Squid: Type-Safe, Hygienic, and Reusable Quasiquotes

Lionel Parreaux, Amir Shaikhha, Christoph E. Koch
EPFL, Switzerland — {firstname.lastname}@epfl.ch

Abstract

Quasiquotes have been shown to greatly simplify the task of metaprogramming. This is in part because they hide the data structures of the intermediate representation (IR), instead allowing metaprogrammers to use the concrete syntax of the language they manipulate. Scala has had "syntactic" quasiquotes for a long time, but still misses a statically-typed version like in MetaOCaml, Haskell and F#. This safer flavor of quasiquotes has been particularly useful for staging and domain-specific languages. In this paper we present Squid, a metaprogramming system for Scala that fills this gap. Squid quasiquotes are novel in three ways: they are the first statically-typed quasiquotes we know that allow code inspection (via pattern matching); they are implemented purely as a macro library, without modifications to the compiler; and they are reusable in the sense that they can manipulate different IRs. Adapting (or binding) a new IR to Squid is done simply by implementing a well-defined interface in the style of object algebras (i.e., tagless-final). We detail how Squid is implemented, leveraging the metaprogramming tools already offered by Scala, and show three application examples: the definition of a binding for a DSL in the style of LMS; a safe ANF conversion; and the introduction of type-safe, hygienic macros as an alternative to the current macro system.

CCS Concepts • **Software and its engineering → Language features**; *Domain specific languages*; *Macro languages*;

Keywords Quasiquotes, Type-Safety, Embedded Domain-Specific Languages

ACM Reference Format:
Lionel Parreaux, Amir Shaikhha, Christoph E. Koch. 2017. Squid: Type-Safe, Hygienic, and Reusable Quasiquotes. In *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3136000.3136005

1 Introduction

Scala allows different forms of quotation to coexist. For example, `"2+2"` denotes a string made of characters `'2'`, `'+'` and `'2'`, but when prefixed with `q` as in `q"2+2"` it represents an *abstract syntax tree* (AST) equivalent to:

```
q"2+2" ==
  Apply(Select(Literal(Constant(2)), TermName("$plus")),
    List(Literal(Constant(2))))
```

As one can see, expressing code using the quoted form `q"2+2"` is much more concise than using the "explicit" (non-quoted) form above. Moreover, the explicit form exposes details of the internal encoding of Scala ASTs that are not usually relevant to metaprogrammers, such as the names of abstract syntax constructs (`Apply`, `Select`, etc.). For these reasons, quasiquotes have been widely adopted by metaprogrammers in languages like Scala [Shabalin et al. 2013], Haskell [Sheard and Jones 2002] and F# [Syme 2006].

Despite having achieved widespread adoption, Scala quasiquotes have important limitations that restrict their potential applications, namely their lack of reusability and their lack of static safety. First, they can only be used to manipulate Scala ASTs, precluding usages in the growing field of domain-specific languages (DSL). This is because "deeply-embedded" DSL programs are typically encoded in specialized intermediate representations (IR) that are more advanced than Scala's general-purpose AST representation.[1] Second, Scala quasiquotes lack static safety, since: 1. they are not statically type-checked, which means that they do not prevent the construction of ill-typed code; and 2. they are not "hygienic," as variable bindings in manipulated programs may interfere with each other (unintended capture), and are therefore not guaranteed to retain the meaning they are intended to have when introduced.

In this paper, we propose an approach to quasiquotation for Scala that resolves these limitations. Our approach is type-safe and hygienic, preventing the construction of ill-typed code and the occurrence of unintended variable capture. Furthermore, it is reusable in the sense that it can be used to manipulate programs encoded with different intermediate representations. The approach is realized as **Squid**,[2] a pure Scala macro library that requires no changes to the Scala compiler. We detail how Squid is implemented leveraging

[1] DSL development typically relies on fast normalizing IR data structures [Rompf and Odersky 2010], which precludes the use of Scala ASTs.
[2] "Squid" stands for the approximate contraction of ***S**cala **qu**oted **D**SLs*. (Much like the framework, squids are smart and flexible animals!) Squid is open source, available online at https://github.com/epfldata/Squid.

Scala's extensive metaprogramming capabilities, and hope that it provide a good use case for future work on the redesign of Scala macros. Squid was successfully used to implement the Quoted Staged Rewriting (QSR) pattern [Parreaux et al. 2017], an approach to library-defined optimizations.

Note that Squid quasiquotes focus on the *expression* side of Scala; they cannot manipulate class, method, object or type definitions. Thus Squid quasiquotes allow applying polymorphic constructs but not defining them. In contrast, existing Scala quasiquotes allow manipulating all Scala constructs, but with much weaker guarantees.

Our contributions are organized as follows:

- We show the limitations of existing metaprogramming approaches, both explicit and quoted (Section 2).
- We show how Squid solves these problems and still enables powerful code manipulations (Section 3).
- We detail a type-safe, quoted ANF conversion to demonstrate the flexibility of our approach (Section 4).
- We develop our solution to reusable quasiquotes, and explain how to bind new IRs to Squid (Section 5).
- We show how Squid can be used as a type-safe and hygienic alternative to current Scala macros (Section 6).

2 Expressing IR Manipulation

In this section, we describe common problems encountered by embedded DSL (EDSL) developers and quasiquote users while writing program transformations.

2.1 Explicit Approach

EDSL IRs represent DSL expressions using algebraic data types (sum and product types). For example, Figure 1a shows one possible definition of an IR for lambda calculus augmented with integers and addition, where all expression nodes have the same base type `Exp`. In this representation, β-reduction (for example) may be written as follows:[3]

```
def beta: Exp => Exp = {
  case App(Fun(f), a) => beta(f(a))
  case App(f,a) => App(beta(f), beta(a))
  case Add(a,b) => Add(beta(a), beta(b))
  case Fun(f) => Fun(x => beta(f(x)))
  case Const(v) => Const(v)               }
```

Manipulating such an IR is error-prone, as it is easy to construct nonsensical terms such as `App(Const(1),Const(2))`. To avoid such problems, the practice has been to *reflect* the type of an object term (a term in the language being manipulated) in the type of its corresponding IR node. This can be done by using Generalized Algebraic Data Types (GADTs) [Kennedy and Russo 2005], as shown in Figure 1b. Notice that `Exp` is now equipped with a type parameter that documents the type of the term it represents. While this generally improves the safety of IR manipulations, it also makes them more cumbersome to write. In Scala, where GADT pattern matching support is less than ideal [Giarrusso 2013], this means we have to use extra type annotations and sometimes unsafe casts, partially defeating the purpose. The beginning of the `beta` function above becomes:

```
def beta[T]: Exp[T] => Exp[T] = {
  case App(Fun(f), a) => f(beta(a))
  ...
```

The GADT pattern-matching above may give a false sense of safety to the programmer — in fact, the latest version of the Scala compiler[4] will happily compile the same code but where `f(beta(a))` has been replaced by `f(Const(1))`. This is unsound, because we have no indications that `f` has type `Exp[Int] => Exp[T]`. In reality, `f` should have type `Exp[t0] => Exp[T]` where `t0` is some existentially-quantified type, but Scala currently fails to handle that case properly.[5]

On the other hand, the need often arises for more advanced IRs than plain AST [Stanier and Watson 2013], such as ANF (Administrative Normal Form) [Flanagan et al. 1993], SSA (Static Single Assignment) or CFG (Control Flow Graph). This is particularly important when DSLs start incorporating effects and mutability, where evaluation order and aliasing become significant. In addition, it is usually desirable to keep the IR internally typed (i.e., nodes should store a runtime representation of their type). Figure 1c presents an ANF IR for our simple language, storing type and effect information. Note that in ANF, all non-trivial expressions are let-bound, so it makes sense to have a representation of code blocks (the `Block` class). Type representations are "captured" as implicit parameters, reminiscent of LMS [Rompf and Odersky 2010]. This is closer to a realistic IR than the previous toy examples. The cases for e.g., `App` and `Add` are now quite complex and hard to read:

```
case LetBind(App(LetBind(f: Fun[t0,T],effFun),a),effApp) =>
  f.lam(beta(a.asInstanceOf[Exp[t0]]))
case s @ LetBind(App(f: Exp[Function1[t0, T]],a),effApp) =>
  val a0 = a.asInstanceOf[Exp[t0]]
  LetBind(App(beta(f), beta(a0))(a0.typ, s.typ), effApp)
case LetBind(Add(a,b), effAdd) =>
  LetBind(Add(beta(a), beta(b)), effAdd)
```

Notice how we need to propagate types manually in the case for `App`, and how secondary meta-information like effects have to be dealt with explicitly. IR transformation becomes very error-prone, especially since one has to be careful to account for effects and avoid performing transformations that would change the evaluation order.

The design of a DSL and of associated program transformations (such as domain-specific optimizations) quickly becomes entangled with these low level IR implementation concerns, which get in the way of DSL designers.

[3] Syntax `{case ...}` is shorthand for `x => x match {case ...}`.

[4] Scala 2.12.2. Dotty, the next-generation compiler for Scala, also still has problems with GADTs (e.g., https://github.com/lampepfl/dotty/issues/2985).

[5] Using explicit type variables in patterns (such as '`f:Fun[t0,T]`') can help, but it is still demonstrably unsound, because the corresponding type `t0` is viewed by Scala as a transparent wrapper over its underlying type `Any`.

```scala
sealed abstract class Exp
case class Const(value: Int) extends Exp
case class Add(lhs: Exp, rhs: Exp) extends Exp
case class Fun(lambda: Exp => Exp) extends Exp
case class App(fun: Exp, arg: Exp) extends Exp
```

(a) Simple uni-typed AST

```scala
sealed abstract class Exp[+T]
case class Const(value: Int) extends Exp[Int]
case class Add(lhs:Exp[Int], rhs:Exp[Int]) extends Exp[Int]
case class Fun[A,R](lam: Exp[A] => Exp[R]) extends Exp[A=>R]
case class App[A,R](fun:Exp[A=>R],arg:Exp[A]) extends Exp[R]
```

(b) GADT AST

```scala
sealed abstract class Exp[+T](implicit val typ: Typ[_ <: T])
  { val eff: Effects }
case class Const(value: Int) extends Exp[Int]{ val eff=Pure }
case class Block[+T:Typ](defs:Buffer[LetBind[_]],ret:Exp[T])
  extends Exp[T] { val eff = /* compute effects ... */ }
case class LetBind[+T](defn: Def[T], eff: Effects)
  extends Exp[T]()(defn.typ)
abstract class Def[+T](implicit val typ: Typ[_ <: T])
case class Add(lhs:Exp[Int], rhs:Exp[Int]) extends Def[Int]
case class Fun[A:Typ,R:Typ](lam: Exp[A] => Exp[R])
  extends Def[A => R]()(funType[A,R])
case class App[A,R](fun: Exp[A => R], arg: Exp[A])
                   (implicit typ: Typ[_ <: R]) extends Def[R]
```

(c) ANF IR storing type representations and effects

Figure 1. Outline of different possible Intermediate Representations (IR) to represent a DSL.

2.2 Existing Scala Quasiquotes

In Scala, quasiquotes can be used as both expressions and patterns. Syntax `${...}` is used to *unquote* terms from inside a quasiquote.[6] (When the unquoted term is a simple variable, the curly braces can be omitted.) In quasiquote expressions, unquoted terms are *inserted* into the surrounding code. In quasiquote patterns, unquotes *extract* the terms found in their positions, matching them with the unquoted pattern. For example, `q"2+1" match { case q"$n+1" => q"$n-1" }` evaluates to `q"2-1"`.

It is straightforward to write a version of the β-conversion function presented above using Scala quasiquotes:

```scala
def beta: Tree => Tree = {
  case q"(($ident: $t0) => $body)($a)" =>
    beta(body transform { case `ident` => a })
  case q"$a + $b" =>
    q"{beta(a)} + {beta(b)}"
  case q"$f($a)" =>
    q"{beta(f)}({beta(a)})"
  case q"($ident: $t0) => $body" =>
    q"($ident: $t0) => {beta(body)}"
  case Literal(Constant(n:Int)) => Literal(Constant(n))
  case Ident(name) => Ident(name)                        }
```

`Tree` is the type of a Scala AST. Syntax `t.transform(f)` traverses some tree `t` trying to apply partial function `f` on each of its subterms. Pattern `` `ident` `` matches any value equal to `ident` (the extracted variable identifier).

Beyond the fact that this version of the `beta` function works only with Scala ASTs and is untyped — i.e., it makes it easy to generate nonsensical terms such as `q"1(2)"` — it suffers from additional limitations:

Hygiene. The transformation is unsound in the presence of shadowing. For example, it will transform `q"((x:Int) => (x:Int) => x)(1)"` into `q"(x:Int) => 1"` instead of transforming it into `q"(x:Int) => x"`. This is because variable references are simple named identifiers. This example is only one part of the hygiene problem, which is also prominent when quasiquotes are used within macro definitions — two other manifestations of the lack of hygiene are: 1. newly-introduced variable bindings may clash with bindings already present in the original program, so one has to manually generate "fresh names" (the `gensym` approach); and 2. references to global symbols (such as `println`) need to be fully-qualified (i.e., `_root_.scala.Predef.println`) to avoid unintended capture of user-defined symbols.

Propagation of internal typing. Even if the original program passed to `beta` was associated with typing information, this information is lost and is not propagated into the transformed program. Essentially, given two ASTs `a` and `b` both assigned with type `Int`, the term `q"$a+$b"` will *not* be assigned type `Int`, unless it is type checked again or manually annotated (e.g., as `q"$a+$b".withType(IntType)`).

Normalization. Given some function `f` of type `Int => Int`, the following code fragments are all equivalent:

```scala
f(Int.MaxValue)          f.apply(Int.MaxValue)
f(scala.Int.MaxValue)    import Int.{MaxValue => MV}; f(MV)
f(Int.MaxValue):Int      f(Int.MaxValue:Int)
```

Yet, a quasiquote pattern such as `q"$fun(Int.MaxValue)"` will only match the first one (yielding `fun = q"f"`). This is problematic, as it means that when macro writers or DSL designers want to match certain usage patterns, they have to handle all equivalent representations and their possible combinations. Note that when the Scala compiler type checks a program, it rewrites all expressions into their "fully-explicit" form — in the example above, all forms except the last two are rewritten into `f.apply(scala.Int.MaxValue)`. However, relying on the assumption that terms are in type-checked form is also problematic, as any subsequent transformations may violate that assumption. Moreover, there is no way of checking that expression and pattern quasiquotes are always written in that form, so it is easy to introduce subtle code transformation problems by not fully adhering to it.

[6] Unquote [Abelson et al. 1991] is also referred to as *anti-quote* [Mainland 2007] or *escape* [Taha and Sheard 2000].

3 Squid Quasiquotes

In this section, we present Squid's approach to quasiquotation, and detail how it achieves both type safety and hygiene while remaining flexible enough for code manipulations.

3.1 Basics

Squid quasiquotes are prefixed with **code**, and manipulate IR nodes of type `Code[T]`, where `T` reflects the type of the represented object term (like in the GADT approach of Figure 1). For example, **code**`"42.toDouble"` has type `Code[Double]`.

The main difference with Scala quasiquotes is that Squid type checks the quoted code fragments, and uses the resulting typing information to create appropriate IR nodes. As a result, IR nodes are always internally represented in a fully-typed form: all type parameters are specified, the code is desugared (e.g., `f(123)` is represented as `f.apply(123)`) and implicit arguments are inferred. This is the case even when the quasiquote itself does not mention type parameters, uses syntax sugar, and/or omits implicits arguments. For example,[7] **code**`"List(1,2).map(_+1)"` is equivalent to:

```
code"List.apply[Int](1,2).map[Int,List[Int]]
                  ((x: Int) => x+1)(List.canBuildFrom[Int])"
```

Under the hood, Squid quasiquotes are macros that produce the boilerplate necessary for constructing or deconstructing IR nodes corresponding to the code being quoted.

3.2 Pattern Matching and Rewriting

Just like Scala quasiquotes, Squid quasiquotes support pattern-matching. However, type annotations are often required to help with Scala's local type inference. For example, pattern **code**`"$x+1"` does not type check, as the Scala type checker does not know which '+' method is implied when the type of `x` is unknown. The example in Section 2.2 is now written:

```
code"2+1" match { case code"($n:Int)+1" => code"$n-1" }
```

To help define sound rewritings, Squid provides a **rewrite** macro that traverses a program and applies a transformation to each of its sub-terms, while checking at compile-time that the transformation is type-preserving.

3.3 Type Evidence Implicits

In order to satisfy the requirement that IR nodes be internally typed (i.e., they should contain runtime information about the types of the terms that they encode), we require functions manipulating code in a generic way to pass along the associated type representations. Like in Section 2.1, the best way to do so is via implicits. Squid defines the `CodeType` type class for this purpose. As an example, the following function returns an empty option term for any type `T`:

```
def foo[T:CodeType] = code"Option.empty[T]"
```

[7] In Scala, `map` takes an implicit `CanBuildFrom` evidence [Odersky and Moors 2009], which semantics is irrelevant to this presentation.

(Note that syntax `T:CodeType` is shorthand for including an implicit parameter of type `CodeType[T]` in the function.) When `foo` is called as e.g., `foo[Int]`, an implicit type representation, of type `CodeType[Int]`, is resolved and passed along with the function call, so that the resulting term is the expected **code**`"Option.empty[Int]"`.

3.4 Type-Parametric Matching

To define type-parametric rewrite rules, Squid allows the extraction of types, not just terms. In the example below, given some `pgrm` fragment we transform calls to `foldLeft` on `List` objects into imperative `foreach` loops:

```
def lower[T](pgrm: Code[T]) = pgrm rewrite {
  case code"($ls: List[$t]).foldLeft[$r]($init)($f)" =>
    code""" var cur = $init
            $ls.foreach(x => cur = $f(cur,x))
            cur                                    """ }
lower(code"List(1,2,3).foldLeft(0)((acc,x) => acc+x) + 4")
```

The call above returns the equivalent of:

```
code"var cur=0; List(1,2,3).foreach(x => cur=cur+x); cur+4"
```

Note that multi-line quotations are introduced with `"""`, and that operator syntax `p` **rewrite** `f` means `p.`**rewrite**`(f)`.

Any type extracted as, e.g., `$t0` results in a *value* of type `CodeType[t0.Typ]`, where `t0.Typ` is a path-dependent type defined on local value `t0` so that it cannot be confused with any other extracted type. For example, one can write:

```
def bar(x: Code[Any]): Code[Any] = x match {
  case code"Some($_ : $t0)" => foo[t0.Typ]  case _ => x }
```

The type evidence passed to `foo[t0.Typ]` is automatically picked up from extracted type representation `t0`.

3.5 Automatic Function Lifting

An important feature of a flexible quasiquotation system is the ability to manipulate open terms. Since Squid quasiquotes are type-checked and hygienic, a quasiquote like **code**`"x+1"` is not valid, as `x` is undefined (contrast this with current Scala quasiquotes, where **q**`"x+1"` is entirely valid).

Some approaches such as MetaML [Taha and Sheard 2000] allow expressions like **code**`"(x:Int) => ${baz(`**code**`"x")}"`, where a function literal *binds* variable `x`, then function `baz` is called on a code fragment that refers to that variable `x` *across quotation boundaries*. Unfortunately, this approach is not possible without modifying the compiler of the host language (and Squid implements quasiquotes using Scala macros only). Thankfully, we can achieve the same effect with *automatic function lifting*: upon insertion, Squid automatically lifts any host-language function, of type `Code[A] => Code[B]`, into an object language function, of type `Code[A => B]`, and immediately inlines it. This way, we can write the pseudo-code above: **code**`"(x:Int) => ${(y:Code[Int]) => baz(y)}(x)"`.

3.6 Higher-Order Patterns Variables

Squid provides a very simple form of higher-order matching [de Moor and Sittampalam 2001; Pfenning and Elliott 1988], that directly mirrors automatic function lifting. In Squid, pattern `code"(x:Int) => $body:Int"` will not match a lambda where `body` makes use of `x`, while the following pattern will: `code"(x:Int) => $f(x):Int"`, giving to `f` type `Code[Int] => Code[Int]`. Applying `f` to some `Code[Int]` will replace all usages that `f` made of `x` with the provided code value. Higher-order pattern variables in quasiquote-based pattern matching were suggested before us by Sheard et al. [Sheard et al. 1999], but we do not know of any actual implementation of the idea, beside ours.

3.7 Beta Redux

Using Squid, we can now rewrite the β-reduction example seen in Section 2.2, but in a type-safe and hygienic way:

```
1  def beta[T:CodeType](x: Code[T]): Code[T] = x match {
2    case code"((p: $t0) => $body(p):T)($a)" =>
3      beta(body(a))
4    case code"($a:Int) + ($b:Int)" =>
5      code"${beta(a)} + ${beta(b)} : T"
6    case code"($f: ($t0 => T))($a)" =>
7      val (f0, a0) = (beta(f), beta(a))
8      code"$f0($a0)"
9    case code"(p: $t0) => $body(p): $t1" =>
10     code"((p: $t0) => ${
11       (r:Code[t0.Typ]) => beta(body(r)) }(p)) : T"
12   case Const(n) => Const(n)
13   case LeafCode() => x                              }
```

`Const` is the constructor for constant values, and `LeafCode()` is a custom extractor defined by Squid to match any simple IR node that has no sub-terms, such as bound variable references and constants.[8]

As a closing remark, notice the `: T` type ascriptions[9] on lines 5 and 11. They are necessary to make the program type check. Indeed, term `code"${beta(a)} + ${beta(b)}"` has type `Code[Int]` instead of the expected `Code[T]`. The Scala compiler has no specific knowledge of Squid quasiquotes, and so has no way to know that in that particular pattern branch, `T` is equivalent to `Int`. This problem is essentially the same as encountered with GADTs in Section 2.1. Fortunately, Squid keeps track of such uncovered type relations, and is able to perform the appropriate type coercions as long as they happen *inside* a quasiquote. As a result, we are able to soundly handle type relation refinements in pattern matching branches, and we avoid the persistent issues with the handling of GADTs in Scala that we described earlier.

4 Quoted ANF Conversion

Correctly handling bindings is one of the most common pitfalls in program manipulation. The higher-order pattern variable (HOPV) technique presented in Section 3.6, which is used to match binding constructs, can seem limiting because it extracts functions instead of directly-inspectable terms. In this section, we demonstrate that HOPVs are in fact fairly flexible, by presenting a more advanced usage example.

```
1  def toANF[T:CodeType](trm: Code[T]) = rec(trm)(identity)
2  def rec[T:CodeType,R:CodeType](trm: Code[T])
3        (k: Code[T] => Code[R]): Code[R] = trm match {
4    case Const(_) => code"val c = $trm; $k(c)"
5    case code"val x: $tx = $x; $body(x)" =>
6      rec(x)(x => rec(body(x))(k))
7    case code"($a:Int) + ($b:Int)" => rec(a)(a => rec(b)(b =>
8      code"val add: T = $a + $b; $k(add)"))
9    case code"($f: $t0 => T)($a)" => rec(f)(f => rec(a)(a =>
10     code"val app: T = $f($a); $k(app)"))
11   case code"(p: $t0) => ($body(p):$t1)" =>
12     code"val f: T = (p: $t0) => ${
13       (p:Code[t0.Typ]) => toANF(body(p))}(p); $k(f)"
14   case code"if ($cnd) $thn else $els" => rec(cnd)(cnd =>
15     code"""val join = $k
16       if ($cnd) ${ rec(thn)(_ : Code[T => R]) }(join)
17       else      ${ rec(els)(_ : Code[T => R]) }(join)""")
18   case _ => k(trm) }
```

Figure 2. Type-safe, hygienic ANF conversion.

Intermediate representations may automatically normalize terms into forms such as SSA, CPS or ANF. When this normalization step is implemented internally, it is transparent to users of the quasiquote-based Squid interface. For example, in the context of an ANF IR, `code"List(readInt)"` and `code"val x = readInt; val ls = List(x); ls"` are expressions that refer to equivalent internal term representations. When the IR is not internally normalized, it is also possible to perform ANF *conversion* as a type-safe, hygienic transformation expressed with quasiquotes. Figure 2 presents such a transformation for our toy lambda calculus with integers and addition, now extended with if-then-else. Squid provides implicit conversions to go back and forth between lifted (`Code[A => B]`) and unlifted (`Code[A] => Code[B]`) function types. Notice that in `rec(thn)(_ : Code[T => R])` (line 16), which is syntactic sugar for `(j:Code[T => R])) => rec(thn)(j)`, value `j` is "unlifted" to type `Code[T] => Code[R]` when it is passed to `rec(thn)`. Conversely, variable `k` on line 15 is lifted in order to be inserted. As an example, the program:

```
val foobar = {
  val foo = 123; val bar = 42; (if(true) foo else foo+2)+bar
}; foobar+1
```

is transformed into:

```
val c_0 = 123; val c_1 = 42; val c_2 = true;
val join_7 = ((lifted_3: scala.Int) => {
  val add_4 = lifted_3.+(c_1); val c_5 = 1;
  val add_6 = add_4.+(c_5); add_6 });
if (c_2) join_7(c_0)
else { val c_8 = 2; val add_9 = c_0.+(c_8); join_7(add_9) }
```

We can generalize our approach to handling other constructs by replacing the cases for integer addition and function application with `case MethodApplication(ma)`, which

[8] In fact, we could do without the `Const(n)` pattern matching branch, as it is already handled by `LeafCode()`. It is only there for presentation purposes.
[9] `T` refers to a *type*, not a type *representation*, so it may not be unquoted.

is a helper extractor defined by Squid. This extracts an object `ma` capable of representing any method application, which can then be rebuilt by applying a type-preserving transformation on each of its arguments, as follows:

```
case MethodApplication(ma) => ma.rebuild(new Code2CodeCPS {
  def apply[T:CodeType,R:CodeType] = rec
})(r => code"val tmp = $r; $k(tmp)")
```

The `rebuild` method takes an instance of `Code2CodeCPS` (an interface to express polymorphic code transformations in continuation-passing style) and a continuation argument that we use to bind the result to a `tmp` variable.

It is interesting to compare our implementation of A-Normalization to the original Scheme algorithm by [Flanagan et al. 1993]. The continuation-based structure is essentially the same, and the size of the program (19 lines of code in their case) is similar — even though, of course, our version is type-safe and propagates internal typing, which they do not. Another difference is that they need to use the error-prone "`gensym` discipline" to avoid introducing name clashes by manually generating fresh names. In our case, Squid takes care of these low-level details automatically — notice that in the example above, non-conflicting names are generated for each introduced binding (e.g., `add_4` and `add_6`).

5 Reusability via Object Algebras

In this section, we describe how Squid abstracts over the intermediate representation and provides general facilities to implement closed-world and open-world quasiquote backends. In this sense, we say that Squid quasiquotes are *reusable*, or "*generic*" in the IR. We are not aware of any previous quasiquotation system with similar capabilities.

5.1 The Intermediate Representation Base

Figure 3 shows the `Base` trait required to be implemented by all Squid backends, taking the form of an *object algebra interface* [Oliveira and Cook 2012] — also known as the *tagless-final* style [Carette et al. 2009] — where abstract type `Rep` represents the type of IR nodes, while types `Val` and `TypeRep` represent the types of bound variables and type representations, respectively.[10] Method `readVal` converts a variable symbol into a variable reference. `ascribe` corresponds to type ascription (of syntax `x:T` in Scala). Classes `Code` and `CodeType` have protected constructors in order to prevent external users from instantiating them arbitrarily.

Notice that Squid does not use a typed view of the IR (we have `Rep` instead of `Rep[T]`). This choice was motivated by simplicity of the Squid implementation and of the code generated by each quasiquote, ensuring faster compilation. Moreover, we noticed that when dealing with low-level IR manipulation, types often get in the way rather than help. Critically, this does not impact the soundness of high-level

[10] We do not show the methods for building type representations (TypeRep), but they follow the same pattern as for term representations (Rep).

```
trait Base {
  type Val ; type Rep ; type TypeRep

  def const(value: Any):                    Rep
  def freshVal(name: String, typ: TypeRep): Val
  def readVal(v: Val):                      Rep
  def lambda(param: Val, body: => Rep):     Rep

  // override if needed:
  def ascribe(self: Rep, typ: TypeRep):     Rep = self

  class Code[+T] protected(val rep: Rep)
  class CodeType[T] protected(val trep:TypeRep){ type Typ=T }

} // ...more helper methods and definitions elided
```

Figure 3. Abstract types and methods required for a Squid IR.

IR manipulation using quasiquotes, as high-level quasiquote terms are wrapped inside the typed `Code[T]` wrapper.

5.2 Closed Worlds

Perhaps surprisingly, the `Base` trait does not feature a function application method. This is because in Scala and Squid, applying a function corresponds to calling the `apply` *method* defined on the `scala.Function` type, and Squid has a special mechanism for encoding methods in a user-extensible way: when generating IR code for a method call inside a quasiquote, Squid looks for a method with a corresponding name in the `Base`. If no such method is found, a compile-time error reports the missing feature. To avoid name clashes, these methods should live in objects whose names reflect the full names of the types where the original methods are defined. For example, to bind the IR in Figure 1a to a Squid base, we include the following definitions:

```
object MyIR extends Base {
  type Rep = Exp ; type TypeRep = Unit
  object `class scala.Int` {
    def typeRep = ()
    def + (self: Rep)(arg: Rep) = Add(self, arg) }
  object `class scala.Function` {
    def typeRep(lhs: TypeRep, rhs: TypeRep) = ()
    def apply(self: Rep)(arg: Rep) = Apply(self, arg) }
  /* ... more definitions ... */ }
```

Remark that in Scala, identifiers delimited with back-ticks may contain any valid characters, so we literally named the objects above "`class scala.Int`" and "`class scala.Function`".

As an example, the code generated for `code"(x: Int) => x+1"` after having imported the '`code`' quasiquote builder from `MyIR` will be of the form:

```
val x = MyIR.freshVal("x", MyIR.`class scala.Int`.typeRep)
val rep = MyIR.lambda(x,
  MyIR.`class scala.Int`.+(MyIR.readVal(x), MyIR.const(1)))
new MyIR.Code[Int](rep)
```

We do not give the full IR binding here for lack of space. The online Squid repository contains several examples of custom Squid IRs, as well as a binding to an existing IR for the LMS-style DSL that was used in [Shaikhha et al. 2016].

5.3 Language Virtualization

For the economy of concepts, many Scala language features are internally encoded using the set of `Base` features that we have seen above. For this purpose, Squid defines a small library of *virtualized* constructs [Jovanovic et al. 2014; Moors et al. 2012]. For example: variables are represented using a `Var` data type supporting operations `.!` and `:=` for variable access and modification respectively; if-then-else and loops are implemented using functions such as `ifThenElse` and `While` taking by-name arguments; by-name arguments themselves are represented as calls to a `byName` function taking a `() => T` function parameter; finally, functions with more than one parameter are implemented with curried functions passed into `uncurryN` methods — for example, `(x:Int, y:Int) => x+y` is represented as `uncurry2((x:Int) => (y:Int) => x+y)`; pattern matching is represented using `isInstanceOf` and `unapply` calls.[11] Finally, by default let-bindings are represented as lambda abstractions immediately applied (*redex*).

Naturally, these virtualized encodings are invisible to the quasiquote user, and DSL designers may convert them into their own IR-specific representations. For example, in:

```
object MyIR extends Base {
  object `object squid.lib`
  { def ifThenElse(cond: Rep, thn: Rep, els: Rep) =
          buildInternalIfThenElseNode(cond, thn, els) }
} // ... more definitions elided
```

5.4 Open Worlds

In the context of metaprogramming "at large," like when writing general-purpose Scala macros (as opposed to DSL program transformations), it is useful to have a way to generate method applications on the fly, without having to define IR bindings manually for all possible methods.

This is possible thanks to the `OpenWorld` trait shown in Figure 4. If a base extends this trait, Squid will default to generating calls to `methodApp` to encode method applications that do not have a direct binding defined. `methodApp` takes a `tp` parameter so that the IR is informed of the type returned by the method call. `loadMtdSymbol` takes an overloading index to identify which method overload is being selected (`0` if the method is not overloaded).

As an example, assuming we do not have in `MyIR` a direct binding for type `Double` and method `toDouble`, the quasiquote `code"2.toDouble"` will expand into the equivalent of:

```
val _Int = MyIR.loadTypSymbol("scala.Int")
val _Double = MyIR.loadTypSymbol("scala.Double")
val _toDouble = MyIR.loadMtdSymbol(_Int, "toDouble", 0)
val rep = MyIR.methodApp(MyIR.const(2),
            _toDouble,Nil,Nil,typeApp(_Double,Nil))
new MyIR.Code[Double](rep)
```

The simplest way to implement methods `loadTypSymbol` and `loadMtdSymbol` is to make use of Scala Reflection's runtime

[11] A more handy representation of pattern matching is left as future work.

```
trait OpenWorld extends Base {
 type MtdSymbol ; type TypSymbol

 def loadTypSymbol(fullName:String):              TypSymbol
 def loadMtdSymbol
  (typ: TypSymbol, symName: String, index: Int):  MtdSymbol
 def methodApp(self: Rep, mtd: MtdSymbol,
  targs:List[TypeRep],argss:List[ArgList], tp:TypeRep): Rep
 def typeApp(typ: TypSymbol, targs: List[TypeRep]): TypeRep

} // ...more helper methods and definitions elided
```

Figure 4. The "open world" trait, which allows using any methods.

metaprogramming capabilities, reusing its `TypeSymbol` and `MethodSymbol` data types. This way, it is possible for an IR to dynamically explore things such as the annotations attached to a Scala method and its parameters, which is especially useful for implementing such mechanisms as annotation-based effect systems. Squid provides a ready-made `ScalaSymbols` trait that defines `loadTypSymbol` and `loadMtdSymbol` using Scala runtime reflection, so it is effortless for an IR to leverage these capabilities.

Finally, notice that using an open world IR generally allows for more flexibility. For example, it is possible to define programs that completely abstract over the `base` that is being used. Moreover, an open-world IR can be used as target to reinterpretation, as we will see in Section 5.6.

5.5 Support for IR Manipulation

In order to support pattern matching and term rewriting, a Squid IR has to extend yet another trait — `InspectableBase`, shown in Figure 5. The `Extract` type represents the result of pattern matching, and contains a mapping from term variable names to extracted terms and from type variable names to extracted type representations. `InspectableBase` defines the semantics of term and type pattern matching (`extract` and `extractTyp`), rewriting[12] (`rewriteRep`), code traversal/-transformation (`transform`), term equivalence (`repEq`) and subtyping (`typLeq`). Term equivalence is needed because Squid allows an extracted variable to be used in the same pattern, as in `case code"($a,a)"` which matches only pairs with twice the same component. Similar to `ScalaSymbols` for symbol loading, Squid provides a ready-made `ScalaTyping` trait that defines `TypeRep`, `typLeq`, `typeHole` and `extractTyp` relying on Scala's runtime type representation facilities.

Pattern matching is implemented by building an IR node representing the pattern, where unquotes are replaced with special "hole" nodes. The IR then provides the semantics of matching a given *expression* node against that *pattern node*. Thus methods `hole` and `typeHole` represent unquotes in patterns. In addition with name and expected type, `hole`

[12] `rewriteRep` can be implemented in terms of `transform` and `extract`, but we found that for advanced IRs such as ANF, it is often useful to have more control on the way rewritings apply, enabling more powerful patterns.

```
trait InspectableBase extends Base {
  type Extract = (Map[String, Rep], Map[String, TypeRep])

  def extract   (xtor: Rep, xtee: Rep):  Option[Extract]
  def rewriteRep(xtor: Rep, xtee: Rep,
    mkCode: Extract => Option[Rep]):     Option[Rep]
  def extractTyp(xtor: TypeRep, xtee: TypeRep,
                         va: Variance): Option[Extract]
  def transform(r: Rep)(pre: Rep=>Rep,post: Rep=>Rep): Rep
  def hole(name: String, typ: TypeRep,
        yes: List[Val], no: List[Val]):              Rep
  def typeHole(name: String):                    TypeRep
  def reinterpret(r: Rep, newBase: OpenWorld): newBase.Rep

  def repEq(a: Rep, b: Rep): Boolean
  def typLeq(a: TypeRep, b: TypeRep): Boolean
}
```

Figure 5. Base for allowing code inspection (e.g., pattern matching).

takes two lists of bound values `yes` and `no`, that specify respectively which bound references the hole is supposed to contain, and which it is forbidden to contain. For example, in pattern `case code"(x:Int,y:Int,z:Int) => $f(x,z)"`, the argument to `yes` will be `List(x,z)` and that to `no` will be `List(y)`. The `hole` method is supposed to extract a function term with arity equal to the length of `yes`. In the case above, it would extract an `(Int,Int) => Int` function term. On extraction, Squid lifts that term into a host-language function of type `(Code[Int],Code[Int]) => Code[Int]` automatically. Importantly, the term extracted by a `hole` may contain any references that appear in neither `yes` nor `no`. This is because code pattern-matching can be used in rewrite rules, which traverse every sub-term of a program and may open an arbitrary number of bindings on the way — as long as the terms extracted by a rule's pattern end up as part of the rule's result,[13] these bindings should not be affected.

In contrast with `Base` and `OpenWorld`, providing an implementation for `InspectableBase` is usually a non-trivial undertaking, the most difficult task being to implement complete pattern matching semantics. On the other hand, once this is in place, one can fully benefit from Squid's powerful and safe IR manipulation capabilities.

5.6 IR Reinterpretation

An important capability shown in Figure 5 is that offered by the `reinterpret` method: an `InspectableBase` may provide the capability to have its programs reinterpreted into a *different* Squid `Base`, which is an important tool that in turn enables many interesting applications (cf. Section 5.7). This is especially useful for optimizing high-level programs by progressively lowering their level of abstraction: at a certain point, we may want to switch to an IR which is more appropriate to deal with low-level programs.

Notice that `reinterpret` takes an `OpenWorld` parameter as the target `Base`, because it has to be able to reinterpret arbitrary features that may or may not be specially handled in the target IR. In practice, it is possible to adapt a non-`OpenWorld` IR to make it `OpenWorld`, using Java reflection to find the correct node creation methods at runtime.

[13] And are not *extruded* by imperative effects like variable update.

5.7 One Interface to Rule Them All

...and in Abstraction Bind Them

In this subsection, we describe how Squid's object algebra interfaces turned out to be a powerful tool that facilitated the implementation of several Squid features.

Code generation backend. It can be useful to convert a program expressed in some custom IR into a standard Scala AST. This is simply done by *reinterpreting* that code into the `ScalaAST` base, in which `type Rep = Tree` (where `Tree` is the type of Scala ASTs). For example, in that base we have `def const(value: Any) = Literal(Constant(value))`, which constructs a Scala AST for a constant literal.

Note that IRs that rely on virtualized constructs [Jovanovic et al. 2014; Moors et al. 2012] will typically refine the behavior of the `reinterpret` method in case the target is a subclass of `ScalaAST`, so that these constructs are correctly de-virtualized. For example, without de-virtualization we might observe the following behavior:

```
scala> code"if (true) 1 else 0" reinterpretIn (new ScalaAST)
result: universe.Tree = q"squid.lib.ifThenElse(true, 1, 0)"
```

To avoid this, the IR can special-case each virtualized construct in `reinterpret`, so that the expression above results in the expected Scala AST: `q"if (true) 1 else 0"`.

Pretty-printing. Pretty-printing is a standard application of object algebras [Oliveira and Cook 2012], and requires defining an algebra where `type Rep = String`. However, when we already have an `InspectableBase`, we can avoid writing a pretty-printer entirely: it suffices to reinterpret the code into `ScalaAST` and then reuse the standard Scala pretty-printer.

Evaluation by runtime reflection. Squid provides the base `ReflectInterpreter` that leverages Java runtime reflection to execute code at runtime. In that base, we have `type Rep = Runner[Any]` (where `Runner` is a data type that is used to build a runnable representation of the code), and `methodApp` uses Java reflection to load the correct method from its method symbol and create the appropriate runner. Thanks to this interpreter, running code from an arbitrary `InspectableBase` is as simple as calling `reinterpret` with a `ReflectInterpreter` instance — in fact, Squid provides a `run:T` helper method on `Code[T]` types that does just that.

Evaluation by runtime compilation A much more efficient but heavyweight way to implement code evaluation is to rely on Scala's runtime compilation capabilities. We can use the Scala compiler to generate extremely efficient JVM byte-code at runtime, a useful capability for performance-sensitive systems that rely on staging.

Modular embedding Remember that Squid leverages the Scala compiler to type check snippets of code, and then uses the result to build the corresponding IR nodes. We call our approach "*modular embedding*," because the IR construction process itself is abstracted, and is expressed in terms of the OpenWorld interface. For example, the case that lifts constants from the type-checked Scala AST is of the form:

```
  case Literal(Constant(x)) => base.const(x)
```

Where base is the OpenWorld Base object used to build the result of the embedding. The call to const refers to the function declared in Figure 3. This approach has the advantage that we can use modular embedding in different contexts:

- In the optimize{...} block construct shown in [Parreaux et al. 2017]: the optimize macro embeds a piece of code at compile time into a given Squid IR where optimizations are performed, then reinterprets the code into the ScalaAST base to produce the result of the macro expansion. A similar mechanism is used in the code generated by the @squidMacro construct presented in Section 6.
- In quasiquotes, which embed code snippets into a specific MirrorBase backend, whose role is to generates the Scala AST necessary to reconstruct the same code at runtime. In this base, const(42) results in the Scala AST q"$base.const(42)", where base identifies the target runtime base. Indeed, the role of quasiquotes is to create *run-time* code representations, as opposed to optimize whose goal is to handle code representations *at compile time*. Interestingly, the code invoked by optimize itself makes use of quasiquote-based "runtime" code manipulation — indeed, the runtime of the optimizer is the compile-time of the user program.

6 Type-Safe & Hygienic Macros for Scala

In this section, we briefly describe another feature of Squid, which acts like an alternative to the current Scala macros. As a motivating example, consider the typical power(x,n) function that raises number x to the nth power. We want to write a power macro that expands into a series of multiplications when the n parameter passed is a known constant.

A first version is shown in Figure 6, where Embedding, which extends InspectableBase, is the name of the IR chosen to manipulate code values within the macro. Annotation macroDef transforms a method definition into a macro. Like in Scalameta [Burmako 2017], the effect is that within the body of the annotated function, the parameters have type Code[T] instead of T, and we can inspect them as code values.

The macro in Figure 6 is "naive" in that it will duplicate the base code, resulting in potentially unnecessary computations and even in changes in program semantics — indeed, program naivePower(readInt,2) will expand into 1.0 * readInt * readInt. To correct this flaw, we have

```
@macroDef(Embedding)
def naivePower(base: Double, exp: Int): Double = {
  // in this scope, base:Code[Double] and exp:Code[Int]
  exp match {
    case Const(exp) =>
      var cur = code"1.0"
      for (i <- 1 to exp) cur = code"$cur * $base"
      cur
    case _ => code"Math.pow($base, $exp.toDouble)"  }}
```

Figure 6. Naive version of the power macro.

```
@macroDef(Embedding)
def power(base: Double, exp: Int): Double = {
  exp match {
    case Const(exp) =>
      code"val b = $base; ${(x:Code[Double]) =>
        var cur = code"1.0"
        for (i <- 1 to exp) cur = code"$cur * $x"
        cur
      }(b)"
    case _ => code"Math.pow($base, $exp.toDouble)"  }}
```

Figure 7. Correct definition of the power macro.

to first assign the value of base to a temporary variable, and duplicate a references to that variable instead. The corrected macro, which binds base to an intermediate variable, is presented in Figure 7.

7 Related Work

7.1 General Quasiquotation Systems

Quasiquotes were pioneered in Lisp [Bawden et al. 1999] as a shorthand for manipulating code in macros. Code as data in its simplest expression meant that no restrictions were in place whatsoever to prevent errors associated with code manipulation, such as unintended variable capture (lack of hygiene) and type mismatches (lack of static typing). Scheme introduced facilities to write hygienic macros [Abelson et al. 1991; Kohlbecker et al. 1986], but this was done by restricting their expressive power: hygienic macros have to consist of a list of pattern–template pairs, and so can only perform basic syntax expansion. Therefore, Scheme and its successor Racket still provide support for (unhygienic) quasiquotes, which are viewed as a lower-level building tool. Hygienic Scheme quasiquotes have been proposed [Rhiger 2012], but in a version that does not support pattern matching. The idea of quasiquotation was picked up in a statically-typed context by MetaML [Taha and Sheard 2000] (and subsequently MetaOCaml [Taha 2004]) to enable Multi-Stage Programming (MSP). The approach was ported to compile-time macros with MacroML [Ganz et al. 2001]. In these systems, quasiquotes can only *generate* and not *inspect* code — though MacroML has some limited form of pattern–template expansion similar to Scheme's hygienic macro system.

Template Haskell (TH) [Sheard and Jones 2002] introduced compile-time metaprogramming to Haskell and offered quasiquotes which had some notion of type awareness and hygiene, but could easily generate ill-typed and ill-scoped code, therefore providing weaker guarantees than MetaOCaml. Typed Template Haskell (TTH) later added type-safe quasiquotes similar to MetaOCaml. Neither MetaOCaml nor TH/TTH support term deconstruction via quasiquote pattern matching. However, a general quasiquotation syntax (not restricted to code quasiquotes) was introduced in Haskell by Mainland [Mainland 2007] and could in principle be used to enable quasiquote-based code pattern matching. A similar general quasiquote system exists in Scala and is used by the Scala Reflection API to provide Lisp-like untyped code quasiquotes with pattern matching [Shabalin et al. 2013], of which an example is given in Section 2.2. Squid uses the same system, but adds static type checking and hygiene. The Scala reflection API has an alternative type-safe and hygienic `reify`/`splice` system that can be used for program generation (`reify` acts like quotation and `splice` like antiquotation), but it does not allow the expression of open code and does not support pattern matching, limiting its usefulness. For example, for both of these reasons it cannot be used to implement the macro in Figure 7. Other languages like F# [Syme 2006] support various flavors of quasiquotes, but they all fall within the categories described above.

7.2 Quasiquotes for Domain-Specific Languages

Quasiquotes in MetaML [Sheard et al. 1999], Haskell [Najd et al. 2016], F# [Syme 2006] and others were used to facilitate the implementation of embedded DSLs such as language-integrated queries [Cheney et al. 2013]. Earlier approaches such as LINQ [Meijer et al. 2006] also provided some level of language-integrated domain-specific program reification. [Najd et al. 2016] use TTH to build DSL programs for their alternative embedding of Feldspar [Axelsson et al. 2010], an approach they call *Quoted DSLs* (QDSL). In this approach, a particular DSL is implemented using the quasiquotation abilities of a host language, which requires significant heavy lifting behind the scenes (for example, retrieving type information [Najd et al. 2016]). Najd et al. propose that "Rather than building a special-purpose tool for each QDSL, it should be possible to design a single tool for each host language." In this paper, we realize this vision for Scala: we present a quasiquote-based metaprogramming framework that simplifies the deep embedding of DSLs and the design of associated program transformations.

The practice of deeply embedding DSLs in host languages, exemplified by the polymorphic embedding approach [Hofer et al. 2008], requires to encode each DSL feature in the host language as a special data type. This translates into a lot of boilerplate, especially when associated with the burden of defining a suitable interface for DSL users, and it reduces the flexibility of the DSL design and implementation process. In contrast, we propose a system where quasiquotes are used both as the front-end for DSL users and the tool used by DSL developers to describe their domain-specific optimizations. This means DSL designers can *immediately* use the shallow interface of their DSL (i.e., defined as a simple library in the host language), and apply custom analyses and rewritings on it without the need for a dedicated deep representation.

7.3 Cross-Stage Persistence

Cross-Stage Persistence (CSP) has been an important design consideration in MetaML. CSP allows a value defined in some stage to be persisted to a further stage. In practice, CSP does not work well in real-world language implementations [Kiselyov 2017], where there is no clear semantics for persisting non-serializable local values (such as mutable references). Squid simply makes a distinction between statically-accessible symbols, such as classes, modules and methods, and *local* values. References to the latter cannot be directly persisted, and they must be serialized manually.

7.4 Type-Safe Code Manipulation

Approaches focusing on staging usually do not permit the inspection of existing code (the *purely generative* approach), or lose well-typed and well-scoped guarantees while doing so, like in LMS [Rompf and Odersky 2010]. While purely generative staging is more powerful than one may think, especially when coupled with effects [Kameyama et al. 2014], our experience in using these and related systems is that code analysis and transformation using inspection is easier to write and understand, especially for complex analyses.

Guarantees about manipulated programs have been encoded via the host language's type system using techniques such as Generalized Algebraic Data Types (GADTs) [Hofer et al. 2008; Rompf and Odersky 2010], Higher-Order Abstract Syntax (HOAS), applicative functors and monads [Kameyama et al. 2014] or static De Bruijn indices [Carette et al. 2009; Sheard et al. 2005]. However, these are often heavyweight and impose a considerable cost on domain experts, who have to deal with advanced type system features, when they would just like to express code transformations as simple rewrite rules. We found that GADTs are particularly hard to manipulate in systems like Haskell and Scala [Giarrusso 2013]. *Type-based embedding* systems like LMS [Rompf and Odersky 2010] use implicit conversions to compose code fragments, but this is not applicable to code pattern-matching.

Acknowledgments

We thank the people who helped develop Squid one way or the other, in particular Vojin Jovanovic and Eugene Burmako. This work was supported by NCCR MARVEL of the Swiss National Science Foundation.

References

H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams, IV, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. 1991. Revised4 Report on the Algorithmic Language Scheme. *SIGPLAN Lisp Pointers* IV, 3 (July 1991), 1–55.

Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. 2010. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*. IEEE, 169–178.

Alan Bawden et al. 1999. Quasiquotation in Lisp. In *PEPM*. Citeseer, 4–12.

Eugene Burmako. 2017. Scala Meta. http://scalameta.org/. (2017). Accessed: 2017-07-20.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 05 (2009), 509–543.

James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-integrated Query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 403–416.

Oege de Moor and Ganesh Sittampalam. 2001. Higher-order matching for program transformation. *Theoretical Computer Science* 269, 1-2 (2001), 135–162.

Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *ACM Sigplan Notices*, Vol. 28. ACM, 237–247.

Steven E Ganz, Amr Sabry, and Walid Taha. 2001. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 74–85.

Paolo G. Giarrusso. 2013. Open GADTs and Declaration-site Variance: A Problem Statement. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 5, 4 pages.

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *Proceedings of the 7th international conference on Generative programming and component engineering*. ACM, 137–148.

Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-Yang: Concealing the Deep Embedding of DSLs *(GPCE 2014)*. ACM, 73–82.

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2014. Combinators for Impure Yet Hygienic Code Generation. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM '14)*. ACM, New York, NY, USA, 3–14.

Andrew Kennedy and Claudio V. Russo. 2005. Generalized Algebraic Data Types and Object-oriented Programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 21–40.

Oleg Kiselyov. 2017. MetaOCaml – an OCaml dialect for multi-stage programming. (2017).

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*. ACM, New York, NY, USA, 151–161.

Geoffrey Mainland. 2007. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. ACM, New York, NY, USA, 73–82.

Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework *(SIGMOD '06)*. ACM, 706–706.

Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*. ACM, 117–120.

Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. 2016. Everything Old is New Again: Quoted Domain-specific Languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2016)*. ACM, New York, NY, USA, 25–36.

Martin Odersky and Adriaan Moors. 2009. Fighting bit Rot with Types (Experience Report: Scala Collections). In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (Leibniz International Proceedings in Informatics (LIPIcs))*, Ravi Kannan and K. Narayan Kumar (Eds.), Vol. 4. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 427–451.

Bruno C d S Oliveira and William R Cook. 2012. Extensibility for the Masses. In *European Conference on Object-Oriented Programming*. Springer, 2–27.

Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Quoted Staged Rewriting: a Practical Approach to Library-Defined Optimizations. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. ACM.

Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, Vol. 23. ACM, 199–208.

Morten Rhiger. 2012. Hygienic quasiquotation in scheme. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*. ACM, 58–64.

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming and Component Engineering*. 127–136.

Denys Shabalin, Eugene Burmako, and Martin Odersky. 2013. *Quasiquotes for Scala*. Technical Report.

Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1907–1922.

Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic. 1999. DSL Implementation Using Staging and Monads. In *Proceedings of the 2Nd Conference on Domain-specific Languages (DSL '99)*. ACM, New York, NY, USA, 81–94.

Tim Sheard, James Hook, and Nathan Linger. 2005. GADTs+ extensible kinds= dependent programming. (2005).

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 1–16.

James Stanier and Des Watson. 2013. Intermediate Representations in Imperative Compilers: A Survey. *ACM Comput. Surv.* 45, 3, Article 26 (July 2013), 27 pages.

Donald Syme. 2006. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution, In Proceedings of the 2006 workshop on ML. *Proceedings of the 2006 workshop on ML*.

Walid Taha. 2004. *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter A Gentle Introduction to Multi-stage Programming, 30–50.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242.