \leftarrow$ false
                    $H \leftarrow H \cup \{(f, (b_1, \ldots, b_n))\}$
                **end if**
            **end if**
        **end for**
    **end for**
**until** done
```

Strictness Analysis

Assessment

- Basic analysis, quite expensive
- Not applicable to full LazyJava
- Does not handle data structures
- Does not handle higher order functions
- Better algorithms exist that handle both
- Used in compilers for, e.g., Haskell

Outline

JSR 335: Higher-Order Functions for Java

This JSR will extend the Java Programming Language Specification and the Java Virtual Machine Specification to support the following features:

- Lambda Expressions (anonymous functions)
- SAM Conversion
- Method References
- Virtual Extension Methods

Scheduled for Java SE 8

Closures

Java already has “closures” in the guise of anonymous inner classes.

Definition

```java
public interface CallbackHandler {
    public void callback(Context c);
}
```

Use

```java
foo.doSomething(new CallbackHandler() {
                    public void callback(Context c) {
                        System.out.println("pippo");
                    }
                });
```

Drawbacks of Anonymous Inner Classes

1. Bulky syntax
2. Inability to capture non-final local variables
3. Transparency issues surrounding the meaning of return, break, continue, and 'this'
4. No nonlocal control flow operators

The proposal mainly addresses items 1, 2, and 3.

Adding Lambda Expressions

- Replacing the machinery of anonymous inner classes
- Without introducing function types
- Instead: SAM conversion
- SAM = Single Abstract Method
    - Many common interfaces and abstract classes have this property, such as `Runnable`, `Callable`, `EventHandler`, or `Comparator`.
    - These are SAM types.
    - SAM-ness is a structural property identified by the compiler
- Introduce syntax to simplify the creation of SAM instances

Syntax of Lambda Expressions

- `#{ -> 42 }` or even `#{ 42 }`
  no arguments, returns 42
- `#{ int x -> x + 1 }`
  an **`int`** argument, returns `x+1`
- In general,
  - body can be an expression or
  - a statement list like a method body.

SAM Conversion

- A lambda expression is only legal in a context, where a SAM type is expected.
- The compiler infers the argument, return, and exception types.
- It checks them for assignment compatibility with the type of the method of the expected SAM type.
- The name of the method is ignored.
- Example:

```
CallbackHandler cb =
  #{ Context c -> System.out.println("pippo") };
```

- Illegal:

```
Object o = #{ 42 };
```

Method References

- Transforming a method reference to a function
- Example

```
class Person {
  private final String name;
  private final int age;

  public static int compareByAge(Person a, Person b)
    { ... }
  public static int compareByName(Person a, Person b)
    { ... }
}

Person[] people = ...
Arrays.sort(people, #Person.compareByAge);
```

Extension Methods

- Existing interfaces cannot be extended without breaking implementations.
- Closures give new opportunities for useful API additions, e.g., in the collection classes.
- Extension methods propose a way out of this dilemma.
- The proposal permits to extend an interface safely by providing a default implementation.
- Example:

```
public interface Set<T> extends Collection<T> {
  public int size();
  // The rest of the existing Set methods
  public extension T reduce(Reducer<T> r)
    default Collections.<T>setReducer;
}
```