Closures in Java

Gregor Bethlen

Universität Karlsruhe

27.05.2010

Outline

1. Introduction
   - Motivation
   - Examples
   - Conclusions
2. Proposals
   - BGGA
   - CICE
   - FCM
3. Summary

Motivation

- treat functions/methods as values for variables and parameters of methods
- ease the passing of callback-functions
- treat some situations more elegantly (for example one object acting as an observer twice)
- ease for example the creation of threads
- obsolete several (API-)interfaces which only exist due to the present absence of closures in Java
- parametrise algorithms with functions

Motivation

- treat functions/methods as values for variables and parameters of methods
- ease the passing of callback-functions
- treat some situations more elegantly (for example one object acting as an observer twice)
- ease for example the creation of threads
- obsolete several (API-)interfaces which only exist due to the present absence of closures in Java
- parametrise algorithms with functions

Motivation

- treat functions/methods as values for variables and parameters of methods
- ease the passing of callback-functions
- treat some situations more elegantly (for example one object acting as an observer twice)
- ease for example the creation of threads
- obsolete several (API-)interfaces which only exist due to the present absence of closures in Java
- parametrise algorithms with functions

Motivation

- treat functions/methods as values for variables and parameters of methods
- ease the passing of callback-functions
- treat some situations more elegantly (for example one object acting as an observer twice)
- ease for example the creation of threads
- obsolete several (API-)interfaces which only exist due to the present absence of closures in Java
- parametrise algorithms with functions

Motivation

- treat functions/methods as values for variables and parameters of methods
- ease the passing of callback-functions
- treat some situations more elegantly (for example one object acting as an observer twice)
- ease for example the creation of threads
- obsolete several (API-)interfaces which only exist due to the present absence of closures in Java
- parametrise algorithms with functions

Motivation

- treat functions/methods as values for variables and parameters of methods
- ease the passing of callback-functions
- treat some situations more elegantly (for example one object acting as an observer twice)
- ease for example the creation of threads
- obsolete several (API-)interfaces which only exist due to the present absence of closures in Java
- parametrise algorithms with functions

Syntax

- our examples use a syntax like the one proposed in First-class methods: Java-style closures
- we have function-types like `#(double(int, int))` for functions taking two integers and return a double
- we have (anonymous) inner methods like

```
#(int a, int b) {
  return (double)(a + b) / 2;
}
```

Syntax

- our examples use a syntax like the one proposed in First-class methods: Java-style closures
- we have function-types like `#(double(int, int))` for functions taking two integers and return a double
- we have (anonymous) inner methods like

```
#(int a, int b) {
  return (double)(a + b) / 2;
}
```

Syntax

- our examples use a syntax like the one proposed in First-class methods: Java-style closures
- we have function-types like `#(double(int, int))` for functions taking two integers and return a double
- we have (anonymous) inner methods like

```
#(int a, int b) {
  return (double)(a + b) / 2;
}
```

Examples

We can assign and invoke

```
#(double(int, int)) avg =
  #(int a, int b) {
    return (double)(a + b) / 2;
  };

double result = avg.invoke(3, 10);
```

with the result 6.5.

Examples

An example without a local variable in the definition-context.

```
public #(int(int)) getAddTwo() {
  #(int(int)) addTwo = #(int a) { return a + 2; };
  return addTwo;
}

public void f() {
  #(int(int)) closure = getAddTwo();
  int result = closure.invoke(33);   //result is 35
}
```

Examples

An example with a local variable in the definition-context.

```
public #(int(int)) getAddX(int x) {
  int summand = x;
  #(int(int)) addX = #(int a) { return a + summand; };
  return addX;
}

public void f() {
  #(int(int)) closure = getAddX(4);
  int result = closure.invoke(33);   //result is 37
}
```

Examples

An example with a local variable in the definition-context which value is changed after the closure-definition.

```
public #(int(int)) getAddX(int x) {
  int summand = x;
  #(int(int)) addX = #(int a) { return a + summand; };
  summand = 17;
  return addX;
}

public void f() {
  #(int(int)) closure = getAddX(4);
  int result = closure.invoke(33);   //result is 50
}
```

Impacts for local variables

We conclude that local variables of a definition-context, which are used in a closure, can not be put on the stack; they must go on the heap.

The same holds for references/pointers to objects.

Impacts for verification

It is not possible to treat a local variable `var` used in method `meth` of object `obj` as a synthetic private attribute `obj.meth_var`, as one may think.

Let there be a closure defined in `meth` using `var`. If `obj.meth` gets called twice, each created closure-instance will use its own version of `var`.

Impacts for verification

It is not possible to treat a local variable `var` used in method `meth` of object `obj` as a synthetic private attribute `obj.meth_var`, as one may think.

Let there be a closure defined in `meth` using `var`. If `obj.meth` gets called twice, each created closure-instance will use its own version of `var`.

All proposals

- it is possible to convert function-types to interface types with single abstract methods (sometimes called SAM-types)
- this way closures can be passed for example to the constructor of Thread,
  `new Thread(#(void) { aClient->startWork() })`.
- this is useful for all kind of callback-functions, which in Java are modelled by SAM-types; even the design pattern »Observer« uses this detour.
- the conversion exists due to compatiblity with the current emulation of closures by SAM-types

All proposals

- it is possible to convert function-types to interface types with single abstract methods (sometimes called SAM-types)
- this way closures can be passed for example to the constructor of Thread,
  `new Thread(#(void) { aClient->startWork() }).`
- this is useful for all kind of callback-functions, which in Java are modelled by SAM-types; even the design pattern »Observer« uses this detour.
- the conversion exists due to compatiblity with the current emulation of closures by SAM-types

All proposals

- it is possible to convert function-types to interface types with single abstract methods (sometimes called SAM-types)
- this way closures can be passed for example to the constructor of Thread,
  `new Thread(#(void) { aClient->startWork() })`.
- this is useful for all kind of callback-functions, which in Java are modelled by SAM-types; even the design pattern »Observer« uses this detour.
- the conversion exists due to compatiblity with the current emulation of closures by SAM-types

All proposals

- it is possible to convert function-types to interface types with single abstract methods (sometimes called SAM-types)
- this way closures can be passed for example to the constructor of Thread,
  `new Thread(#(void) { aClient->startWork() })`.
- this is useful for all kind of callback-functions, which in Java are modelled by SAM-types; even the design pattern »Observer« uses this detour.
- the conversion exists due to compatiblity with the current emulation of closures by SAM-types

BGGA

The BGGA-Proposal by Gilad Bracha, Neal Gafter, James Gosling and Peter von der Ahé uses an implicit return, like in

```
{ int x, int y => x := x + 2; y := y + 2; x + y }
```

The return value of a closure is the value of the last expression.

BGGA

The proposal does not only introduce method-calls in the functionality we have seen so far. Furthermore there are user-defined control-structures, like

```
withProtocol(aProtocol) {
  System.out.println("A");
  System.out.println("B");
}
```

User-defined control-structures

A possible implementation for `withProtocol` and an invocation of `withProtocol` with the techniques we have seen so far.

```
public static withProtocol(Protocol protocol,
                           {=> void} body) {
  protocol.inform();
  body.invoke();
  protocol.inform();
}

...
  withProtocol(aProtocol, {=> System.out.println("A");
                              System.out.println("B");
                           });
...
```

User-defined control-structures

A possible implementation for `withProtocol` (unchanged) and an invocation of `withProtocol` with the proposed improved syntax.

```
public static withProtocol(Protocol protocol,
                           {=> void} body) {
  protocol.inform();
  body.invoke();
  protocol.inform();
}

...
  withProtocol(aProtocol) {
    System.out.println("A");
    System.out.println("B");
  }
...
```

User-defined control-structures
Behavior of return and this

We have to be aware of the target of `return` (improved and regular syntax). This holds for `this`, too.

```
...
  withProtocol(aProtocol) {
    System.out.println("A");
    if (strangeError) return;
    System.out.println("B");
  }
...
...
  withProtocol(aProtocol, {=> System.out.println("A");
                              if (strangeError) return;
                              System.out.println("B");
                           });
...
```

We can even jump out of scope.

User-defined control-structures
Behavior of return and this

We have to be aware of the target of `return` (improved and regular syntax). This holds for `this`, too.

```
...
  withProtocol(aProtocol) {
    System.out.println("A");
    if (strangeError) return;
    System.out.println("B");
  }
...
...
  withProtocol(aProtocol, {=> System.out.println("A");
                              if (strangeError) return;
                              System.out.println("B");
                           });
...
```

We can even jump out of scope.

User-defined control-structures
Behavior of continue and break

We have to be aware of the target of `break` (improved syntax).

```
...
  while (!done) {
    withProtocol(aProtocol) {
      System.out.println("A");
      if (strangeError) break;
      System.out.println("B");
    }
    ...
  }
...
```

User-defined loops

An example for a user-defined loop.

```
public static for myLoop(int times, {=> void} body) {
  int i = 0;
  while (i < times) {
    i++; body.invoke();
  }
}

...
  for myLoop(5, {=> System.out.println("A");});
...

...
  for myLoop(5) {
    System.out.println("A");
  }
...
```

User-defined loops
Behavior of continue and break

This time `continue` refers to the next iteration of the user-defined control-structure (improved syntax).

```
...
  while (!done) {
    for myLoop(5) {
      System.out.println("A");
      continue;
      System.out.println("B");
    }
  ...
  }
...
```

CICE

The second proposal is »Concise Instance Creation Expressions: Closures without Complexity« by Bob Lee, Doug Lea and Josh Bloch.

This proposal does not define closures – it just introduces a more compact syntax for the creation of anonymous classes by omitting a few keywords and identifiers.

There are no function-types nor is there any ability to assign closures to a variable or return a closure back to a caller.

FCM

The last proposal is »First-class methods: Java-style closures« by Stephen Colebourne and Stefan Schulz.

This proposal allows usage of normal methods of classes and objects and even constructors as closures. For example we can use

```
#(int     (int, int))      cl1      = Math#min(int, int);
#(int     (Object))        cl2      = aList#indexOf(Object);
#(int     (List, Object)) cl3      = List#indexOf(Object);
#(Integer(int))            ctor     = Integer#(int);
#(void    ())              callback = this#callMe();

new Thread(this#callMe());
```

Summary

- local variables are not local any more, at least not in the temporal sense
- this may force verification to model an explicit heap
- the BGGA-proposal also introduces user-defined control-structures, requiring to capture the targets of `return`, `this`, `break` and `continue` and thus adding additional complexity
- the CICE-proposal does not introduce closures
- the FCM-proposal introduces everything one wants to have regarding closures and nothing beyond

Summary

- local variables are not local any more, at least not in the temporal sense
- this may force verification to model an explicit heap
- the BGGA-proposal also introduces user-defined control-structures, requiring to capture the targets of `return`, `this`, `break` and `continue` and thus adding additional complexity
- the CICE-proposal does not introduce closures
- the FCM-proposal introduces everything one wants to have regarding closures and nothing beyond

Summary

- local variables are not local any more, at least not in the temporal sense
- this may force verification to model an explicit heap
- the BGGA-proposal also introduces user-defined control-structures, requiring to capture the targets of `return`, `this`, `break` and `continue` and thus adding additional complexity
- the CICE-proposal does not introduce closures
- the FCM-proposal introduces everything one wants to have regarding closures and nothing beyond

Summary

- local variables are not local any more, at least not in the temporal sense
- this may force verification to model an explicit heap
- the BGGA-proposal also introduces user-defined control-structures, requiring to capture the targets of `return`, `this`, `break` and `continue` and thus adding additional complexity
- the CICE-proposal does not introduce closures
- the FCM-proposal introduces everything one wants to have regarding closures and nothing beyond

Summary

- local variables are not local any more, at least not in the temporal sense
- this may force verification to model an explicit heap
- the BGGA-proposal also introduces user-defined control-structures, requiring to capture the targets of `return`, `this`, `break` and `continue` and thus adding additional complexity
- the CICE-proposal does not introduce closures
- the FCM-proposal introduces everything one wants to have regarding closures and nothing beyond