Accepted for PU'09 - International Workshop on Program Understanding,
19-23 June, Altai Mountains, Russia

SPSC: a Simple Supercompiler in Scala *

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

1 Introduction

Supercompilation [3,4,5] is a program transformation technique, suggested by V.F. Turchin in the early 1970-s. There is a good deal of literature devoted to supercompilation. For the most part, however, the presentations suffer from the techniques being described either in a fragmentary and incomplete way, or at an excessively high level of abstraction (as specifications or inference rules, rather than concrete algorithms). As a result, the suggested techniques appear to be difficult to test for practical usability. If a presentation is incomplete, the reader is supposed to be able to guess the missing parts. This puzzle may be a non-trivial one (besides, there is no guarantee that the reader's solution will be the same as expected by the paper's authors). On the other hand, if the level of a presentation is too abstract, turning it into ready-to-use programs may be an intricate task.

Thus, from a working programmer's perspective, supercompilation appears as something obscure, esoteric and sophisticated in implementation. And, in turn, without an implementation at hand, the potential of supercompilation may be difficult to judge.

The goal of the paper is to demystify supercompilation by presenting the complete source code of a simple, yet operational supercompiler. This may be helpful for a working programmer giving him an opportunity to familiarize himself with supercompilation formulated in terms of his "native language", i.e. in form of concrete programs. Besides, these programs are complete and ready to run, so that the supercompilation techniques described in the paper can be immediately tried in practice.

We use Scala [1] as an implementation language. The reason is that various algorithms related to supercompilation are different in nature. Some algorithms are easy to formulate in functional style, while others can be naturally written in object-oriented terms. A peculiarity of Scala is that it smoothly integrates features of object-oriented and functional programming, including mixins, algebraic datatypes with pattern matching, genericity, and more. In addition, Scala is a production programming language seamlessly integrated with underlying implementation platform (Java).

The paper describes a simple supercompiler SPSC for a lazy first-order functional language [2]. Each section of the paper starts by introducing an important

* Supported by Russian Foundation for Basic Research projects No. 08-07-00280-a and No. 09-01-00834-a.

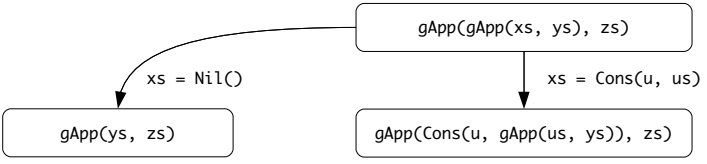
aspect of supercompilation and then provides an implementation of the corresponding algorithms in Scala. We assume the reader to be familiar with Java (or other modern object-oriented language). The features of Scala that are not present in Java are given short explanations.

2 Supercompilation in a Nutshell

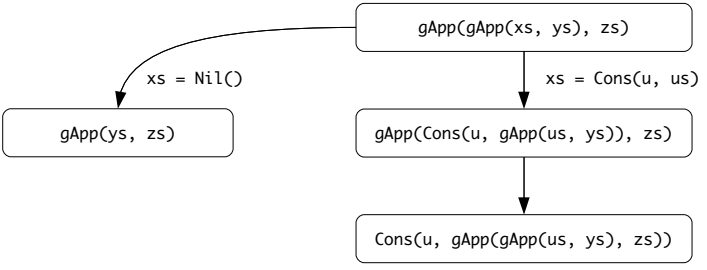
Consider a simple functional program appending two lists:

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

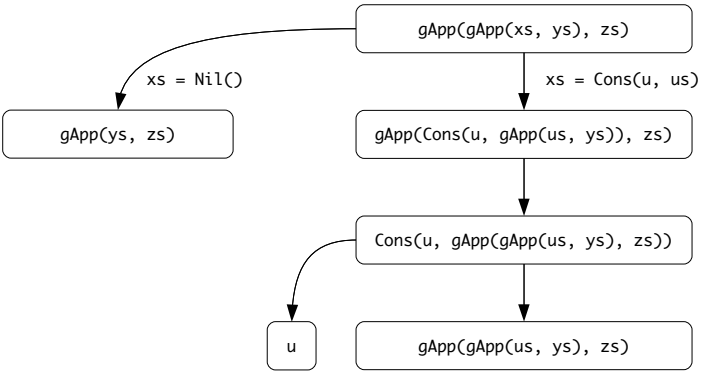
Suppose we need to concatenate three lists. This can be achieved by evaluating the expression `gApp(gApp(xs, ys), zs)`. However, this involves an inefficiency: the list xs is traversed twice (by the inner call to `gApp`, and by the outer one). By supercomipiling this program, we can obtain a more efficient program. Let us try to evaluate the expression (*configuration*) `gApp(gApp(xs, ys), zs)` "symbolically". First of all, we unfold the inner call `gApp(xs, ys)`. Since `gApp` analyzes the structure of the first argument, we have to consider two cases (or using the jargon of supercompilation - to *split* the configuration). In the first case `xs = Nil()`, and in the second case `xs = Cons(u, us)`, where `u` and `us` are fresh variables. We get:



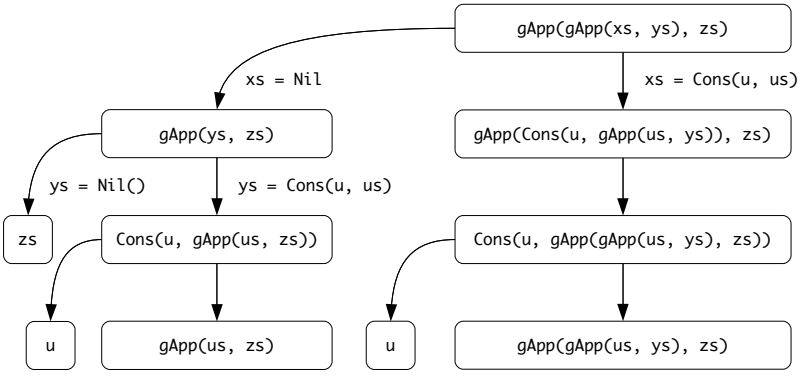
Now, the outer call to `gApp` in the right configuration can be unfolded in unambiguous way:



The rightmost configuration has a constructor at the outermost level - we can do nothing useful with it, so we continue by symbolically evaluating its arguments:



The rightmost configuration `gApp(gApp(us, ys), zs)` is the same as the root configuration (up to variable renaming). It is clear that further evaluating this configuration makes little sense, because it would produce the same result again. Thus we continue by unfolding the leftmost configuration `gApp(ys, zs)`. The result of two unfolding steps is as follows:



Now the construction of the configuration tree terminates: each leaf contains either a variable, a nullary constructor, or an already processed configuration (up to variable renaming). One can say that the tree thus constructed represents all possible ways of evaluating the root expression (configuration), which correspond to different values of the variables appearing in the root configuration.

To construct a program from a given tree, we generate a function definition for each node `alpha` with child node `beta`. The left and right sides of the definition are derived from `alpha` and `beta` respectively. Particularly, in this example we rename the expression `gApp(gApp(xs, ys), zs)` into `gApp1(xs, ys, zs)` and get the following:

```
gApp1(Nil(), y, z) = gApp2(y, z);
gApp1(Cons(v1, v2), y, z) = Cons(v1, gApp1(v2, y, z));
gApp2(Nil(), z) = z;
gApp2(Cons(v3, v4), z) = Cons(v3, gApp2(v4, z));
```

It can be easily seen that the evaluation of `App1(xs, ys, zs)` results in the list xs being traversed only once.

To sum up, the transformation has involved 3 procedures: symbolic computation, the search for regularities, and constructing the residual program from a tree. Symbolic computation was used for adding children to the current leaf. The search for regularities prevented the unfolding of already encountered configurations. And finally, when the tree had stopped to grow, a new term and a corresponding residual program were generated.

3 SLL: The Input Language of the Supercompiler

Our supercompiler deals with programs written in a simple first-order functional language (SLL). We suppose that there are countably infinite sets of symbols representing variables x ∈ X, constructors c ∈ C and function names f∈F and g∈G. All symbols have fixed arity. SLL-programs deal with data represented by (possibly infinite) trees built by means of constructors. The syntax of SLL is shown in Fig. 1. A program is a sequence of function definitions. All functions are divided into two classes: *f-functions* and *g-functions*[2]. The definition of a g-function is a sequence of one or more rules, each rule performing *pattern matching* on its first argument. The definition of an f-function is a single rule whose arguments are just variables, so that an f-function performs no analysis of its input data.

p	$::=$	$d_1 \ \ldots \ d_n$	program
d	$::=$	$f(x_1, \ \ldots \ , x_n) = \mathrm{e}\,;$	f-function
	$\mid$	$g(q_1, \ x_1, \ \ldots \ , x_n) = e_1\,;$	g-function
		$\ldots$	
		$g(q_m, \ x_1, \ \ldots \ , x_n) = e_m\,;$	
e	$::=$	x	variable
	$\mid$	$c(e_1, \ \ldots \ , e_n)$	constructor
	$\mid$	$f(e_1, \ \ldots \ , e_n)$	call to f-function
	$\mid$	$g(e_1, \ \ldots \ , e_n)$	call to g-function
q	$::=$	$c \ v_1 \ldots v_n$	pattern

Fig. 1. SLL Grammar

To simplify our presentation, variable names are required to start with a lowercase letter, constructor names with an uppercase letter, and the names of f- and g-functions with `f` and `g` respectively.

The intended operational semantics of the language is normal-order graph reduction to weak head normal form. Thus, essentially, SLL is a small subset of Haskell.

The SLL program considered in the previous section deals with lists built with the constructors `Nil` and `Cons`, and consists of a single rule defining the g-function `gApp` concatenating two lists.

4 Abstract Syntax

The implementation of the SLL abstract syntax in the language Scala is presented in Fig. 2. The first lines resemble Java, however, there are some differences.

The abstract class `Term`, which is used for representing SLL expressions, is declared first. Since the class body is empty, it can be omitted in Scala (but not in Java).

The concrete class `Var` is a subclass of `Term` and represents SLL variables. In Scala, every concrete class has a primary constructor, whose arguments are declared right after the class name. These arguments automatically become members of the class. So, since the constructor `Var` has the string argument `name`, `name` becomes a class member. The keyword `case` in a class definition indicates that all instances of the class can be matched against *patterns*. For instance, if the value of `t` is an instance of the class `Var`, then the contents of the field `name` is printed, otherwise nothing is done:

```
t match { case Var(name) => print(name); case _ => }
```

Scala has two kinds of variables: `vals` and `vars`. A `val` is similar to a final variable in Java. Once initialized, a val can never be reassigned. A `var`, by contrast, is similar to a non-final variable in Java. A `var` can be reassigned throughout its lifetime. Methods are declared using the keyword `def`.

Constructors, f-calls and g-calls are arranged in the same way - they have a name and arguments. So it is natural to represent them by a single class with an extra parameter denoting the term kind. The possible term kinds are enumerated in the `object TKind` (in the way similar to Java `enum`). The keyword `object` declares a *singleton object*. "If you are a Java programmer, one way to think of singleton objects is as the home for any static methods and values you might have written in Java."[1]

The class `CFG` is meant for representing these terms, the field `kind` being the term's kind, the field `name` being the term's name, and the field `args` being the list of the term's arguments. The type of `args` is declared as `List[Term]`, which means "a list of objects of the class Term". Note that in Scala square brackets are used for specifying type parameters, rather than array elements. (And in Java type parameters are written in angular brackets).

```
abstract class Term
case class Var(name: String) extends Term
case class CFG(kind: TKind.Value, name: String, args: List[Term])
 extends Term {
  def replaceArgs(newArgs: List[Term]) = CFG(kind, name, newArgs)
}
case class Let(term: Term, bindings: List[(Var, Term)]) extends Term
case class Pat(name: String, args: List[Var])

object TKind extends Enumeration {val Ctr, FCall, GCall = Value}

class CFGObject(kind: TKind.Value) extends ((String, List[Term]) => CFG) {
  def apply(name: String, args: List[Term]) = CFG(kind, name, args)
  def unapply(e: CFG) = if (e.kind == kind) Some(e.name, e.args) else None
}
object Ctr extends CFGObject(TKind.Ctr)
object FCall extends CFGObject(TKind.FCall)
object GCall extends CFGObject(TKind.GCall)

abstract class Def {def name: String}
case class FFun(name: String, args: List[Var], term: Term) extends Def
case class GFun(name: String, p: Pat, args: List[Var], term: Term) extends Def

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}
}
```

Fig. 2. SLanguage.scala

To simplify manipulations with terms we define *functional objects* `Ctr`, `FCall`, `GCall`. A method `apply` is only used to mimick a constructor: `c = Ctr(n, ts)` expands to `c = Ctr.apply(n, ts)`. A method named `unapply` is used for generalized pattern-matching yielding an *extractor*: the pattern `case Ctr(n, ts)` will cause an invocation of `Ctr.unapply`. `Some(n, args)` corresponds to a successful match, while `None` is a failure.

In addition to the list of definitions `defs` (declared in the constructor), the class `Program` contains three value fields: `f`, `g`, and `gs`. `f` is a map, where key is a name of an f-function and a value is the f-function itself. `g` is a map which uses a pair (a g-function name, a pattern name) as a key and the corresponding g-function as a value. `gs` is a map with a g-function name as a key and a list of g-functions as a value. In Scala, a piece of code appearing inside a class, but outside the method definitions, is considered to be a part of the constructor body. So, in the class Program, the maps `f`, `g` and `gs` are initialized when the constructor is called. To populate the map `f`, we traverse the list of function definitions. If an f-function is encountered then it is added to `f`. Note that this can be written in Scala in an elegant way by means of higher-order functions. We use the method `:\` of the class List (where `:\` is an alias for `foldRight`) possessing the following property:

```
([a0, a1, a2 .. an] :\ z) op = op(a0, op(a1, op(..., op(an, z) ...)))
```

In this equation, `op` is a function, which, in Scala, can be written in-line as a construction of the form `{case p1 => e1; ... case pN => eN;}`, representing an anonymous function performing pattern matching on its argument. The logic is as follows: we start with the empty map and traverse the function definition list. If the current element is an f-function, we add it to the map, otherwise the map remains unchanged. In this case, the operation `op` is a function that takes as input a pair: the current function definition and the map accumulating function definitions. The maps `g` and `gs` are built in a similar manner.

Maps in Scala (along with lists) are "genuine" functions: getting the value associated with a key `k` in a map `d` is written as `d(k)`:

```
val f: FFun = program.f("fApp2") // f-functiion
val g: GFun = program.g("gApp", "Nil") // g-funtion corresponding to pattern
val gs: List[GFun] = program.gs("gApp") // list of g-functions
```

So we have defined the abstract syntax of SLL in functional style: all syntax objects are immutable values.

Let-expressions do not correspond to any SLL construct: they are generated by the supercompiler SPSC and will be described later.

5 Term Algebra

A substitution is a list of pairs `(v, t)`, where `v` is a variable and `t` is a term. A substitution can be conveniently represented in Scala by a map of the type `Map[Var, Term]`.

Informally, the result of applying a substitution to a term is defined as follows: find all the variables in the term that belong to the domain of the substitution and replace them by their values. This is implemented in the method `subst` in Fig. 3. The method `findSubst` takes two terms `t1` and `t2` and tries to find a substitution `m` such that `subst(t1, m) = t2`. If there exists such a substitution, then `t2` is an *instance* of `t1`. The idea is as follows: we start with the empty substitution and populate it by simultaneously traversing both terms `t1` and `t2`. This is implemented in the method `walk`. The method `inst` checks whether the second term is an instance of the first one. The method `equiv` checks whether two terms are equivalent up to variable renaming. Here we make extensive use of the higher-level methods of Scala collections: `map`, `/:` (an alias for `foldLeft`), `filter`, `forall`, and local partial functions. The underscore is used for creating curried versions of functions: `subst(_, m)` is equivalent to `{(x) => subst(x, m)}`.

6 Process Tree

The SLL interpreter may be seen as a machine simplifying a given expression step-by-step according to a set of rules that specify the semantics of the language. It should be noted that the standard interpreter is only capable of simplifying the terms that are *ground*, i.e. contain no variables. In this case the rule to be applied is determined unambiguously.

```
object Algebra {
  def subst(term: Term, m: Map[Var, Term]): Term = term match {
    case v: Var     => m.getOrElse(v, v)
    case e: CFG => e.replaceArgs(e.args.map(subst(_, m)))
  }

  def equiv(t1: Term, t2: Term): Boolean = inst(t1, t2) && inst(t2, t1)
  def inst(t1: Term, t2: Term): Boolean = findSubst(t1, t2) != null
  def shallowEq(e1: CFG, e2: CFG) = e1.kind == e2.kind && e1.name == e2.name

  def findSubst(t1: Term, t2: Term): Map[Var, Term] = {
    val map = scala.collection.mutable.Map[Var, Term]()
    def walk(t1: Term, t2: Term): Boolean = (t1, t2) match {
      case (v1: Var, _) => map.getOrElse(v1, t2) == (map+(v1 -> t2))(v1)
      case (e1: CFG, e2:CFG) if shallowEq(e1, e2) =>
        List.forall2(e1.args, e2.args)(walk)
      case _ => false
    }
    if (walk(t1, t2)) Map(map.toList:_*).filter{case (k, v) => k != v} else null
  }

  def vars(t: Term): List[Var] = t match {
    case v: Var   => (List(v))
    case e: CFG => (List[Var]() /: e.args) {_ union vars(_)}
  }

  def trivial(expr: Term): Boolean = expr match {
    case FCall(_, _) => false
    case GCall(_, _) => false
    case _ => true
  }
}
```

Fig. 3. Algebra.scala

In the case of supercompilation, an expression to be transformed (a configuration) may contain variables, in which case it "symbolically" represents a set of ground expressions. Thus, there may arise situations in which an expression cannot be reduced by applying a single reduction rule, and a case analysis is needed.

In particular, if the first argument of a g-function call is a variable `x`, and the g-function definition contains N rules, we have to consider N cases, each case corresponding to a rule in the g-function definition. If a rule has the form `g(c(x1,...,xM),...)=e`, we replace `x` in the configuration with `c(x1,...,xM)`, so that the rule becomes applicable, and a reduction step can be performed by unfolding the call to `g`.

A "process tree" is an oriented tree with a start configuration placed in the root, and is built as a result of symbolic computation (or metacomputation [5]) described above. If a node has been produced by the application of a g-rule, the arrow connecting the parent and the child nodes is assigned the *contraction* of the form `x = c(x1,...,xM)` where c is the constructor from the left hand side of the g-rule.

We use 3 classes for representing process trees (see Fig. 4), the names of classes being self-explanatory.

```
import Algebra._

case class Contraction(v: Var, pat: Pat)

class Node(val expr: Term, val parent: Node, val contr: Contraction) {
  def ancestors: List[Node] =
    if (parent == null) Nil else parent :: parent.ancestors
  def fnode =
    ancestors.find{n => !trivial(n.expr) && equiv(expr, n.expr)}.getOrElse(null)

  def isProcessed = expr match {
    case Ctr(_, Nil) => true
    case v: Var => true
    case _ => fnode != null
  }
}

class Tree(val root: Node, val children: Map[Node, List[Node]]) {
  def addChildren(n: Node, cs: List[(Term, Contraction)]) =
    new Tree(root, children + (n -> (cs map {case (t, b) => new Node(t, n, b)})))

  def replace(n: Node, exp: Term) =
    if (n == root) new Tree(n, Map().withDefaultValue(Nil))
    else {
      val p = n.parent
      val cs = children(p) map {m => if (m == n) new Node(exp, p, n.contr) else m}
      new Tree(root, children + (n -> cs))
    }

  def leaves_(node: Node): List[Node] =
    if (children(node).isEmpty) List(node)
    else List.flatten(children(node) map leaves_)
  def leaves() = leaves_(root)
}
```

Fig. 4. ProcessTree.scala

The class `Tree` has a number of utility methods. `leaves` returns a list of the tree's leaves. `replace` returns a new tree by replacing the expression in a node and deleting the subtrees under the node (if any). `addChildren` takes a list of terms, turns each term into a node, and returns a new tree with the nodes thus obtained added as children to the node in question. The other members of the classes are described in the following sections.

7 Supercompiler: the First Look

In a general case, the process tree constructed by metacomputation will be infinite. Supercompilation is based on the idea that, for a given source program, the construction of the process tree should eventually produce a configuration `conf2` that is similar to an earlier configuration `conf1`. For example, suppose that `conf2` is identical to `conf1` modulo variable renaming. It means that the evaluation of `conf2` will produce the same subtree as the evaluation of `conf1`. So it seems reasonable, to just add an arrow from `conf2` to `conf1` to the tree, thereby turning the tree into a graph. The node labeled with `conf1` is called a *functional node* (or a base node) and the node labeled with `conf2` a *repeat node*.

```
import Algebra._
class BaseSuperCompiler(p: Program){
  def driveExp(expr: Term): List[(Term, Contraction)] = expr match {
    case Ctr(name, args) => args.map((_,null))
    case FCall(name, args)  =>
      List((subst(p.f(name).term, Map(p.f(name).args.zip(args): _*)), null))
    case GCall(name, Ctr(cname, cargs) :: args) =>
      val g = p.g(name, cname)
      List((subst(g.term, Map((g.p.args:::g.args) zip (cargs:::args): _*)), null))
    case gCall @ GCall(name, (v : Var) :: args) =>
      for (g <- p.gs(name); fp = freshPat(g.p); cons = Ctr(fp.name, fp.args))
        yield driveExp(subst(gCall, Map(v -> cons))) match
          {case (k, _) :: _ => (k, Contraction(v, fp))}
    case GCall(name, args) =>
      driveExp(args(0)) map {case (k, v) => (GCall(name, k :: args.tail), v)}
    case Let(term, bs) => (term, null) :: bs.map {case (_, v) => (v, null)}
  }

  def buildProcessTree(e: Term): Tree = {
    var t = new Tree(new Node(e, null, null), Map().withDefaultValue(Nil))
    while (t.leaves.exists{!_.isProcessed}) {
      val b = t.leaves.find(!_.isProcessed).get
      t = b.ancestors.find(a => !trivial(a.expr) && inst(a.expr, b.expr)) match {
       case Some(a) => t.replace(b, Let(b.expr, findSubst(b.expr, a.expr).toList))
       case None => t.addChildren(b, driveExp(b.expr))
      }
    }
    t
  }

  def freshPat(p: Pat) = Pat(p.name, p.args map freshVar)
}
```

Fig. 5. BaseSuperCompiler.scala

In a more complicated case, `conf2` is not the same as `conf1`, but there exists a substitution `theta` such that `conf2 = subst(conf1, theta)`. If so, the supercompiler can transform `conf2` into a configuration containing `conf1`s as a constituent part by making use of the let-construct. And again this enables a loop in the graph to be created.

The partial process tree, produced by symbolic computation (also known as *driving* [3,4,5] in the world of supercompilation) contains enough information for the residual program (equivalent to the input one) to be constructed.

Now we describe an algorithm constructing the partial process tree. The method `fnode` in the class `Node` finds a corresponding functional node (if any).

A term is *trivial*, if it is either a constructor, a variable, or a let-expression - see corresponding method in `object Algebra` in Fig. 3. A node is called processed, if it is either a repeat node, a node labeled by a variable, a node labeled by a nullary constructor, or a repeat node - the method `isProcessed` in the class `Node` implements this logic.

The construction of the partial process tree starts with labeling the root node with the initial expression. If all leaves have already been processed, then the supercompilation is complete. Otherwise let `b` be an unprocessed node:

– If `b` is trivial, then drive the configuration in `b`.

```
import Algebra._

class ResidualProgramGenerator(val tree: Tree) {
  private val sigs = scala.collection.mutable.Map[Node, (String, List[Var])]()
  private val defs = new scala.collection.mutable.ListBuffer[Def]
  lazy val result = (walk(tree.root), Program(defs.toList))

  private def walk(n: Node): Term = if (n.fnode == null) n.expr match {
    case v: Var => v
    case Let(_,bs) => subst(walk(tree.children(n).head),
         Map(bs map {case (k, _) => k} zip (tree.children(n).tail map walk):_*))
    case Ctr(name, _) => Ctr(name, tree.children(n).map(walk))
    case FCall(name, args) => walkCall(n, name, args)
    case GCall(name, args) => walkCall(n, name, args)
  } else sigs(n.fnode) match {
    case (name, args) =>
      if (tree.children(n.fnode).head.contr == null)
           subst(FCall(name, args), findSubst(n.fnode.expr, n.expr))
      else subst(GCall(name, args), findSubst(n.fnode.expr, n.expr))
  }

  def walkCall(n: Node, name: String, args: List[Term]) = {
    val vs = vars(n.expr)
    if (tree.children(n).head.contr != null) {
      val (gname, _) = sigs.getOrElseUpdate(n, (rename(name, "g"), vs))
      for (cn <- tree.children(n))
        defs += GFun(gname, cn.contr.pat, vs.tail, walk(cn))
      GCall(gname, vs)
    } else if (tree.leaves.exists(_.fnode == n)) {
      val (fname, fargs) = sigs.getOrElseUpdate(n, (rename(name, "f"), vs))
      defs += FFun(fname, fargs, walk(tree.children(n).head))
      FCall(fname, vs)
    } else walk(tree.children(n).head)
  }

  def rename(f: String, b: String) = {b + f.drop(1) + (sigs.size + 1)}
}
```

Fig. 6. ResidualProgramGenerator.scala

- If, among the ancestors of `b`, there is a node `a`, such that `b` is an instance of `a`, then find a corresponding substitution and replace the expression in `b` by a let-expression.
- Otherwise, drive the configuration in **b**.

The most interesting case is driving a call to a g-function, when the first argument is a variable. If so, the configuration is split up according to the patterns in the g-function definition.

The composite patterns, like `GCall(name, Ctr(n, cargs) :: args)` are very useful for the implementation of driving. Scala also enables the names of methods and constructors to be used as binary operators, for instance `::` is a List constructor, which can be used as either `::(head, tail)` or `head::tail`.

8 Residual Program Generator

The process of extracting the residual program from a process tree is conceptually simple. The initial configuration produces a function definition. Each cycle

produces a definition of a recursive function. Each splitting of configuration produces a g-function definition.

The method `walk` traverses a partial process tree in top-down order. Visiting a functional node results in a function definition being generated. In the case of a configuration splitting, a g-function is generated, otherwise an f-function. A function definition consists of a signature (the name and the arguments of the function) and a function body. The function's arity depends on the number of variables in the given configuration. The signature is generated from a functional node, and the recursive function call is generated from the repeat node containing a reference to the functional node. Visiting a node labeled by a constructor, results in the corresponding constructor being generated, whose arguments are produced by visiting the child nodes. Otherwise the result of visiting a node is obtained by visiting its (single) child node. It should be noted that the names of generated functions are syntactically correct: the names of f-functions start with `f` and the names of g-functions with `g`. Residual Program Generator is shown in Fig. 6. Its `result` is a pair - a new expression and a new program.

9 An Example

Let us consider again the program that concatenates lists. In order to supercompile the term `gApp(gApp(xs, ys), zs)`, the supercompiler SPSC can be invoked in the following way (where triple quotes are used for representing multi-line strings):

```
val code = """gApp(Nil(), vs) = vs;
              gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));"""
val sc = new BaseSuperCompiler(SParsers.parseProg(code))
val pt = sc.buildProcessTree(SParsers.parseTerm("gApp(gApp(xs, ys), zs)"))
val (resTerm, resProgram) = new ResidualProgramGenerator(pt).result
println(resTerm)
println(resProgram)
```

The result of supercompilation is as follows.

```
gApp1(xs, ys, zs)
gApp1(Nil(), y, z) = gApp2(y, z);
gApp1(Cons(v1, v2), y, z) = Cons(v1, gApp1(v2, y, z));
gApp2(Nil(), z) = z;
gApp2(Cons(v3, v4), z) = Cons(v3, gApp2(v4, z));
```

The list `xs` in the residual program is traversed only once. The output contains two function definitions produced from the source function definition `gApp`: `gApp1` and `gApp2`. `gApp2` is isomorphic to `gApp`, while `gApp1` is a "specialized" version of `gApp`, which concatenates three lists. The object `SParsers` will be described later.

10 Whistle: Homeomorphic Embedding

The version of the supercompiler considered so far may never terminate in trying to produce an infinite process tree. To prevent the non-termination of that kind, a special technique can be used, which is known as *whistle* in the world

```
import Algebra._
object HE {
  def he_*(t1: Term, t2: Term): Boolean = he(t1, t2) && b(t1) == b(t2)
  def he(t1: Term, t2: Term) = heByDiving(t1, t2) || heByCoupling(t1, t2)

  private def heByDiving(t1: Term, t2: Term): Boolean = t2 match {
    case e: CFG => e.args exists (he(t1, _))
    case _ => false
  }

  private def heByCoupling(t1: Term, t2: Term): Boolean = (t1, t2) match {
    case (x: CFG, y: CFG) if shallowEq(x, y) => List.forall2(x.args, y.args)(he)
    case (Var(_), Var(_)) => true
    case _ => false
  }

  private def b(t: Term): Int = t match {
    case GCall(_, args) => b(args.head)
    case Var(_) => 1
    case _ => 0
  }
}
```

Fig. 7. HE.scala

of supercompilation. This technique is based on the use of the homeomorphic embedding relation $\triangleleft$. Informally speaking, a configuration `a` is embedded into a configuration `b`, if `b` can be transformed into `a` by erasing some parts of `b`.

Here are a few examples:

```
       b ◁ f(b)
    c(b) ◁ c(f(b))
 c(b, b) ◁ c(f(b), f(b))
```

Some non-examples of homeomorphic embedding are:

```
f(c(b)) ◁̸ c(b)
f(c(b)) ◁̸ c(f(b))
f(c(b)) ◁̸ f(f(f(b)))
```

More formally, the homeomorphic embedding relation is defined by the rules:

$$v_1 \triangleleft v_2$$
$$\exists i \in \{1, \ldots, n\}: e \triangleleft e_i \Rightarrow e \triangleleft \mathtt{h}(e_1, \ldots, e_n)$$
$$\forall i \in \{1, \ldots, n\}: e_i \triangleleft e'_i \Rightarrow \mathtt{h}(e_1, \ldots, e_n) \triangleleft \mathtt{h}(e'_1, \ldots, e'_1)$$

Here `h` denotes either a constructor or a function call.

We will use a slightly refined version of the homeomorphic embedding relation, classifying all expressions according to the value of the following characteristic function `B`. The additional requirement is that two expressions `e1` and `e2` are tested for embedding only if they belong to the same class, i.e. `B(e1) = B(e2)`.

$$\mathtt{B}(\mathtt{g}(e_0, e_1, \ldots, e_m)) = \mathtt{B}(e_0)$$
$$\mathtt{B}(\mathtt{f}(e_1, \ldots, e_m)) = 0$$
$$\mathtt{B}(\mathtt{c}(e_1, \ldots, e_m)) = 0$$
$$\mathtt{B}(\mathtt{x}) = 1$$

The method `he_*` in Fig. 7 implements the extended homeomorphic embedding relation.

```
import Algebra._
case class Gen(t: Term, m1: Map[Var, Term], m2: Map[Var, Term])
object MSG {
  def msg(t1: Term, t2: Term): Gen = {
    val v = freshVar()
    var g = Gen(v, Map(v -> t1), Map(v -> t2))
    var exp = g.t
    do {exp = g.t; g = commonSubst(commonFun(g))} while (exp != g.t)
    g
  }

  def commonFun(g: Gen): Gen = {
    for (v <- g.m1.keys) (g.m1(v), g.m2(v)) match {
      case (e1: CFG, e2: CFG) if shallowEq(e1, e2) =>
       val vs = e1.args map freshVar
       val t = subst(g.t, Map(v -> e1.replaceArgs(vs)))
       return Gen(t, g.m1 - v ++ vs.zip(e1.args), g.m2 - v ++ vs.zip(e2.args))
      case _ =>
    }
    g
  }

  def commonSubst(gen: Gen): Gen = {
    for ((v1, e1) <- gen.m1; (v2, e2) <- gen.m1)
      if ((v1 != v2 && e1 == e2) && (gen.m2(v1) == gen.m2(v2)))
        return Gen(subst(gen.t, Map(v1 -> v2)), gen.m1 - v1, gen.m2 - v1)
    gen
  }
}
```

Fig. 8. MSG.scala

11 Generalization

A generalization of two configurations `t1` and `t2` is a triple `(t, s1, s2)`, where `t` is a configuration, and `s1`, `s2` are substitutions, such that:

```
subst(t, s1) == t1 && subst(t, s2) == s2
```

The most specific generalization of expressions `t1` and `t2` is a generalization `(t, s1, s2)`, such that for every generalization `(t', s1', s2')`, `t` is an instance of `t'`.

The most specific generalization of two expressions `t1` and `t2` can be found by exhaustively applying the following rewrite rules to the initial trivial generalization `(v, {v:=t1}, {v:=t2})`:

$$\begin{pmatrix} e \\ \{v := h(e'_1,\ \ldots,\ e'_n)\} \cup \theta' \\ \{v := h(e''_1,\ \ldots,\ e''_n)\} \cup \theta'' \end{pmatrix} \Rightarrow \begin{pmatrix} subst(e,\ \{v := h(v_1,\ \ldots,\ v_n)\}) \\ \{v_1 := e'_1,\ \ldots,\ v_n := e'_n\} \cup \theta' \\ \{v_1 := e''_1,\ \ldots,\ v_n := e''_n\} \cup \theta'' \end{pmatrix}$$

$$\begin{pmatrix} e \\ \{v_1 := e'_1,\ v_2 := e'_2\} \cup \theta' \\ \{v_1 := e''_1,\ v_2 := e''_2\} \cup \theta'' \end{pmatrix} \Rightarrow \begin{pmatrix} subst(e,\ \{v_1 := v_2\}) \\ \{v_2 := e'_2\} \cup \theta' \\ \{v_2 := e''_2\} \cup \theta'' \end{pmatrix}$$

An algorithm computing MSG is shown in Fig. 8.

```
import Algebra._

class SuperCompiler(p: Program) extends BaseSuperCompiler(p){

  override def buildProcessTree(e: Term): Tree = {
    var t = new Tree(new Node(e, null, null), Map().withDefaultValue(Nil))
    while (t.leaves.exists{!_.isProcessed}) {
      val b = t.leaves.find(!_.isProcessed).get
      t = if (trivial(b.expr)) {
        t.addChildren(b, driveExp(b.expr)) //drive
      } else {
        b.ancestors.find(a => !trivial(a.expr) && HE.he_*(a.expr, b.expr)) match {
          case Some(a) => {
            if (inst(a.expr, b.expr)) abs(t, b, a)
            else if (equiv(MSG.msg(a.expr, b.expr).t, Var("z"))) split(t, b)
            else abs(t, a, b)
          }
          case None => t.addChildren(b, driveExp(b.expr)) // drive
        }}}
    t
  }

  def abs(t: Tree, a: Node, b: Node) =
    ((g: Gen) => t.replace(a, Let(g.t, g.m1.toList))) (MSG.msg(a.expr, b.expr))

  def split(t: Tree, n: Node) : Tree = n.expr match {
    case e : CFG =>
      val vs = e.args map freshVar
      t.replace(n, Let(e.replaceArgs(vs), vs zip e.args))
    }
}
```

Fig. 9. SuperCompiler.scala

12 Supercompiler Revisited

Now we are ready to "patch" our supercompiler by adding the whistle and generalization. Here is a revised algorithm constructing the partial process tree.

The construction of partial process tree starts with labeling the root node with the initial expression. If all leaves are processed, then the supercompilation is complete. Otherwise, let `b` be an unprocessed node:

- If `b` is trivial, then drive the configuration in `b`.
- If, among the ancestors of `b`, there is a node `a`, such that `a.expr` $\triangleleft$ `b.expr`:
  - If `b.expr` is an instance of `a.expr`, then find a corresponding substitution and replace the expression in `b` by a let-expression.
  - If the most specific generalization of `a.expr` and `b.expr` is a variable, then split `a.expr`.
  - Otherwise, generalize `a.expr`.
- Otherwise, drive the configuration in `b`.

The revised supercompiler is presented in the Fig. 9.

13 One More Example

Consider the following initial configuration `gApp(gApp(x, y), x)`. Without the whistle and generalization, the supercompilation would never terminate. This is an example of the accumulating side effect. The progressively larger terms would be encountered again and again:

```
gApp(gApp(x, y), x) ◁ gApp(gApp(us, y), Cons(u, us))
  ◁ gApp(gApp(vs, y), Cons(u, Cons (v, vs))))
```

`BaseSuperCompiler` will not terminate on this input:

```
val code = """gApp(Nil(), vs) = vs;
              gApp(Cons(u, us), vs) = Cons(u, gApp(us, vs));"""
val sc = new SuperCompiler(SParsers.parseProg(code))
val pt = sc.buildProcessTree(SParsers.parseTerm("gApp(gApp(x, y), x)"))
val (resTerm, resProgram) = new ResidualProgramGenerator(pt).result
println(resTerm)
println(resProgram)
```

However, with the whistle we can detect these growing terms. So the initial configuration is generalized and the result of supercompilation is as follows:

```
gApp1(x, y, x)
gApp1(Nil(), y, z) = gApp2(y, z);
gApp1(Cons(v1, v2), y, z) = Cons(v1, gApp1(v2, y, z));
gApp2(Nil(), z) = z;
gApp2(Cons(v3, v4), z) = Cons(v3, gApp2(v4, z));
```

14 SLL Parser

This section describes a parser transforming an SLL-program in the concrete syntax (as a character sequence) into the corresponding SLL-program represented by an abstract syntax tree.

This stuff is not essential for understanding supercompilation, however it has been included in order to make the SPSC source code complete and ready-to use. Besides, this section illustrates some powerful features of the Scala programming language.

The standard Scala library provides a package for writing combinator parsers. The idea of combinator parsing is to construct sophisticated parsers by composing elementary parsers by means of combinators, which are functions and operators defined in Scala that take parsers as input and produce new parsers. More specifically, parsers are implemented in Scala as classes making use of some advanced object-oriented features. Namely, a concrete parser is a subclass of the abstract class defined in the library:

```
abstract class Parser[+T] extends (Input => ParseResult[T]) {...}
```

Since the class `Parser` has a functional type `Input => ParseResult[T]`, the objects of this class are genuine functions. On the other hand, since all functions in Scala are objects, they can provide additional fields and methods.

Thus, parsers are functions from `Input` to `ParseResult[T]`. In our case `Input` is a token stream (constructed by the standard library lexer from a character stream) and `T` corresponds to different classes that represent SLL abstract

```
import scala.util.parsing.combinator.ImplicitConversions
import scala.util.parsing.combinator.syntactical.StandardTokenParsers
import scala.util.parsing.input.{CharSequenceReader => Reader}

object SParsers extends StandardTokenParsers with ImplicitConversions {
  lexical.delimiters += ("(", ")", ",", "=", ";")
  def prog = definition+
  def definition: Parser[Def] = gFun | fFun
  def term: Parser[Term] = fcall | gcall | ctr | vrb
  def uid = ident ^? {case id if id.charAt(0).isUpperCase => id}
  def lid = ident ^? {case id if id.charAt(0).isLowerCase => id}
  def fid = ident ^? {case id if id.charAt(0) == 'f' => id}
  def gid = ident ^? {case id if id.charAt(0) == 'g' => id}
  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
  def parseProg(s: String) = Program(prog(new lexical.Scanner(new Reader(s))).get)
  def parseTerm(s: String) = term(new lexical.Scanner(new Reader(s))).get
}
```

Fig. 10. SParsers.scala

syntax trees. `ParseResult[T]` is an abstract class with subclasses `Success[T]` and `Failure`. The class `StandardTokenParsers` provides a few basic parsers that transform text into a stream of tokens, each token corresponding to an identifier, an operator, a keyword, etc.

Now we need to construct an abstract syntax tree from the token stream. This can be done by making use of the standard library of basic parsers and parser combinators. We will use the following combinators:

- `x | y` - a parser combinator for alternative composition
- `x ~ y` - a parser combinator for sequential composition
- `x*` - returns a parser that accepts x zero or more times
- `x ^^ f` - a parser combinator for function application
- `x ^? f` - a parser combinator for partial function application
- `x ~> x` - a parser combinator for sequential composition which keeps only the right result
- `x <~ y` - a parser combinator for sequential composition which keeps only the left result
- `repsep(x, del)` - a parser generator for interleaved repetitions

Since Scala allows the method and operator names to contain non-alphanumeric characters, and permits the method names to be used as infix operators, a parser written in Scala looks like the standard BNF-notation.

First, we supply the `lexer` with the information about the delimiters of our language. Then we define our custom building blocks: parser `fid` corresponds to an identifier starting with the letter `f`. Here we use the standard parser `ident` which matches a sequence of letters and digits starting with a letter. We define

a new parser using the combinator `^?`. Note that a sequence of case expressions in Scala is actually a partial function (defined for the arguments that match one of the patterns). The following parsers are defined in a similar manner: `gid, uid` and `lid` parsers that match identifiers starting with the letter `g`, an upper-case letter and a lower case letter, respectively. Other parsers are self-explanatory. Parser `vrb` transforms a string into a `Var` (object of SLL abstract syntax), etc. Note that a Scala constructor can be considered as a function of the corresponding arity.

SPSC uses the parser as a "black box", calling the following methods:

- `parseProg: String => Program`, which transforms a (correct) text representation of SLL program into an object of class Program
- `parseTerm: String => Term`, which transforms a (correct) text representation of SLL term into an abstract syntax tree

15 Conclusions

In this paper we have described a simple supercompiler for a lazy first-order functional language. The main focus was on providing the complete and ready-to-use sources of the supercompiler.

To reduce the size of the paper, we have presented a slightly simplified version of SPSC. In particular, the input programs are supposed to be correct, so that all checks concerning context-dependent restrictions have been removed. A more advanced version of SPSC can be found at the project page: `http://spsc.appspot.com`

References

1. M. Odersky, L. Spoon and B. Venners. Programming in Scala: A Comprehensive Step-by-step Guide. Artima Inc. 2008.
2. M.H. Sørensen. Convergence of program transformers in the metric space of trees. In Mathematics of Program Construction, volume 1422 of Lecture Notes in Computer Science. Springer, 1998.
3. V. F. Turchin. The Language Refal: The Theory of Compilation and Metasystem Analysis. Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.
4. V. F. Turchin. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems (TOPLAS), 8(3):292325, 1986.
5. S.M. Abramov and L.V. Parmenova. Metacomputations and their application. Supercompilation (In Russian). Ailamazyan University of Pereslavl, 2006.
6. M.H. Sørensen and R. Glück. Introduction to Supercompilation. In Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, volume 1706 of Lecture Notes in Computer Science. Springer, 1998.
7. M.H. Sørensen, R. Glück and N. Jones. A positive supercompiler. Journal of Functional Programming, 6, 1996.
8. R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In Selected Papers from the International Seminar on Partial Evaluation, volume 1110 of Lecture Notes in Computer Science. Springer, 1996