

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



Ole Lehrmann Madsen

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

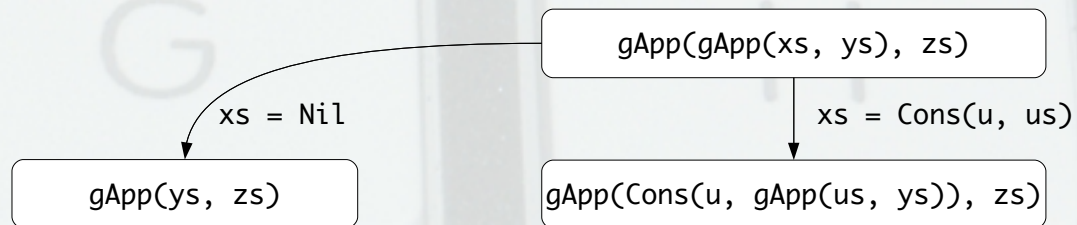
Supercompilation in a Nutshell

```
gApp(Nil(), vs) = vs;  
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
```

```
gApp(gApp(xs, ys), zs)
```

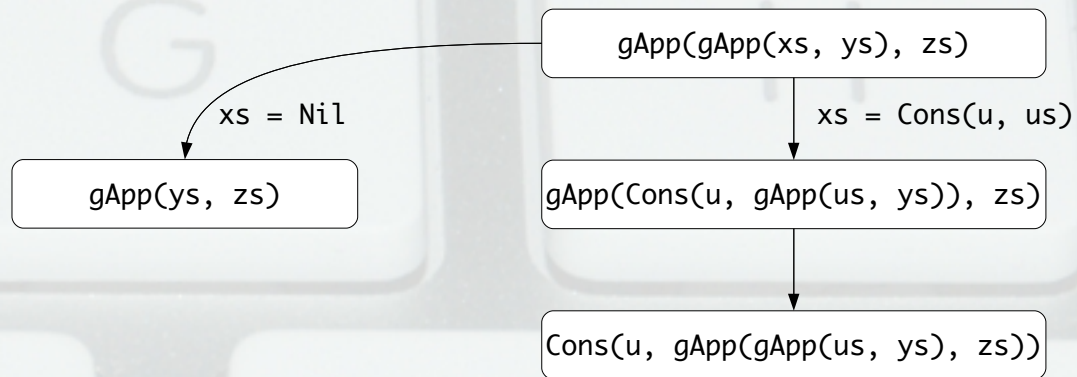
Supercompilation in a Nutshell

```
gApp(Nil(), vs) = vs;  
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
```



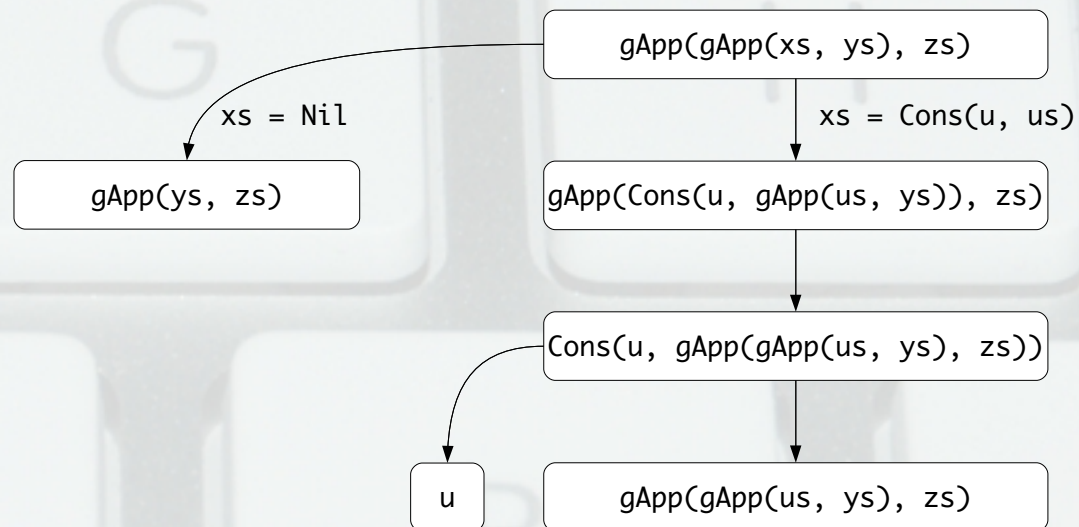
Supercompilation in a Nutshell

```
gApp(Nil(), vs) = vs;  
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
```



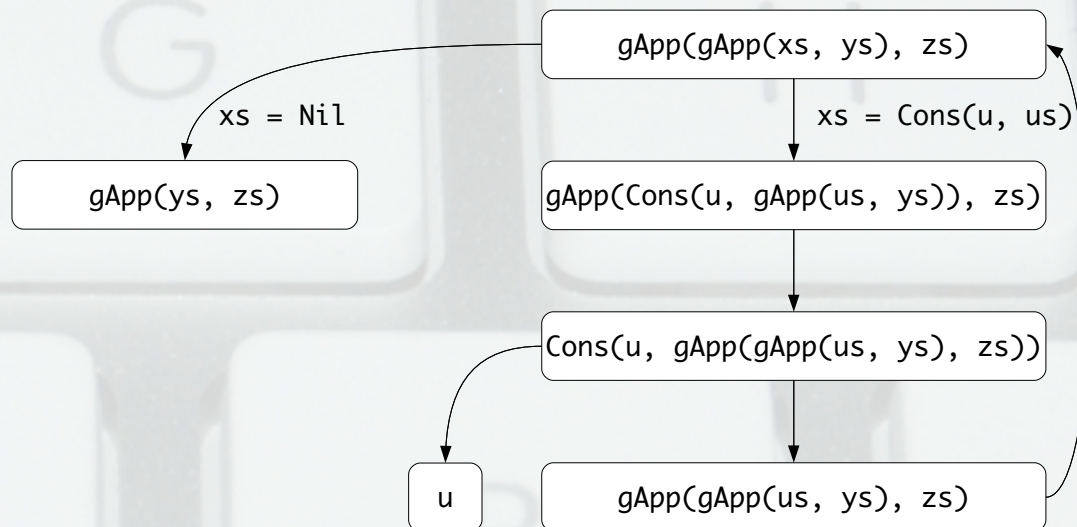
Supercompilation in a Nutshell

```
gApp(Nil(), vs) = vs;  
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
```



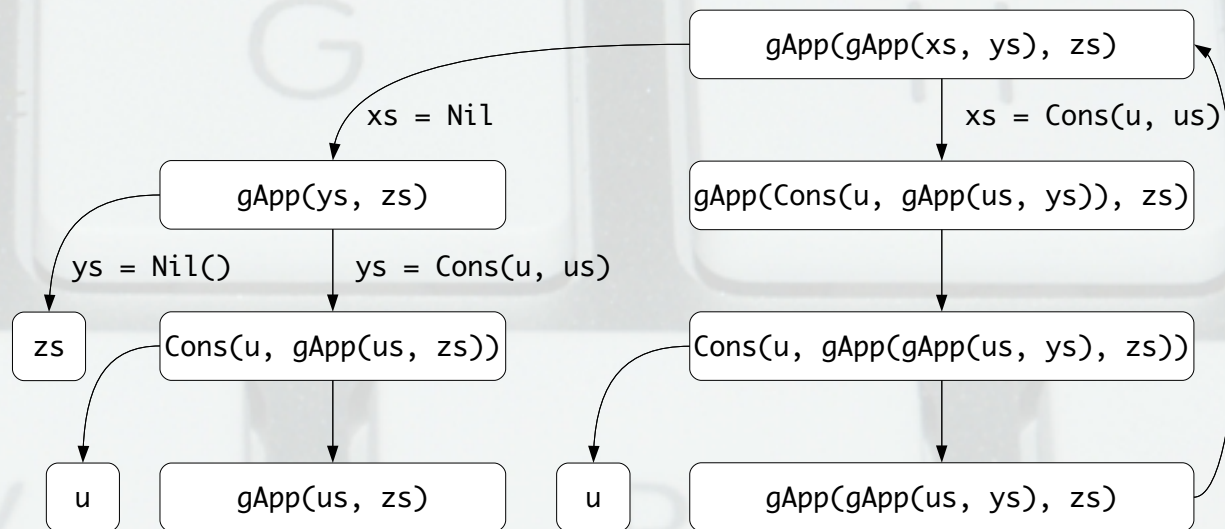
Supercompilation in a Nutshell

```
gApp(Nil(), vs) = vs;  
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
```



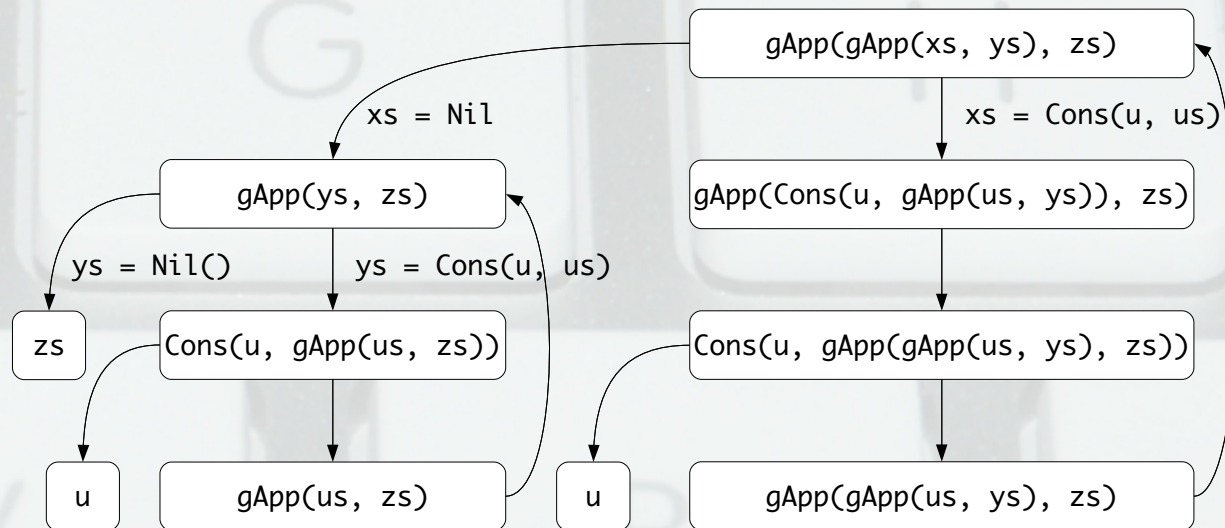
Supercompilation in a Nutshell

```
gApp(Nil(), vs)           = vs;  
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
```



Supercompilation in a Nutshell

```
gApp(Nil(), vs)           = vs;  
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
```



Supercompilation in a Nutshell

```
gApp(Nil(), vs) = vs;  
gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));
```

```
gApp(gApp(xs, ys), zs)
```

```
gApp3(xs, ys, zs)
```

```
gApp3(Nil(), y, z) = gApp2(y, z);  
gApp3(Cons(v1, v2), y, z) = Cons(v1, gApp3(v2, y, z));
```

```
gApp2(Nil(), z) = z;  
gApp2(Cons(v3, v4), z) = Cons(v3, gApp2(v4, z));
```

Quest for practical supercompilers

1974	V. Turchin presented supercompilation to a group of students at seminars in Moscow
1980...	V. Turchin developed first supercompilers for the functional programming language Refal (CUNY, New York)
1980 - 1990...	Papers by V. Turchin on supercompilation of Refal
1993 - 2000...	Andrei Nemytykh (IPS RAS, Pereslavl-Zalessky) continued work on Turchin's supercompiler and completed it
1998 - 2000...	Java Supercompiler by Andrei Klimov, Arkady Klimov and Artem Shvorin (Keldysh Institute of Applied Mathematics, RAS, Moscow)

Quest for understandable supercompilers

- | | |
|------|---|
| 1986 | V. Turchin "The Concept of a Supercompiler" |
| 1988 | V. Turchin "The Algorithm of Generalization in the Supercompiler" |
| 1993 | R. Glück and A. Klimov "Occam's razor in metacomputation: the notion of a perfect process tree" |
| 1993 | S. Abramov "Metacomputation and its applications" (in rus) |
| 1996 | M.H. Sørensen, R. Glück, N.D. Jones "A Positive Supercompiler" |
| 2009 | I. Klyuchnikov, S. Romanenko "SPSC: a Simple Supercompiler in Scala" |

Input Language

$p ::= d_1 \dots d_n$ program

$d ::= f(x_1, \dots, x_n) = e;$ f-function
| $g(q_1, x_1, \dots, x_n) = e_1;$ g-function
...
| $g(q_m, x_1, \dots, x_n) = e_m;$

$e ::= x$ variable
| $c(e_1, \dots, e_n)$ constructor
| $f(e_1, \dots, e_n)$ call to f-function
| $g(e_1, \dots, e_n)$ call to g-function

$q ::= c v_1 \dots v_n$ pattern

Implementation language??

- Easy to understand
- Easy to use (IDE, debugger, libs, ...)
- Functional
- Also cool

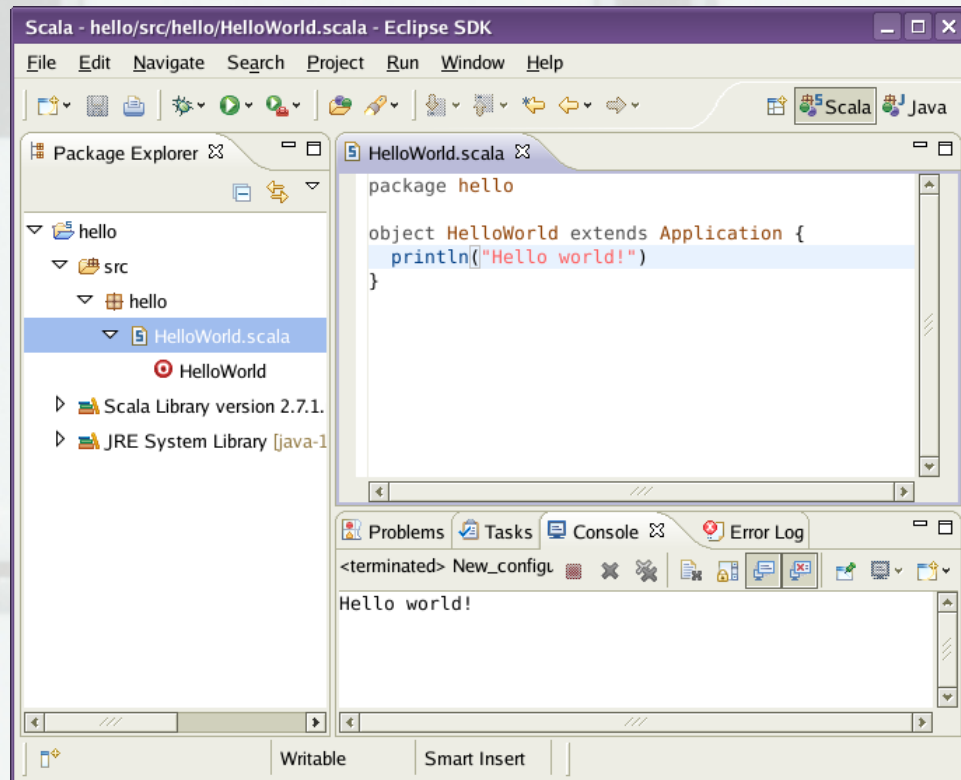
Scala makes a buzz



Scalable Language

- 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

Scala by Example

pragmatic OOP

```
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 + ";"  
  }
```

Scala by Example

pattern matching with case classes

```
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

functional objects

```
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")
}
```

```
val g = p.g(name, cname)
```

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")
}
```

Scala by Example

to mutate (and how) or not to mutate?

```
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  
}
```

Scala by Example

real DSL

```
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

spsc_web beta

http://localhost:8080/spsc_web/spsc_result

spsc_web beta

Input code

```
append(Nil, vs) = vs;  
append(Cons(u, us), vs) = Cons(u, append(us, vs));  
appendXYaZ(xs, ys, zs) = append(append(xs, ys), zs);
```

Function to be supercompiled

```
appendXYaZ
```

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));  
appendXYaZ(a, b, c) = append1(append2(a, b), c);
```

Partial process tree

```
graph TD  
  A[appendXYaZ(xs, ys, zs)] --> B[let $248=append(xs, ys), $249=zs in append($248, $249)]  
  A --> C[append(xs, ys)]  
  A --> D[zs]  
  B --> E[append($248, $249)]  
  B --> D  
  E --> F[$248=Cons($256, $257)]  
  E --> G[$248=Nil()]  
  C --> H[xs=Cons($250, $251)]  
  C --> I[xs=Nil()]
```

Thanks!

[SPSC: a Simple Supercompiler in Scala](#)

<http://spsc.appspot.com/>

<http://hosc.appspot.com/>

<http://pat.keldysh.ru/>