Type-Safe Metaprogramming and Compilation Techniques for Optimizing High-Level Programs

Lionel Parreaux

in collaboration with

Amir Shaikhha
Antoine Voizard
Simon Peyton Jones
Christoph E. Koch

EPFL, December 2019
Type-Safe Metaprogramming and Compilation Techniques for Optimizing High-Level Programs
Clever Techniques
for Optimizing
High-Level Programs
What does this program do?

```scala
persons.filter(_.age > 7).groupBy(_.firstName)
```
What does this program do?

```scala
var cur = persons
val groups = mutable.Map.empty[String, mutable.Buffer[Person]]
while (cur.nonEmpty) {
  val p = cur.head
  cur = cur.tail
  if (p.age > 7) {
    var gpeOpt = groups.get(p.firstName)
    val gpe = gpeOpt match {
      case None =>
        var buf = mutable.Buffer.empty[Person]
        groups(p.firstName) = buf
        buf
      case Some(buf) =>
        buf
    }
    buf += p
  }
}
groups
```
Dilemma of the Modern Programmer

- High-level code better for maintainability & correctness
  Important for productivity, flexibility
- Low-level code better for performance

⇒ Programmers often write lower level code;
  harder to read, debug, evolve...

What if we did not have to choose?
  compilers *should* do the rewriting for us automatically

But compilers are not smart enough
Making high-level programs efficient

Alternative 1:

Make compilers smarter, better at functional code

⇒ new compilation techniques

Alternative 2:

Give users more control on how programs are compiled
safely teach the compiler how to optimize common patterns

⇒ new metaprogramming techniques
Two Philosophies

Compilation techniques

*not enough on their own*

Metaprogramming techniques

*not enough on their own*

*my work: synergy of advances*

in both approaches
Metaprogramming
Metaprogramming – Definition + Motivation

**Meta**programming =

writing programs that manipulate programs

Manipulate = generate, analyse, transform, evaluate

Ubiquitous: code generation, DSL implementation, compile-time macros, program analysis, optimization and compilation, boilerplate generation, multi-stage programming...

Hard to get right — reserved for experts

→ bad user experience, security vulnerabilities, etc.
Metaprogramming – Problem

Metaprogramming is often **error-prone**

Example in the wild: Scala 2 macros

useful for optimizing a library; e.g., parser combinators

Very hard to make a macro robust & to debug it

```
[error] found  : shapeless.::[Int,shapeless.::[scalatex.stages.Ast.Block,shapeless.HNil]]
[error] required: scalatex.stages.Ast.Block
[error]   new Parser(input, offset).Body.run().get
[error] ^
```

Metaprogramming errors are **not nice**
Squid: Type-Safe Metaprogramming & Compilation Framework
The Squid Framework

Squid – Scala Quoted DSLs  

Extension of Scala via compile-time macros

Type-safe metaprogramming via quasiquotes [POPL 2018]

Ensures metaprograms are free of runtime type errors:

no type mismatches; no unbound variables

Extensible optimizing compiler framework [Scala 2017]

⇒ both aspects work together [GPCE 2017, Best Paper]
The Squid Framework — Architecture

High-level, type-safe API

```
code"..."
mash Code[T, C]
rewrite analyse
run without
```

Lower-level, internal APIs

- Program generation
- Cross-stage persistence
- Code analysis
- Strategic rewriting
- Runtime compilation

Encapsulates

- CPS IR
  - rewrite engine, JIT, ...
  - API "interpreters"

- ANF IR
  - effect system, ...

- Graph IR
Type-Safe Metaprogramming: Quasiquote
Terminology

Quotation

“a group of words taken from a text”

Quasiquotation

“a group of $x$ taken from a text” where $x = \text{“words”}$

Quasiquotation in programming (e.g. Scala, Haskell, ...)

```
val x = "words" ; s"a group of $x$ taken from a text"
```

In this work: code quasiquotation for metaprogramming

```
val lsc = code"List(1,2,3)" ; code"$lsc.map(x => x + 1)"
```
Two Flavors of (code) Quasiquotes

**Analytic Quasiquotes**
Lisp, Scala, F#, Template Haskell…
Pioneered by Lisp – treat code as data and data as code
Limitation: metaprogramming is hard
code evaluation raises unbound variables errors; type mismatches…

```plaintext
code"max(x,true)".run  // runtime crash!
```

**Statically-Typed Quasiquotes**
MetaOCaml, F#, Template Haskell…
Appeared in context of multi-stage programming (MSP, or staging)
Static guarantee: generated programs are well-typed

```plaintext
code"max(x,0)"  // ill-typed  code"max(0,1)" : Code[Int]
code"(x:Int) => {{code"x".run; code"x"}}" : Code[Int => Int]
```

Limitation: purely-generative generated code is a black box!
Statically-typed quasiquotes, of type \( \text{Code}[\text{Type, Context}] \)

\[
\text{val } c0 = \text{code}(x: \text{Int}) \Rightarrow x \times 2 + 1 \\
c0 : \text{Code[Int => Int, Any]}
\]

Code evaluation

\[
\text{code}"c0(3)".run \rightarrow 7
\]

First-class variables

\[
\text{val } v = \text{Variable[Int]} \quad /\!/ \text{ of type } \text{Variable[Int]} \\
\text{val } c1 = \text{code}($v: \text{Int}) \Rightarrow \$v \times 2 + 1 \quad /\!/ \Rightarrow c0
\]
Free variables and contexts

Variable dependencies are statically tracked

```scala
val v = Variable[String]
val open = code"$v * 2 + 1" : Code[Int, v.Ctx]
open.run // type error: "Cannot prove that v.Ctx =:= Any"
```

( code"val $v = 3; $open" : Code[Int, Any] ).run → 7

Multiple free variables

```scala
val w = Variable[Int]
val cde = code"$v * $w" : Code[String, v.Ctx & w.Ctx]
```
Quasiquotes for Pattern Matching

```scala
val p = code"Some(2 + 2)"

Pattern matching

p match {
  case code"Some($v)" => v
}

→ code"2 + 2"
```
Application Example:
Polymorphic but Efficient Linear Algebra Library
Application: polymorphic linear algebra library

Many data analytics tasks boil down to linear algebra

But different problems use different representations/types

\[ f(x) = \begin{bmatrix} 0 & 0 & 1 & x + 3 \\ 8 & 0 & 2x - 1 & 6 \\ 0 & 3x + 3 & 0 & 0 \\ 0 & 0 & x & 0 \end{bmatrix}^2 = \begin{bmatrix} 0 & 3x + 3 & x^2 + 3x & 0 \\ 0 & 6x^2 + 3x - 3 & 6x + 8 & 8x + 24 \\ 24x + 24 & 2x + 3 & 6x^2 + 3x - 3 & 18x + 18 \\ 0 & 3x^2 + 3x & 0 & 0 \end{bmatrix} \]

\[ f'(x) = \begin{bmatrix} 0 & 3 & 2x + 3 & 0 \\ 0 & 12x + 3 & 6 & 8 \\ 24 & 0 & 12x + 3 & 18 \\ 0 & 6x + 3 & 0 & 0 \end{bmatrix} \]

\[ f(2) = \begin{bmatrix} 0 & 9 & 10 & 0 \\ 0 & 27 & 20 & 40 \\ 72 & 0 & 27 & 54 \\ 0 & 18 & 0 & 0 \end{bmatrix} \]

\[ f'(2) = \begin{bmatrix} 0 & 3 & 7 & 0 \\ 0 & 27 & 6 & 8 \\ 24 & 0 & 27 & 18 \\ 0 & 15 & 0 & 0 \end{bmatrix} \]
Application: polymorphic linear algebra library

In practice, each workload uses a specialized implementation → duplicated work

Idea: make a *maximally-polymorphic*, reusable library
And remove overhead using Squid metaprogramming

tagless-final + quoted code matching
Polymorphism actually helps generate and optimize programs!

⇒ PILATUS, Squid-based linear algebra library [ECOOP 2019]
Type-Safe Metaprogramming: Dealing with scopes/contexts
val p = code"(x:Int) => x * 2 + 1" : Code[Int => Int, Any]

Pattern matching

p match {
    case code"Some($v)" => ???
    case code"($y:Int) => $e + 1" => ...
    e : Code[Int, y.Ctx] == code"$y * 2"
    code"val $y = 3; $e" : Code[Int, Any]
    == code"val x = 3; x * 2"
    val c = code"(x: Int, $y: Int) => $e"
    c → code"(x_0: Int, x_1: Int) => x_1 * 2"
Context weakening and strengthening

Scala subtyping lattice: top is \texttt{Any}; bottom is \texttt{Nothing}

Context weakening

\[
C \& D \ <: \ D \ <: \ Any
\]

\[
\text{Code}[T, \text{Any}] \ <: \ \text{Code}[T, D] \ <: \ \text{Code}[T, C \& D]
\]

(Based on variance: \texttt{class Code[+T,-C]})

Context strengthening: \texttt{c.without(v)} tests whether \(v\) free in \(c\)

\[
\text{val cde1: Code[String, v.Ctx \& w.Ctx] = code"$v \ast \$w"
}
\]
\[
cde1.without(w) : \text{Option[Code[String, v.Ctx]]} \rightarrow \text{None}
\]

\[
\text{val cde2: Code[String, v.Ctx \& w.Ctx] = code"$v \ast 123"
}
\]
\[
cde2.without(w) : \text{Option[Code[String, v.Ctx]]} \rightarrow \text{Some(cde2)
}
Advanced example: binding order manipulation

reverseBindingsOrder(code"val a = 1; val b = 2; val c = 3; a + b + c")
== Some(code"val c = 3; val b = 2; val a = 1; a + b + c")

Possible thanks to Scala's advanced type system:

```scala
def reverseBindingsOrderBy[T, C](p: Code[T, C]): Option[Code[T, C]] = {
  go(p, [A, B] => identity)

    case code"val $v: Int = $init; $body" =>
      go(body, [S, D] => (cde: Code[S, D & C & v.Ctx]) =>
        code"val $v = $init; ${k[S, D & v.Ctx](cde)}"
      ).flatMap(b => b.without(v).toList)
    case expr => Some(k(expr))
  }
}
```
Term rewriting

Rewrite entire program fragment bottom-up

```pgrm rewrite {
    case code"($ls: List[$ta]).map($f).map($g)"
        => code"$ls.map($g.compose($f))"
}
```

Primitive `abort()` for when rewriting turns out impossible

```pgrm rewrite { case code"..." => ... abort() ... ... }
```

Still ensures context safety

Various transformation strategies
- (top-down, bottom-up, fixed-point, speculative)
Usage example: list maximum

Simple program – list maximum or default value

```scala
def maxOr(xs: List[Int], default: Int): Int =
  xs.foldLeft(None) {
    (acc, x) => Some(acc.fold(x)(y => max(x, y)))
  }.getOrElse(default)

Some(a).fold(d)(f) == f(a)
None .fold(d)(f) == d
List(1,2,3,4).foldLeft(z)(f) == f(f(f(f(z,1),2),3),4)
```
Usage example: list maximum

More efficient imperative version:

```scala
def maxOr(xs: List[Int], default: Int): Int = {
  var cur = xs
  var acc_val = 0
  var acc_isdef = false
  while (cur.nonEmpty) {
    val x = cur.head
    cur = cur.tail
    if (acc_isdef) { acc_val = max(x, acc_val) } else { acc_val = x }
    acc_isdef = true
  }
  if (acc_isdef) acc_val else default
}
```
Usage example: flatten variable of option type

```scala
object Opt extends Transformer with BottomUp with FixedPoint {
  def apply[T,C](pgrm: Code[T,C]): Code[T,C] = pgrm match {
    case code"($ls: List[$t0]).foldLeft[$t1]($z)($f)" =>
      code""" var cur = $ls; var res = $z;
      while (cur.nonEmpty) { val e = cur.head;
        cur = cur.tail; res = $f(res, e) }
      res """
    case code"($opt:Option[$t0]).getOrElse($e)" =>
      code"$opt.fold($e)(identity)"
    case code"None.fold[$str]($z)($e)" => z
    case code"Some[$t0]($v).fold[$str]($z)($f)" => code"$f($v)"
  }
}
```
Usage example: flatten variable of option type

```scala
object Opt extends Transformer with BottomUp with FixedPoint {
  def apply[T,C](pgrm: Code[T,C]): Code[T,C] = pgrm match {
    case code"($ls: List[$t0]).foldLeft[$t1]($z)($f)" =>
      code""" var cur = $ls;
      var res = $z;
      while (cur.nonEmpty) {
        val e = cur.head;
        cur = cur.tail;
        res = $f(res, e)
      }
      res"
    case code"($opt:Option[$t0]).getOrElse($e)" =>
      code"$opt.fold($e)(identity)"
    case code"None.fold[$str]($z)($e)" => $z
    case code"Some[$t0]($v).fold[$str]($z)($f)" => code"$f($v)"
  }
}
```

Usage example: flatten variable of option type

```
if mistake (eg: fold(identity)($e)) → type error at compile time
```
Usage example: flatten variable of option type

Tricky part of

\[
\text{def apply}[\text{T,C}](\text{pgrm: Code}[\text{T,C}]): \text{Code}[\text{T,C}]
\]

\[
\ldots
\]

\[
\text{case code"var } \text{r: Option}[\text{t0}] = \text{v0}; \text{ body" =>}
\]

\[
\text{val r_isdef} = \text{Variable}[\text{Boolean}]
\]

\[
\text{val r_value} = \text{Variable}[\text{t0.Typ}]
\]

\[
\text{val b2} = \text{body rewrite } \{
\text{case code"$$r.\text{fold}[\text{str}](z)(f)" =>}
\text{code"if (r_isdef) f(r_value) else z"}
\text{case code"$$r = \text{opt}" =>}
\text{code"opt.fold(())(v => r_value = v)" } \}
\]

\[
\text{code"""var r_value = v0.getOrElse($t0.defaultValue);}
\text{var r_isdef = v0.isDefined; b2"""}
\]

\[
.\text{without(r).getOrElse(abort())}
\]
Usage example: flatten variable of option type

Tricky part of

```scala
def apply[T,C](pgrm: Code[T,C]): Code[T,C] 
...

case code"var $r: Option[$t0] = $v0; $body" =>

val r_isdef = Variable[Boolean]
val r_value = Variable[t0.Typ]

val b2 = body rewrite {
  case code"$$r.fold[$tr]($z)($f)" =>
    code"$$r.fold[$tr]($z)($f)" 
    case code"$$r.fold[$tr]($z)($f)" =>

    forget to capture or substitute FV?
    → type error at compile time

  case code"var $r: Option[$t0] = $v0; $body" =>
    code"var $r: Option[$t0] = $v0; $body"

  case code"$r.isdef = $v0.isDefined; $b2""" 
    .without(r).getOrElse(abort())

  case code"$r.isdef = $v0.isDefined; $b2"""
    .without(r).getOrElse(abort())

  case code"$r.isdef = $v0.isDefined; $b2"""
    .without(r).getOrElse(abort())
```

List maximum, reloaded: **high-level and efficient**

```scala
def maxOr(xs: List[Int], default: Int): Int = Opt.optimize {
  xs.foldLeft(None) {
    (acc, x) => Some(acc.fold(x)(y => max(x,y)))
  }.getOrElse(default)
}
```

Can now change high-level code freely

optimization happens automatically
Type-Safe Metaprogramming: Limitations
Limitations of naive metaprogramming

Simple approach, many limitations

- function boundaries
- binding boundaries, side effects
- control-flow boundaries (join points)
Limitation: function boundaries

Rewrite rules are blind to escaping usages

```java
var opt = ... ; ... myFunction(opt).fold(...)(...) ...
```

Usual solution: aggressive inlining

Problem: when to inline what?

- eagerly inlining high-level constructs...

  removes high-level optimization opportunities!

such as list fusion

```java
case code"($ls: List[$ta]).map($f).map($g)" => ...
```
Solution: lowering + normalization phases

Lowering phases:

At each phase

- inline functions at the current level of abstraction
- normalize usages
- apply optimizing rewritings
Limitation: binding boundaries, side effects

Examples

```scala
val tmp = Some(expr); ... tmp.fold(...)(...) ...
val a = { val b = Some(expr); b }; a.fold(...)(...)
```

Need to commute and see through bindings

But what if `expr` has side effects?

What if `f` and `g` has side effects in `ls.map(f).map(g) ...`?

Solution:

rewrite engine needs to be side-effects-aware

i.e., need internal effect system
Limitation: control-flow boundaries

Finally, consider — patterns that commonly arises:

\[ \text{opt.fold}(d)(f).\text{fold}(e)(g) \quad \text{OR} \quad (\text{if} \,(\ldots) \,\ldots \,\text{else} \,\ldots).\text{fold}(e)(g) \]

Rewrite them to?

\[ \text{opt.fold}(d.\text{fold}(e)(g))(x => f(x).\text{fold}(e)(g)) \]
\[ \text{if} \,(\ldots) \,\ldots \text{fold}(e)(g) \,\text{else} \,\ldots \text{fold}(e)(g) \]

Does not scale: duplicates \( e, g \)

Solution: notion of join points

Requires smarter intermediate representation (CPS, ANF, CFG)

*Squid supports user-defined IR!*
Conclusion: rewriting needs compiler help

A practical optimizer needs:

- Progressive lowering + normalization phases
- Ability to look across and commute bindings
- Notion of side effects
- Practical representation of control-flow

These are what the "extensible compiler" part of Squid is about
Application Example:
Fusion for a Streaming Library
Expressive core constructs

```python
type Stream[A] <: {
    def map[B](f: A => B): Stream[B]
    def flatMap[B](f: A => Stream[B]): Stream[B]
    def take(n: Int): Stream[A]
    def filter(p: A => Bool): Stream[A]
    def zipWith[B](that: Stream[B]): Stream[(A, B)]
    def fold[B](z: B)(f: (B, A) => B): B
    ...
}

def fromRange(from : Int, until: Int): Stream[Int]
def unfold[A, B](init: B)(next : B => Option[(A, B))]: Stream[A]
```
"Shorthand" constructs

```
@phase("Sugar") def fromArray[A](xs: Array[A]): Stream[A] =
  fromRange(0, xs.length).map(i => xs(i))
```

"Low-level" constructs

```
@phase("Low") def flatMap[B](f: A => Stream[B]): Stream[B]
```

"Internal" constructs

```
@phase("Internal") def doWhile(f: A => Bool): Unit
```
QSR: Stream Fusion

Example optimization:

```kotlin
@bottomUp @fixedPoint val Flow = rewrite {
    // Floating out pullable info
    case code"pull($as) map $f "
        => code"pull($as map $f)"
    case code"pull($as) filter $pred "
        => code"pull($as filter $pred)"
    case code"pull($as) take $n "
        => code"pull($as take $n)"
    case code"pull($as) flatMap $f" => code"$as flatMap $f"
    // Folding
        ^ flatMap is not 'pullable'
    case code"pull($as) doWhile $f " => code"$as doWhile $f"
    case code"$as map $f doWhile $g"
        => code"$as doWhile ($f andThen $g)"
    case code"$as filter $pred doWhile $f"
        => code"$as doWhile { a => !$pred(a) || $f(a) }"
    case code"$as take $n doWhile $f"
        => code""""var tk = 0
            $as doWhile { a => tk += 1; tk <= $n && $f(a) }"""
    case code"$as flatMap $f doWhile $g"
        => code""""$as doWhile { a => var c = false
            $f(a) doWhile {b => c = $g(b); c}; c }"""
    // Zipping
    case code"$as zip pull($bs) doWhile $f " => code""
        $as.doZip($bs.producer()){ (a,b) => $f((a,b)) }
    case code"pull($as) zip $bs doWhile $f " => code""
        $bs.doZip($as.producer()){ (b,a) => $f((a,b)) }
}

Figure 7. Algebraic rewrite rules for stream fusion.
Stream Fusion

Example optimized program:

```scala
// Source:
fromRange(0, n).zip(
    fromRange(0, m).map(i => fromRange(0, i)).flatMap(x => x)
) filter { x => x._1 % 2 == 0 } foreach println

// After Desugaring:
pullable(fromRangeImpl(0, n)).zip(
    pullable(fromRangeImpl(0, m))
    .map(i => pullable(fromRangeImpl(0, i)))
    .flatMap(x => x)
).filter { x => x._1 % 2 == 0 } .doWhile { x => println(x); true }

// After Flow:
val p = fromRangeImpl(0, n).producer()
fromRangeImpl(0, m) doWhile { i =>
    var cont_0 = false
    fromRangeImpl(0, i) doWhile { b =>
        var cont_1 = false
        p { a => if (a % 2 == 0) println((a, b)); cont_1 = true }
        cont_0 = cont_1
        cont_0 }
    cont_0 ; cont_0 }

// After Lowering, the code has only variables and loops
```
Microbenchmarks for functional stream pipelines

- **1–2 orders of magnitude faster** vs. unstaged/iterators
- **As fast as hand-written loops** and staged code
  (but much more user-friendly than both)
Metaprogramming beyond expressions
Metaprogramming beyond expressions

Squid so far: powerful manipulation of expressions (composition, matching, rewriting, execution...)

Great for:

- static targeted optimization with `optimize { ... }`
- generation of specialized computation kernels
- on-the-fly compilation of small components

What about definitions? Want to generate full programs?

Squid’s solution: `first-class classes`
Normal Scala class:

```scala
class Vector(val x: Int, val y: Int, val z: Int)
val v = new Vector(0,1,2)  →  Vector(0,1,2)
val f = (u: Vector) => u.x + u.y
f(v)  →  1
```

Squid-virtualized representation of same class:

```scala
object Vector extends Class { val x, y, z = param[Int] } 
val v = Vector(c"0", c"1", c"2")  →  c"new Vector(0,1,2)"
val f = c"(u: Vector.T) => ${Vector.x}(u) + ${Vector.y}(u)"
val f = c"(v: Vector.T) => v.x + v.y"  // syntax sugar
val f = c"$f($v)"  →  c"0 + 1"  // automatically reduce pure classes
```
Application: I'm working on **dbStage**

- Users define their integrated database with Scala constructs
- Users define queries using a Scala DSL
- dbStage generates efficient implementations

```scala
class TableClass(name: String) extends Class("Row_"+name) {
  { val params: Array[Param[_]] } 

  val rowClasses = schema.tables.map { tbl =>
    new TableClass(tbl.name) {
      val params = tbl.columns.map {
        case (cname, ctyp) => param[ctyp.Typ]("col_"+cname)
      }
    }
  }

  // generates:
  class Row_Person(val col_name: String, col_age: Int)
  ...
```
Conclusions

Metaprogramming made type- and scope-safe without pain:

- quoted program fragments `code"..."`
- contextual types made of intersections `C & v.Ctx`
- hygienic, first-class variables `Variable[T]`
- automatic weakening based on variance `Code[+T,-C]`

Optimizations enabled by synergy with compiler technology

- strategic rewriting `rewrite, abort`
- advanced intermediate representations: ANF, Graph, ...
- effect system
- progressive lowering phases to remove abstraction

Squid: open-source implementation for Scala 2, soon Scala 3
Thank you!
A bit about the type system

Typing judgement  \( \Gamma \vdash t : T \rightarrow \Gamma \)

context of current term  context outside current quote

No need for stack of context: only unquote variables

code"""" foo(code"bar($f($x))")"""" // nested quote/unquote
⇔ code"""" val u = f($x); foo(code"bar($u)")"""

T-Quote

\[
\begin{array}{c}
C \vdash t : T \rightarrow \Gamma \\
\hline
\Gamma \vdash \text{code}"t" : \text{Code} [T, C] \rightarrow \Gamma'
\end{array}
\]

T-Unquote

\[
\begin{array}{c}
(x : \text{Code}[T, C]) \in \Gamma' \\
\hline
\Gamma \vdash \$x : T \rightarrow \Gamma'
\end{array}
\]
def optimize[T,C](pgrm: Code[T,C]): Code[T,C] = pgrm rewrite {
  case code"val arr = new Array[($ta,$tb)]($size); $body" =>
    val a = code"?a:Array[$ta]"; val b = code"?b:Array[$tb]"
  val body2 = body rewrite {
    case code"${body.arr}($i)._1" => code"$a($i)"
    case code"${body.arr}($i)._2" => code"$b($i)"
    case code"${body.arr}.size" => code"$a.size"
  }
  val body3 = body2.arr ~> abort()
  code"""val a = new Array[$ta]($size)
              val b = new Array[$tb]($size)
              "$body3"""""""
Advanced example: Scala 2 version (verbose)

reverseBindingsOrder(code"val a = 1; val b = 2; val c = 3; a + b + c")
   == Some(code"val b = 2; val a = 1; val c = 3; a + b + c")

Possible thanks to Scala's advanced type system:

```scala
/** Safely reverse the order of val bindings, if possible. */

def reverseBindingsOrder[T: CodeType, C](p: Code[T, C]): Option[Code[T, C]] = {
  class K[C] {
  }
  def go[T: CodeType, C](p: Code[T, C], k: K[C]): Option[Code[T, C]] = p match {
    case code"val \$v: Int = \$init; \$body" =>
      go(body, new K[C & v.Ctx] {
        override def apply[S: CodeType, D](cde: Code[S, D & C & v.Ctx]) =
          code"val \$v = \$init; ${k[S, D & v.Ctx](cde)}"
      }).flatMap(b => b.without(v))
    case expr => Some(k(expr))
  }
  go(p, new K[C])
}
```

58
Publications


Multi-Stage Programming (a.k.a., Staging)

Squid features allow multi-stage programming (staging) i.e., separate execution of program into code-generation phases.

Each stage executes parts of program known at this stage; delays execution of rest by emitting code.

Last stage: much simpler/more efficient program, where most abstractions have been removed.

Canonical example of MSP: power function.

Start from normal implementation; add staging annotations.
Limitations of Pure Staging

Traditional staging: purely generative
But Squid also supports rewriting... can be combined!
i.e., *Quoted Staged Rewriting* (QSR)