SPSC: a Simple Supercompiler in Scala

Ilya Klyuchnikov, Sergei Romanenko

presented by Andrei Klimov

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Program Analysis and Transformation Sector
http://pat.keldysh.ru

PU'09
To program is to understand

Kristen Nygaard
A program should reflect an understanding of the problem domain
We would like...

- To demystify supercompilation for a programmer
- In order to do this we want:
  - To present the core of supercompilation in the form of a program
  - The code of the supercompiler should be small and clear
  - A minimalistic input language
Supercompilation in a Nutshell

\[
gApp(\text{Nil}(), \text{vs}) = \text{vs}; \\
gApp(\text{Cons}(u, \text{us}), \text{vs}) = \text{Cons}(u, gApp(\text{us}, \text{vs}));
\]

\[
gApp(gApp(xs, ys), zs)
\]
Supercompilation in a Nutshell

gApp(\text{Nil}(), \text{vs}) = \text{vs};
gApp(\text{Cons}(u, \text{us}), \text{vs}) = \text{Cons}(u, \text{gApp}(\text{us}, \text{vs}));
Supercompilation in a Nutshell

\[
gApp(\text{Nil}(), \text{vs}) = \text{vs};
gApp(\text{Cons}(u, \text{us}), \text{vs}) = \text{Cons}(u, gApp(\text{us}, \text{vs}));
\]
Supercompilation in a Nutshell

\[
gApp(Nil(), vs) = vs; \\
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs)); \\
gApp(gApp(xs, ys), zs) \\
gApp(Cons(u, gApp(us, ys)), zs) \\
Cons(u, gApp(gApp(us, ys), zs)) \\
\]

\[
xs = Nil \\
gApp(ys, zs) \\n\]

\[
xs = Cons(u, us) \\
gApp(ys, zs) \\n\]

\[
u \\
gApp(gApp(us, ys), zs) \\n\]
Supercompilation in a Nutshell

\[ g\text{App}(\text{Nil}(), \text{vs}) = \text{vs}; \]
\[ g\text{App}(\text{Cons}(u, \text{us}), \text{vs}) = \text{Cons}(u, g\text{App}(\text{us}, \text{vs})); \]
Supercompilation in a Nutshell

gApp(Nil(), vs) = vs;
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
Supercompilation in a Nutshell

\[
gApp(\text{Nil}(), \text{vs}) = \text{vs};
gApp(\text{Cons}(u, us), \text{vs}) = \text{Cons}(u, gApp(us, vs));
\]
Supercompilation in a Nutshell

\[
g\text{App}(\text{Nil}(), \text{vs}) \quad = \quad \text{vs}; \\
g\text{App}(\text{Cons}(u, \text{us}), \text{vs}) \quad = \quad \text{Cons}(u, g\text{App}(\text{us}, \text{vs}));
\]

\[
g\text{App}1(\text{Nil}(), y, z) \quad = \quad g\text{App}2(y, z); \\
g\text{App}1(\text{Cons}(v1, v2), y, z) \quad = \quad \text{Cons}(v1, g\text{App}1(v2, y, z));
\]

\[
g\text{App}2(\text{Nil}(), z) \quad = \quad z; \\
g\text{App}2(\text{Cons}(v3, v4), z) \quad = \quad \text{Cons}(v3, g\text{App}2(v4, z));
\]
## Quest for practical supercompilers

**1974**  
V. Turchin presented supercompilation to a group of students at seminars in Moscow

**1980s**  
V. Turchin developed first supercompilers for the functional programming language Refal (CUNY, New York)

**1980s – 1990s**  
Papers by V. Turchin on supercompilation of Refal

**1993 – 2000s**  
Andrei Nemytykh (IPS RAS, Pereslavl-Zalessky) continued work on Turchin’s supercompiler and completed it

**1998 – 2000s**  
Java Supercompiler by Andrei Klimov, Arkady Klimov and Artem Shvorin (Keldysh Institute of Applied Mathematics, RAS, Moscow)

[AK]
Quest for understandable supercompilers

1986  V. Turchin "The Concept of a Supercompiler". ACM TOPLAS 8(3): 292-325

...

1993  The first understandable supercompiler and paper by R. Glück and A. Klimov "Occam's razor in metacomputation: the notion of a perfect process tree". LNCS 724: 112-123

1993  Book by S. Abramov "Metacomputation and its application" (in Rus)

...

2009  I. Klyuchnikov, S. Romanenko "SPSC: a Simple Supercompiler in Scala". PU'09
Input Language

\[ p ::= d_1 \ldots d_n \text{ program}\]

\[ d ::= f(x_1, \ldots, x_n) = e; \quad \text{f-function}\]

\[ | g(q_1, x_1, \ldots, x_n) = e_1; \quad \text{g-function}\]

\[ \ldots \]

\[ g(q_m, x_1, \ldots, x_n) = e_m; \]

\[ e ::= x \text{ variable}\]

\[ | c(e_1, \ldots, e_n) \quad \text{constructor}\]

\[ | f(e_1, \ldots, e_n) \quad \text{call to f-function}\]

\[ | g(e_1, \ldots, e_n) \quad \text{call to g-function}\]

\[ q ::= c \, v_1 \ldots v_n \quad \text{pattern}\]
Implementation language??

- Easy to understand
- Easy to use (IDE, debugger, libs, …)
- Functional
- Also cool
Scala makes a buzz

- FP support
- OOP support
- Compiles to JVM bytecode
- Ready for production – Twitter is rewritten into Scala
Scala makes a buzz

- IDE Support
- Convenient debugger
- Great community
- Strong theoretical base
abstract class Def { def name: String 

    class FFun(name: String, args: List[Var], term: Term) extends Def {
        override def toString = 
            name + args.mkString("(" + ", " + ", " + ")") + " = " + term + ";
    }

    class GFun(name: String, p: Pat, args: List[Var], term: Term) extends Def {
        override def toString = 
            name + (p :: args).mkString("(" + ", " + ", " + ")") + " = " + term + ";
    }
}
abstract class Def {def name: String}

case class FFun(name: String, args: List[Var], term: Term) extends Def {
  override def toString = 
    name + args.mkString("(" ,", " ,")") + " = " + term + ";
}

case class GFun(name: String, p: Pat, args: List[Var], term: Term) extends Def {
  override def toString = 
    name + (p :: args).mkString("(" ,", " ,")") + " = " + term + ";
}

case class Program(defs: List[Def]){
  val f = (defs :\ (Map[String, FFun]()))
  {case (x: FFun, m) => m + (x.name -> x); case (_, m) => m}
  val g = (defs :\ (Map[(String, String), GFun]()))
  {case (x: GFun, m) => m + ((x.name, x.p.name) -> x); case (_, m) => m}
  val gs = (defs :\ Map[String, List[GFun]]().withDefaultValue(Nil))
  {case (x: GFun, m) => m + (x.name -> (x :: m(x.name))); case (_, m) => m}
  override def toString = defs.mkString("\n")
}
abstract class Def {def name: String}

case class FFun(name: String, args: List[Var], term: Term) extends Def {
  override def toString = 
    name + args.mkString("(" , ", " , ")") + " = " + term + ";"
}

case class GFun(name: String, p: Pat, args: List[Var], term: Term) extends Def {
  override def toString = 
    name + (p :: args).mkString("(" , ", " , ")") + " = " + term + ";"
}

case class Program(defs: List[Def]){
  val f = (defs :\ (Map[String, FFun]()))
    {case (x: FFun, m) => m + (x.name -> x); case (_, m) => m}
  val g = (defs :\ (Map[[String, String], GFun]()))
    {case (x: GFun, m) => m + ((x.name, x.p.name) -> x); case (_, m) => m}
  val gs = (defs :\ Map[String, List[GFun]]().withDefaultValue(Nil))
    {case (x: GFun, m) => m + (x.name -> (x :: m(x.name))); case (_, m) => m}
  override def toString = defs.mkString("\n")
}
Scala by Example
higher-order functions, almost any names for defs...

abstract class Def {def name: String}

case class FFun(name: String, args: List[Var], term: Term) extends Def {
  override def toString =
    name + args.mkString("(", ", " ,")") + " = " + term + ";
}

case class GFun(name: String, p: Pat, args: List[Var], term: Term) extends Def {
  override def toString =
    name + (p :: args).mkString("(", ", " ,")") + " = " + term + ";
}

case class Program(defs: List[Def]){
  val f = (defs :\ (Map[String, FFun]()))
    {case (x: FFun, m) => m + (x.name -> x); case (_, m) => m}
  val g = (defs :\ (Map[(String, String), GFun]() )
    {case (x: GFun, m) => m + ((x.name, x.p.name) -> x); case (_, m) => m}
  val gs = (defs :\ Map[String, List[GFun]]() .withDefaultValue(Nil))
    {case (x: GFun, m) => m + (x.name -> (x :: m(x.name))); case (_, m) => m}
  override def toString = defs.mkString("\n")
}
def findSubst(t1: Term, t2: Term) = {
    val map = scala.collection.mutable.Map[Var, Term]()
    def walk(t1: Term, t2: Term): Boolean = (t1, t2) match {
        case (v1: Var, _) =>
            map.put(v1, t2) match {
                case None => true
                case Some(t3) => t2 == t3
            }
        case (e1: CFG, e2: CFG) if shellEq(e1, e2) =>
            List.forall2(e1.args, e2.args)(walk)
        case _ => false
    }
    if (walk(t1, t2)) map.readOnly else null
}
Scala by Example

to mutate (and how) or not to mutate?

def findSubst(t1: Term, t2: Term) = {
  var map = Map[Var, Term]()
  def walk(t1: Term, t2: Term): Boolean = (t1, t2) match {
    case (v1: Var, _) => map.get(v1) match {
      case None => map += (v1 -> t2); true
      case Some(t3) => t2 == t3
    }
    case (e1: CFG, e2:CFG) if shellEq(e1, e2) =>
      List.forall2(e1.args, e2.args)(walk)
    case _ => false
  }
  if (walk(t1, t2)) map else null
}
object SParsers extends StandardTokenParsers with ImplicitConversions {
    ...
    def prog = definition+
    def definition: Parser[Def] = gFun | fFun
    def term: Parser[Term] = fcall | gcall | ctr | vrb
    def vrb = lid ^^ Var
    def pat = uid ~ ("(" ~> repsep(vrb, "," ) <~ ")") ^^ Pat
    def fFun =
        fid ~ ("(" ~> repsep(vrb, "," ) <~ ")") ~ ("=" ~> term <~ ";") ^^ FFun
    def gFun =
        gid ~ ("(" ~> pat) ~ ((("," ~> vrb)*) <~ ")") ~ ("=" ~> term <~ ";") ^^ GFun
    def ctr = uid ~ ("(" ~> repsep(term, "," ) <~ ")") ^^ Ctr
    def fcall = fid ~ ("(" ~> repsep(term, "," ) <~ ")") ^^ FCall
    def gcall = gid ~ ("(" ~> repsep(term, "," ) <~ ")") ^^ GCall
}
Time to try it

Input code

append(nil, vs) = vs;
append(cons(a, uz), vs) = cons(a, append(uz, vs));
appendxy(a, yz, za) = append(appendx(a, yz), za);

Function to be supercompiled

appendxya2

Supercompiled code

append1(nil(), a) = a;
append1(cons(a, b), c) = cons(a, append1(b, c));
append2(nil(), a) = a;
append2(cons(a, b), c) = cons(a, append2(b, c));
appendxy(a, b, c) = append1(append2(a, b), c);

Partial process tree
Thanks!

Writing it is easy, understanding it is hard.

Anonymous

http://pat.keldysh.ru