Compiler Construction 2009/2010
Functional Programming Languages

Peter Thiemann

December 22, 2009

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, ...

Outline

Three Flavors of FP

FunJava

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

PureFunJava

- FunJava w/o side effects
- Pure, HO functional language

LazyFunJava

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

FunJava, the Language

MiniJava + function types

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

FunJava, the Language

MiniJava + function calls

$$
\begin{aligned}
Exp &= Exp(ExpList) \\
Exp &= Exp.id
\end{aligned}
$$

- 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

Closures

Representation of Function Values

- 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 ...)
```

- Not sufficient for nested functions like `h` and `g`:
  - where does `n` come from?
  - where does `f` come from?
- Solution: represent function value by a <u>closure</u>
- Closure = record of code address and values of free variables (environment)
- Similar to object with one method and several instance variables

Activation Records

- Function may return a locally defined function

⇒ This function may refer to parameters and local variables

⇒ Parameters and local variables cannot be allocated on the stack, but must be put on the heap

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

Outline

Immutable Variables

- Equational reasoning not sound for FunJava

⇒ 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 object (partial copies) instead of changing existing ones

Special Constructor Syntax

Syntax Changes for PureFunJava

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

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: `answer` type

Interface for functional I/O

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

class ContIO {
  public answer readByte (intConsumer c);
  public answer putByte (int i, cont c);
  public answer 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 statment following the declarations

PureFunJava, Example Program

```
public answer getInt (intConsumer done) {
  public answer nextDigit (int accum) {
    public answer eatChar (int dig) {
      return if (isDigit (dig))
                nextDigit (accum*10+dig-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

Inline Expansion

- Replace a function call by its definition
- Substituting actuals for formals

- Essential optimization for FP
  - many short functions
  - specializes higher-order functions
- Further optimization possible after inlining

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

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

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

Actual parameters are variables

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

Inline Expansion Algorithm

Actual parameters are variables

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

Actual parameters are expressions

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

Comments on Inline Expansion Algorithm

- 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
- Correct inlining avoids these problems:

```
{ i = g (x); return i+i; }
```

Comments on Inline Expansion Algorithm

- 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
- Correct inlining avoids these problems:

  `{ i = g (x); return i+i; }`
- Remarks
  - 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) -> answer

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

Inlining Recursive Functions

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

```
public answer printTableDL (list l, cont c) {
  return
    if (l===null)
      c ();
    else {
      public answer 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 answer printTableDL (list l, cont c) {
  return
    if (l===null)
      c ();
    else {
      public answer 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 int $f(a_1, \ldots, a_n)B$
Transform to

$$\begin{array}{ll}
\texttt{int} & f(a'_1, \ldots, a'_n)\{ \\
 & \texttt{int}\ f'(a_1, \ldots, a_n)B[f \mapsto f'] \\
 & \texttt{return}\ f'(a'_1, \ldots, a'_n); \\
\} &
\end{array}$$

Inlining Recursive Functions

Loop-Preheader Transformation

Given `int` $f(a_1, \ldots, a_n)B$
Transform to

$$
\begin{array}{ll}
\texttt{int} & f(a'_1, \ldots, a'_n)\{ \\
 & \texttt{int}\ f'(a_1, \ldots, a_n)B[f \mapsto f'] \\
 & \texttt{return}\ f'(a'_1, \ldots, a'_n); \\
\} &
\end{array}
$$

- Inlining now copies the specialized local function f' into the target

Inlining Recursive Functions

Loop-Preheader Transformation Applied

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

- Observation: arguments `f` and `c` are <u>loop invariants</u>
- Replace by outer parameters

Inlining Recursive Functions
Hoisting Loop-Invariant Arguments

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

Inlining Recursive Functions

Inlining of `doList` into `printTable` continued

```
public answer printTable (list lX, cont c) {
  public answer doListX (list l) {
    return
      if (l===null)
        c ();
      else {
        public answer 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 answer printTable (list lX, cont c) {
  public answer doListX (list l) {
    return
      if (l===null)
        c ();
      else {
        public answer doRest () {
          return doListX (l.tail);
        }
        return {
          int i = l.head;
          public answer again() {return putInt (i+i, doR
          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 different implementation class for each function, as the free variables differ

Closure Conversion

Example

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

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

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

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

Remedy: LazyJava & 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 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 twenty = addFive (15);
```

is transformed to

```
th_int twenty = new intThunk (this);  // this |-> addFive
```

With supportive definitions (requiring assignment)

```
class intThunk {public int eval(); int memo; boolean done;}

class c_int_int {public int exec (int x);}
class intFuncThunk {public c_int_int eval();
                    c_int_int memo; boolean done;}

class twentyThunk extends intThunk {
  intFuncThunk addFive;
  public int exec () {
    if (!done) {
      memo = addFive.eval().exec (15);
      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 $(f, (1, \ldots, 1, 0, 1, \ldots, 1)) \notin H$

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

```
H ← {}
repeat
    done ← true
    for each function f(x_1, ..., x_n) = B do
        for each sequence (b_1, ..., b_n) ∈ {0, 1}^n do
            if (f, (b_1, ..., b_n)) ∉ H then
                σ ← {x_i | b_i = 1}
                if M(B, σ) then
                    done ← false
                    H ← H ∪ {(f, (b_1, ..., 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