Partial evaluation based on three-address form

© Miguel Garcia, LAMP, EPFL
http://lamp.epfl.ch/~magarcia

November 18th, 2011

Abstract

Performing partial evaluation [5] for arbitrary Scala ASTs requires tracking both side-effects and higher-order functions. A subset of those techniques, focusing on VM-level ASTs only, is also of practical interest. For example, they are enough to specialize a function "`power(base, n)`" to, say, "`power_3(base)`" (that raises `base` only to the third power, Sec. 1). More generally, those optimizations can be chosen that improve both space and time measures (thus always being sensible to apply) which is useful in particular when targeting mobile devices.

These notes describe a proof of concept for the capability sketched above. The compiler plugin in question ("`partialeval`") performs three tasks:

1. Converting ASTs after `cleanup` into three-address-form
2. As part of that, optimizations are performed, relying on the fact that side-effects are accounted for in the three-address form. This is more powerful than constant propagation alone. For example, "`if(m() * 0 == 0) n() else p();`" is reduced to "`m(); n();`". Given that these reductions are applied during the same traversal as (1) above, they come almost for free.
3. Conversion back to an expression language that `GenICode` can process. In this step all intermediate (i.e., SSA) variables are eliminated by replacing their single use with their RHS, to emit compact bytecode afterwards.

In terms of performance, the prototype is not competitive at this point because too few optimizations are included that would amortize the cost of getting in and out of three-address form. In particular, the all important iterative dataflow and method inlining are missing (Sec. 3). That can be solved however: the plugin is a testbed to gauge the potential of candidate optimizations (e.g., all the optimizations on Soot's Jimple [6] could also be applied between steps (2) and (3) above). Given that `partialeval` runs before `GenICode`, all backends stand to benefit from it (as well as future backends targeting non-stack based VMs that do without `GenICode`).

Another area where the prototype proves useful is understanding ASTs. Did you know that simplifying "`if(true) E1 else E2`" into "`E1`" isn't always safe? Any clue why? Details in Sec. 2.3.

Contents

1 Overview

1.1 How to build, run, and test

Following the standard recipe to package a compiler plugin:

1. compile `PartialEvaluation.scala` and `InfoOnGotos.scala` from `http://lampsvn.epfl.ch/trac/scala/browser/scala-experimental/trunk/partialeval/src`
2. the resulting classfiles (say, at `myplugins/partialeval/classes`) are packed with

   ```
   jar -cf partialeval.jar -C myplugins/partialeval/classes scala -C myplugins/partialeval/resources/ .
   ```

3. where `myplugins/partialeval/resources` contains the plugin manifest `scalac-plugin.xml`

   ```
   <plugin>
          <name>imp</name>
          <classname>scala.tools.imp.PEvalPlugin</classname>
   </plugin>
   ```

Afterwards, `scalac` is run with `-Xplugin:where/to/find/partialeval.jar` (on Linux) and `-Xplugin where/to/find/partialeval.jar` (on Windows).

A StackOverflow entry[1] covers how to use compiler plugins when building with Maven.

As for any semantics-preserving transformation, the Scala testsuite can be enlisted to compare the output of programs compiled with `-Xplugin partialeval.jar`. This amounts to a small addition in the `partest` script:

```
[ -n "$SCALAC_OPTS" ] || SCALAC_OPTS="-deprecation -Xplugin partialeval.jar" /*- don't forget the -Xplugin */
```

The above is for Windows. On Linux should read: `-Xplugin:where/to/find/partialeval.jar`.

1.2 Structure of the plugin

Helper utilities around GOTOs are grouped in the `InfoOnGotos` trait. Statistics gathered over all compilation units in a compiler run are tracked by a `Phase` subclass, so that at the end of `run()` it informs:

```
Console.println("[partialeval] Applied " + grandTotal + " optimizations. " +
                "Killed " + kGotos + " gotos, " + kLabels + " labels.")
```

Conversion to three-address and on-the-fly optimizations are the responsibility of the inheritance chain from `ThreeAddrMaker` to `Gatekeeper`. Besides `Gatekeeper`, another subclass (`BackToExprLang`) can also be instantiated:

```
def newTransformer(unit: CompilationUnit) =
  new BackToExprLang(unit) // either Gatekeeper or BackToExprLang
```

which will take care of invoking a transformer (`UndoThreeAddr`) that replaces (bottom-up) each use of an SSA variable with it single definition. This conversion is not strictly necessary (`GenICode` accepts the three-address form) but leads to faster loading of classfiles by the VM.

[1] `http://stackoverflow.com/questions/4955779/how-do-i-set-the-scala-compiler-to-use-a-plugin-when-i-build-us`

Listing 1: Sec. 1.3

```
object Bug257Test {
  def sayhello(): Unit = { Console.println("I should come 1st and 2nd"); };
  def sayhi(): Unit = { Console.println("I should come last"); };

  def f1(x: Unit): Unit = ();
  def f2(x: Unit)(y: Unit): Unit = ();

  def f(x: => Unit): Unit => Unit = {
    f1(x);
    f2(x);
  }

  def main(args: Array[String]) {
    f(sayhello())(sayhi())
  }
}
```

1.3 Passing closures to by-name and to strict params

It's instructive to see the three-address formulation of closures being passed as arguments, for example as shown in Listing 1 (that's `files/run/bugs.scala` from the Scala testsuite). The difference can be seen for function `f()` (whose param is by-name) vs. functions `f1()` and `f2()` (whose params are strict).

1. Right after `CleanUp`, `main()` looks as follows:

```
def main(args: Array[String]): Unit = {
  Bug257Test.this.f({
    (new anonymous class Bug257Test$$anonfun$main$1(): Function0)
  }).apply({
    Bug257Test.this.sayhi();
    scala.runtime.BoxedUnit.UNIT
  });
  ()
}
```

2. And the three-address formulation of the body of that method is:

```
{
  val tmp13: Function0 = {
    val tmp11: anonymous class Bug257Test$$anonfun$main$1 =
        new anonymous class Bug257Test$$anonfun$main$1();
    val tmp12: Function0 = tmp11;
    tmp12
  };
  val tmp14: Function1 = Bug257Test.this.f(tmp13);
  val tmp16: scala.runtime.BoxedUnit = {
    Bug257Test.this.sayhi();
    val tmp15: scala.runtime.BoxedUnit = scala.runtime.BoxedUnit.UNIT;
    tmp15
  };
  val tmp17: Object = tmp14.apply(tmp16);
  tmp17;
  ()
}
```

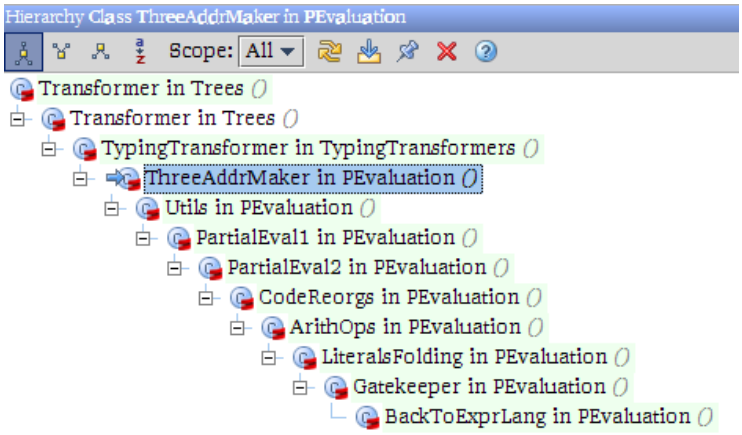

Figure 1: Sec. 2.1

3. Finally, after `val expressified = resolver.back2SL(tree.symbol, a3tree)` has run, the formulation devoid of SSA variables looks as below. *All this to show* that side-effects are produced as in the original program:

```
{
  Bug257Test.this.f(new anonymous class Bug257Test$$anonfun$main$1()).apply({
    Bug257Test.this.sayhi();
    scala.runtime.BoxedUnit.UNIT
  });
  ()
}
```

2 On-the-fly reductions

2.1 Laundry list

The subclasses of `ThreeAddrMaker` depicted in Figure 1 perform the following reductions, grouped by the class realizing them (for an example see Sec. 2.2):

1. `PartialEval1` and `PartialEval2`

   (a) `If` nodes with compile-time-known condition value.

   ```
   From
     if(true) { then-branch } [ else-branch ]
   To
     then-branch
   Similarly for if(false) ...
   ```

   (b) Match expression with compile-time-known selector value.

   ```
   From
     literal match { case literal_1 => e_1; case literal_n => e_n }
   To
     e_i, where literal == literal_i
   ```

(c) conditional-and, conditional-or (more on this in Sec. 2.2).

2. `CodeReorgs`

(a)
```
From
  try { <empty> } catches { <non-empty> } finally { F }
To
  try { <empty> } finally { F }
```

(b) An `If` node with negated condition, with both then and else branches. Simplify the condition and exchange branches. Sensible in theory but causes `GenICode` to complain (e.g., for `Patterns.scala`) about an "Unknown label target" (i.e., after exchanging the branches, there's in the then-branch a GOTO to a `LabelDef` deep in the else branch). Therefore we *further* constrain the applicability condition, to those cases where no label shows up in the `If`.

```
From
  if(!b) { e1 } else { e2 }
To
  if( b) { e2 } else { e1 }
```

3. `ArithOps`

```
From
  an integral multiplication a * b, where (a == 0) || (b == 0)
To
  the operand that is literal zero.
```

4. `LiteralsFolding`

"`constfold`": Reduction of expressions involving only compile-time constants and well-known operations. Sometimes similar reductions can be devised for cases where only some operands' values are compile-time-known.

2.2 Division of labor between `ThreeAddrMaker` and its subclasses ("`conditional-and`" example)

A previous prototype ("`imp`", [4]) turned post-`cleanup` ASTs into three-address form. That functionality remains in place but, due to subclasses of `ThreeAddrMaker`, the trees that `transform()` returns are not only in three-address form but moreover have been optimized. For example, an override handles short-circuit-and (Listing 2) where, for example, "`false && b`" is reduced to "`false`" and no code is emitted to evaluate "`b`". The "`false`" operand in turn may have resulted itself from a reduction.

In contrast, without the `transform()` override in class `PartialEval2`, the conversion to three-address alone would have transformed "`op1 conditional-and op2`" into "`if(op1) { op2 } else { false }`" (as shown below) i.e. code would have been emitted for the 2nd operand.

Listing 2: Sec. 2.2

```
/*
   Short-circuit-And.

   The reduction from 'false && b' to 'false' is safe because no code is emitted to evaluate 'b'
     (i.e., we don't call 'transform('b')'.
   To recap, transform(b) returns 'b' converted into three-address-form.

 */
case Apply(fun @ Select(rand1, _), List(rand2)) if fun.symbol == definitions.Boolean_and =>
  val a3rand1 = transform(rand1)
  if     (isLiteralFalse(a3rand1)) {
    appliedOptimizations += 1
    elideAccess(a3rand1)
    kill(rand2)
    LITFalse(rand1.pos)
  } else if(isLiteralTrue(a3rand1)) {
    elideAccess(a3rand1)
    val a3rand2 = transform(rand2)
    if(isLiteralFalse(a3rand2)) {
      appliedOptimizations += 1
      elideAccess(a3rand2)
      LITFalse(rand2.pos)
    } else if(isLiteralTrue(a3rand2)) {
      appliedOptimizations += 1
      elideAccess(a3rand2)
      LITTrue(rand2.pos)
    } else { a3rand2 }
  } else {
    val newAnd = treeCopy.Apply(tree, treeCopy.Select(fun, a3rand1, fun.name), List(rand2))
    super.transform(newAnd)
  }
```

```
case Apply(fun @ Select(op1, _), List(op2)) if fun.symbol == definitions.Boolean_and =>
  // 'op1 conditional-and op2' is rewritten into 'if(op1) { op2 } else { false }'
  val res = typedWithPos(app.pos) { If(op1, op2, Literal(Constant(false))) }
  transform(res) // transform() needed to turn the If into a stmt, that's where op1, op2 will be transformed.
```

2.3 Labels and their GOTOs

Usually "`if(true) E1 else E2`" can be reduced to "`E1`". Unless the else-branch happens to contain a `LabelDef` that is targeted from some jump which is not dead code (background info on all things labels and jumps at [3]).

Therefore, we adopt a conservative applicability condition that prevents from marking as dead any branch containing live `LabelDef`s. Those live labels could well become unreachable later, after their GOTOs have been killed, but in its current form `partialeval` doesn't try that hard to kill that code. Rather, it marks via "`enqueueKillRequest(tree.asInstanceOf[If], thenp)`" such intention, in anticipation of iterative dataflow capability (Sec. 3).

The relevant snippet is shown in Listing 3.

Listing 3: Sec. 2.3

```
case If(condp, thenp, elsep) =>

  val a3Cond = transform(condp)

    def conservative() = {
      val newIf = treeCopy.If(tree, a3Cond, thenp, elsep)
      super.transform(newIf)
    }

  if(isLiteralTrue(a3Cond)) {
    if(lacksLiveLabels(elsep)) {
      appliedOptimizations += 1
      elideAccess(a3Cond)
      kill(elsep)
      transform(thenp)
    } else {
      enqueueKillRequest(tree.asInstanceOf[If], elsep)
      conservative()
    }
  } else if(isLiteralFalse(a3Cond)) {
    if(lacksLiveLabels(thenp)) {
      appliedOptimizations += 1
      elideAccess(a3Cond)
      kill(thenp)
      transform(elsep)
    } else {
      enqueueKillRequest(tree.asInstanceOf[If], thenp)
      conservative()
    }
  } else conservative()
```

3 Iterative reductions

```
TODO To be added as a transformer running in between
    getting in and out of three-address form.
```

4 Conversion back to an expression language

The structure of the transformers in charge of converting into and out of three-address are similar (both sport `transform()`, `pushXpop()`, and delegate transforming certaing tree shapes to similarly-named helper methods). In fact, `UndoThreeAddr` can be seen as a "parser" for the instruction stream that "`ThreeAddrMaker`" serialized. The main differences between both kinds of transformers follow.

An `UndoThreeAddr.transform(expr)` step starts by poping from the stack of previous instructions the RHSs for SSA-usages occurring in `expr`. In turn, the invoker of `transform(expr)` will push that expanded expression onto the stack. That stack is summarily called "`seen`" (to convey the idea that, well, those expressions have been seen already). It all starts with:

```
   override def transform(tree0: Tree): Tree = {
```

Listing 4: Sec. 4

```
private def preTrans(stmt: Tree): Tree = {
  val usages = { usagesColltor.hits.clear; usagesColltor.traverse(stmt); usagesColltor.hits.toList.reverse }
  val newStmt0 =
    if(usages.isEmpty) {
      stmt
    } else {
      val rhsides = {
        val usagesIter = usages.iterator
        val rhssidesBuffer = collection.mutable.ListBuffer.empty[Tree]
        while(usagesIter.hasNext) {
          val use = usagesIter.next

          var futureStats: List[Tree] = Nil
          var stop = false;
          do {
            futureStats = seen.pop :: futureStats;
            stop = futureStats.head.isInstanceOf[ValDef] && (futureStats.head.symbol == use)
          } while (!stop)

          val ValDef(_, _, _, rhs) = futureStats.head
          rhssidesBuffer += rhs
          futureStats.tail foreach { s =>
            assert(s.isInstanceOf[ValDef] && isIntermediate(s))
            seen.push(s)
          }
        }
        rhssidesBuffer.toList
      }
      usages foreach { u => consumeVar(u) }
      new TreeSubstituter(usages, rhsides) transform stmt
    }
  newStmt0
}
```

```
    val tree = preTrans(tree0)

    val res = tree match {

    . . .
```

As suggested above, the method in charge of expanding SSA-usages with their single-definitions is `preTrans()` (Listing 4), where the replacement itself is carried out by "`new TreeSubstituter(usages, rhsides) transform stmt`".

5 Future work

There are a number of optimizations that can be added (like, all those performed by Soot's Jimple [6]). In particular the interplay of value propagation and method inlining has potential (which goes by the name of "*Polyvariant program-point specialization*", [5]):

- From:

```
int power(int base, int n)
{
  int pow;
  for (pow = 1; n; n--)
    pow *= base;
  return pow;
}
```

- To:

```
int power_3(int base)
{
  int pow;
  pow = 1;
  pow *= base;
  pow *= base;
  pow *= base;
  return pow;
}
```

Having a comprehensive set of optimizations in the form of one or more `Transformer`s is all fine. An intriguing idea is making that functionality configurable, by specifying optimizations in terms of a declarative language. After all, the transformation rules in `ArithOps` (Sec. 2.1) could have been given as algebraic *rules*. Such declarative language would enormously lower the barrier of entry for library-specific optimizations, the best example of which are those for MapReduce, so far implemented in dedicated compilers [2].

6 Decompiling bytecode in external libraries into three-address form (e.g., to enable optimizations)

Currently, the Scala compiler can parse bytecode into ICode, but stops short of "decompiling" those ICode ASTs into "Scala ASTs in stack-based style with GOTOs" (unlike Scala source, Scala ASTs may contain arbitrary GOTOs [3], as well as pre-super field initialization).

1. One way to obtain "Scala ASTs in stack-based style with GOTOs" from ICode can be found in the tutorial by Bebenita on obtaining SSA from bytecode, `http://michael.bebenita.com/storage/ssa_class_notes.pdf` From there, the existing `imp` plugin can convert those ASTs to three-address style. One of the difficulties involves structured exception handling. Given that neither Java compilers nor Scala emit non-nested protected blocks for exception handling, that's the subset of all-feasible exception handling patterns we want to address.

2. Another way is exemplified by Soot, in particular `coffee.CFG`. As with other "decompilers", a three-address form is obtained directly. For this, [1] is a useful reference ("Bellamy's type inference for local variables"). Code to look at: `soot.jimple.toolkits.typing.fast.TypeResolver`. One of its steps need not be performed for ICode (disambiguating small integers),

that's done in `soot.jimple.toolkits.typing.fast.TypeResolver.typePromotion(Typing)`. The original Soot algorithm for inserting casts (still necessary in `fast.TypeResolver` in a few cases) is implemented by `soot.jimple.toolkits.typing.integer.TypeResolver.resolve(JimpleBo`

3. Another way to obtain three-address code is using `sawja`, as described at

   ```
   Delphine Demange, Thomas Jensen, and David Pichardie.
   A provably correct stackless intermediate representation for Java bytecode.
   In Proc. of the 8th Asian Symposium on Programming Languages and Systems (APLAS 2010),
   Vol 6461 of LNCS, pages 97-113. Springer-Verlag, 2010.
   http://www.irisa.fr/celtique/pichardie/papers/aplas10.pdf
   ```

References

[1] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. In Gregor Kiczales, editor, *OOPSLA'07: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2008. `http://progtools.comlab.ox.ac.uk/publications/oopsla08abc`.

[2] Michael J. Cafarella and Christopher Ré. Manimal: relational optimization for data-intensive programs. In *Procceedings of the 13th International Workshop on the Web and Databases*, WebDB '10, pages 10:1–10:6, New York, NY, USA, 2010. ACM. Available online at `http://www.eecs.umich.edu/~michjc/papers/WebDB-Manimal.pdf`.

[3] Miguel Garcia. GOTO elimination for Scala ASTs, 2011. Notes at *The Scala Compiler Corner*. `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q1/JumpsRemover.pdf`.

[4] Miguel Garcia. Moving Scala ASTs one step closer to C, 2011. Notes at *The Scala Compiler Corner*. `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q2/Moving3A.pdf`.

[5] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[6] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, McGill University, 1998. `http://www.sable.mcgill.ca/publications/techreports/sable-tr-1998-4.ps`.