Unifying Analytic and Statically-Typed Quasiquotes

LIONEL PARREAUX, EPFL, Switzerland
ANTOINE VOIZARD, University of Pennsylvannia, USA
AMIR SHAIKHHA, EPFL, Switzerland
CHRISTOPH E. KOCH, EPFL, Switzerland

Metaprograms are programs that manipulate (generate, analyze and evaluate) other programs. These tasks are greatly facilitated by quasiquotation, a technique to construct and deconstruct program fragments using quoted code templates expressed in the syntax of the manipulated language. We argue that two main flavors of quasiquotes have existed so far: Lisp-style quasiquotes, which can both construct and deconstruct programs but may produce code that contains type mismatches and unbound variables; and MetaML-style quasiquotes, which rely on static typing to prevent these errors, but can only construct programs. In this paper, we show how to combine the advantages of both flavors into a unified framework: we allow the construction, deconstruction and evaluation of program fragments while ensuring that generated programs are well-typed and well-scoped, a combination unseen in previous work. We formalize our approach as $\lambda^{\{\}}$, a multi-stage calculus with code pattern matching and rewriting, and prove its type safety. We also present its realization in Squid, a metaprogramming framework for Scala, leveraging Scala's expressive type system. To demonstrate the usefulness of our approach, we introduce *speculative rewrite rules*, a novel code transformation technique that makes decisive use of these capabilities, and we outline how it simplifies the design of some crucial query compiler optimizations.

CCS Concepts: • **Software and its engineering** → **Language features**; **Semantics**;

Additional Key Words and Phrases: quasiquotes, metaprogramming, rewriting, static typing

ACM Reference Format:
Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying Analytic and Statically-Typed Quasiquotes. *Proc. ACM Program. Lang.* 2, POPL, Article 13 (January 2018), 33 pages. https://doi.org/10.1145/3158101

1 INTRODUCTION

In this paper, we tackle the problem of statically typing *metaprograms*: programs that construct, deconstruct, rewrite and evaluate other programs. Well-typed metaprograms should not "go wrong," and in particular they should not run into type mismatches and unbound variable errors at runtime, which could arise from erroneous combinations of these operations.

We describe our solution in Squid,[1] a Scala macro library that makes use of structural subtyping and intersection types to express the contextual dependencies of program fragments. We also formalize the essence of Squid as $\lambda^{\{\}}$, a multi-stage calculus with support for code pattern matching and rewriting, and prove its soundness as progress and type preservation. This work was motivated by the needs of real-world metaprogramming applications; we give several use cases inspired by our work on query compilation, and describe *speculative rewrite rules*, a new optimization pattern enabled by our support for flexible open code manipulation.

[1] The Squid type-safe metaprogramming framework is open source, accessible online at https://github.com/epfldata/squid/.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/1-ART13
https://doi.org/10.1145/3158101

1.1 Basics of Quasiquotation

Our solution is based on code quasiquotation — or just *quasiquotes*: quoted code templates that offer a way for program fragments to be composed (put together to form bigger programs) and decomposed (inspected and broken down into smaller parts). As a basic example, in Squid the value denoted by `code"2 + 2"` is *not* a string of characters, but an abstract syntax tree (AST) representing the expression `2 + 2`; it can be viewed as syntactic sugar for manually constructing an AST — in explicit pseudo-syntax `IntAdd(Const(2),Const(2))`. Within these quotes, it is possible to leave holes (also called *unquotes*, or *antiquotes*) to be filled in later. Holes are written `${...}` or `$id` where `id` is an identifier. In expressions, holes enable code *insertion*: they are substituted with the provided code values. For example, in a context where `x = code"2"`, expression `code"2 + $x"` is equivalent to `code"2 + 2"`. In patterns, holes enable code *extraction*: they pull code values out of the matched programs, making the result available to the right-hand side of the corresponding pattern matching branch. As an example of code pattern-matching,[2] `code"print(27+1)" match { case code"print($x)" => x }` *extracts* the code fragment passed to `print` in the original program, and thus evaluates to `code"27+1"`.

In this work, we focus on the quasiquotation of code in the same language as the host language (the language in which code manipulation is done — here Scala). This is not an important restriction in practice, as long as the host language is powerful enough to express the programs we want to manipulate. For example, many domain-specific languages (DSL) have been successfully embedded in expressive languages with a flexible syntax such as Haskell [Axelsson et al. 2010; Hudak 1996; Najd et al. 2016] and Scala [Lee et al. 2011; Ofenbeck et al. 2013; Rompf and Odersky 2010].

1.2 Early Example of Rewriting

To motivate the rest of this paper, we now detail a more substantial example. Consider the problem of finding all local variables that hold a pair of values `(a,b)`, removing these variables and rewriting their uses into direct accesses to `a` and `b`. The following Scala program uses Squid's **`rewrite`** primitive to traverse a `pgrm` term bottom up matching any variable `p` bound to a pair of integers; in the scope `body` of each such binding, it replaces projections to `p`'s first and second components (syntax `p._1` and `p._2`) with the corresponding pair element `a` or `b`:

```
pgrm.rewrite { case code"val p: (Int,Int) = ($a,$b); $body" =>
  val body2 = body.rewrite { case code"${body.p}._1" => a  case code"${body.p}._2" => b }
  body2.p ~> code"($a,$b)"                                                              }
```

The syntax is explained in detail in later sections. For now, what is important to see is that our system statically keeps track of the fact that variable `p` is *free* in program fragments `body` and `body2` (i.e., these terms are "open in `p`"). Syntax `body.p` is used to refer to that unbound variable. In patterns, `body.p` matches any free occurrences of `p`. Syntax `body2.p ~> x` returns the *substitution* in `body2` of all occurrences of that free variable with provided code fragment `x`. This is used to replace all remaining occurrences of `p` (if any) with an in-place reconstruction of the original pair. For example, program `code"val my = (1,2+2); print(my); my._1 + my._2"` is first rewritten into `body2 = code"print(my); 1 + (2+2)"`, and then into `code"print((1,2+2)); 1 + (2+2)"`.

Squid is hygienic because it will not mix up the matched binding `p` with bound variables present in the original program `pgrm`, even if they also happened to be named `p`. In addition, forgetting to substitute `p` in `body2` at the end of the rewriting will result in a type error, reported at compilation time — otherwise, our rewriting could result in programs with unbound references to `p`.

Before we delve further into the description of our system, we present some background on quasiquotation necessary to situate our contributions.

[2] Scala expression `s match {case p => e}` corresponds to SML's `case s of p => e` or Caml's `match s with p -> e`.

1.3 Analytic Quasiquotes

Code quasiquotes have been present in research and industry under two main flavors [Ganz et al. 2001], which we will refer to as the ***Analytic*** and ***Statically-Typed*** flavors. Analytic quasiquotes were pioneered in the context of Lisp, where source code is essentially made of arbitrarily-nested lists and symbols (S-expressions), and programs manipulate data structures encoded using the same format. For example, function `(lambda (x) (print x))` can be represented with datum `` `(lambda (x) (print x)) `` — the "back-tick" at the beginning indicates the start of a quasiquotation, to distinguish it from plain Lisp source code. Therefore, programs can naturally manipulate source code like any other data structure (code as data). In addition, the built-in `eval` function is used to interpret any datum as source code by executing it (data as code). Antiquotation in Lisp is written with a comma, so expression `` `(lambda (x) ,(id `x)) `` seen in Table 1 first executes the identity function `id` on `` `x `` returning `` `x ``, then places that code fragment into a bigger program, constructing an implementation of the identity function `` `(lambda (x) x) ``.

The fact that Lisp programs can also analyze (inspect) source code derives directly from the idea of code as data. Allowing quasiquotes in *pattern matching* is one convenient way to do it, and is in fact standard in several dialects of Lisp, including Scheme. This is what motivates our terminology: these quasiquotes have *analytic* capabilities.

1.4 Statically-Typed Quasiquotes

Lisp is *dynamically typed*, which means that it does not statically prevent the occurrence of type mismatches and unbound variable references at runtime. For example, since `x` is a valid Lisp program, `(eval `x)` is also valid but raises a runtime error in the style of "`x is undefined and cannot be evaluated`" unless it is executed in a context where some `x` is defined. In contrast, in a *statically typed* programming language, type mismatches and undefined variable errors are never supposed to happen at runtime. This makes the notions of code as data and data as code significantly harder to satisfy. Indeed, we must make sure that program fragments containing unbound references are never evaluated, and that all constructed programs are well-typed. We must reject programs such as `code"x".run` (where method `run` has the same functionality as `eval` in Lisp). With MetaML, Taha and Sheard [2000] introduced statically-typed quasiquotes, allowing the expression of type-safe program generators that cannot generate ill-typed or ill-scoped programs. In MetaML, the quotation ⟨`x+1`⟩ (similar to Lisp's `` `(+ x 1) ``) is only valid if it is surrounded by a code fragment *at the same quotation depth* containing a binder for `x` with type `int`, so that `x+1` will end up in a place where `x` is bound. Together with antiquotes, written `~(...)`, we can rewrite in MetaML the Lisp example we just saw, as ⟨`fun x → ~(id ⟨x⟩)`⟩, which evaluates to ⟨`fun x → x`⟩. MetaML historically faced some challenges. The first was to statically prevent the evaluation of open code (code that contains variables which have not yet been bound). For example, ⟨`fun x → ~( run ⟨x⟩ ; ⟨x⟩ )`⟩ should be rejected: ⟨`x`⟩ cannot be run as it has not yet been inserted in a context where `x` is bound. Another challenge was that of *scope extrusion*, where a piece of code *escapes* its enclosing scope by ways of imperative features such as mutable references, as in ⟨`fun x → ~( some_ref := ⟨x⟩ ; ⟨x⟩ )`⟩. Two general approaches have been proposed to solve these problems; the first makes use of contextual types [Rhiger 2012b], where the environments that terms depend on are reflected in their types; the second uses *environment classifiers*, which abstract over these contexts using partially-ordered type variables [Kiselyov et al. 2016; Taha and Nielsen 2003].

MetaML quasiquotes are not as expressive as analytic quasiquotes like those of Lisp, because constructed code is viewed as a black box and cannot be inspected; quasiquotes cannot be used in patterns,[3] and it is not possible to express program rewritings like the one we saw in Section 1.2.

[3] Quasiquote patterns for MetaML were suggested by Sheard et al. [1999], but were neither implemented nor formalized.

	Typ	Scp	Ana	Hyg	Syntax Example
Squid (This Paper)	●	●	●	●	`code"(x: T) => ${ id(code"?x:T") }"`
Squid, old version (0)	●	◐	◐	●	`code"(x: T) => ${ id(_:Code[T]) }(x)"`
Scala-reflection QQ (1)	○	○	●	○	`q"(x: T) => ${ id(q"x") }"`
Scala-refl. `reify/splice`	◐	○	○	●	— cannot express open terms —
MetaOCaml (2)	●	◐	○	●	`.< fun x → .~( id .< x >. ) >.`
Template Haskell (3)	○	◐	○	◐	`[| \x -> $( id [| x |] ) |]`
Typed Template Haskell	●	●*	○	●*	`[|| \x -> $$( id [|| x ||] ) ||]`
Stratego (4)	○	○	●	○	`[[ (x: int) => ~( id ([[ x ]]) ) ]]`
Lisp/Scheme QQ (5)	○	○	●	○	`` `(lambda (x) ,( id `x )) ``
$\lambda^{\{\}}$ (This Paper)	●	●	●	●	$\lceil \lambda x : T.\ \lfloor \mathrm{id}\ \lceil x : T \rceil \rfloor \rceil$
MetaML Calculus (7)	●	◐	○	●	$\langle\ \lambda x.\ \sim(\mathrm{id}\ \langle\ x\ \rangle)\ \rangle$
Rhiger's $\lambda^{[\,]}$ (6)	●	●	○	●	$\uparrow(\ \lambda x : T.\ \downarrow(\ \mathrm{id}\ \uparrow x\ )\ )$
Nanevski's $v^{\Box}$ (8)	●	●	◐	●	$\mathbf{let\ box}\ u = \mathit{id}\ X\ \mathbf{in\ box}\ \lambda x.\ \{\ X \doteq x\ \}\ u$

Table 1. Comparison of quasiquotes in several systems. Criteria: statically ensure program fragments are *well-typed* and *well-scoped* (columns *Typ* and *Scp*); support *analysis* via pattern-matching (*Ana*); support *hygiene* (*Hyg*); see Sections 6.1, 6.2 and 6.3 for a full discussion. References: (0) Parreaux et al. 2017b; (1) Shabalin et al. 2013; (2) Taha 2004; (3) Sheard and Jones 2002; (4) Visser 2002; (5) Bawden et al. 1999; (6) Rhiger 2012a; (7) Taha and Nielsen 2003; (8) Nanevski and Pfenning 2005.

1.5 Best of Both Worlds

Quasiquotes that belong to the analytic category include those of the Lisp family [Bawden et al. 1999], Stratego [Visser 2002] and Scala reflection [Shabalin et al. 2013]. The safer, statically-typed category includes the quasiquotes of MetaML [Taha and Sheard 2000], MetaOCaml [Taha 2004], MacroML [Ganz et al. 2001] and Typed Template Haskell, a variant of Template Haskell [Sheard and Jones 2002]. Table 1 summarizes the properties of these systems, and is discussed further in Section 6. In this paper, we show how to combine the advantages of both flavors into a unified framework. We allow the construction and inspection of code fragments while ensuring that generated code is always well-typed and well-scoped, and we demonstrate the use of code rewriting to define type-safe domain-specific optimizations. The contributions we make are organized as follows:

- In Section 2 we explain how Squid approaches the problem of open code manipulation, and we introduce *speculative rewrite rules*, a novel use of quasiquotes to design type-safe program transformations. As an example, we present a data structure optimization.
- In Section 3 we distill the essence of Squid as $\lambda^{\{\}}$, a multi-stage calculus that can analyze and rewrite code fragments. We formalize the static and dynamic (big-step) semantics of $\lambda^{\{\}}$, and we prove *type preservation* and *progress*.
- In Section 4 we detail how Squid is embedded in Scala, a modern object-oriented and functional programming language. In particular, we show how to leverage Scala's advanced type system, and how to abstract over variable names and context requirements in a type-safe way.
- In Section 5, we detail some real-world use cases, explaining how Squid facilitates the design and robust implementation of two query compiler optimizations.

Note that we focus solely on *expression* quasiquotes, which cannot directly manipulate module, class and method definitions. This restriction is similar to other staging frameworks, such as MetaML [Taha and Sheard 2000], MetaOCaml [Kiselyov 2014], and LMS [Rompf and Odersky 2010].

2 PRESENTATION OF SQUID AND APPLICATION EXAMPLES

Making sure that bindings are manipulated correctly so that program transformations do not result in unintentional capture (when bindings with the same names interfere; a.k.a., lack of hygiene) or ill-scoped code (when variable references are extruded from their binders' scopes) is a notoriously hard and subtle problem. In this section, we describe and exemplify how this is done in Squid.

2.1 Handling of Open Code in Squid

The way Squid allows the type-safe manipulation of open program fragments is twofold:

- Represent free variables explicitly: instead of using the type system to resolve variable references across quotation boundaries like in MetaML, we require users to explicitly declare which variables are free in a given quasiquote by prepending a '?' to their names.[4] The example given in Table 1 can be written in Squid `code"(x: Int) => ${id( code"?x : Int" )}"`. The main advantage is that it renders the composition of code more modular: an open code fragment can be written out without having bindings for all its free variables syntactically surrounding it; i.e., Squid quasiquotes do not have to be lexically scoped. In the previous example, we can extract the inner expression into a let binding: `val inner = id(code"?x: Int"); code"(x: Int) => $inner"`, which does not work in MetaML. Allowing violations of lexical scoping based on explicit annotations is related to the approach taken by Kim et al. [2006]. This property also translates into welcome practical benefits: it integrates well with Scala's local type inference, and allows us to implement the entire quasiquote system as a macro library, requiring no modifications to Scala's type checker.
- Reflect context requirements in the type of code fragments: in order to track which fragments are open and when their free variables are to be captured, as well as to decide when code is safe to be evaluated, we make the types of quoted terms directly reflect their *context requirements*, i.e., the names and types of the free variables contained in those terms. For instance, fragment `val fx = code"(?x: Int) + 1"` has type `Code[Int,{x:Int}]`, which means that it represents a term of type `Int` that needs to be inserted in some context where a variable `x` of type `Int` is defined. By composition, `code"$fx.toDouble"` has type `Code[Double,{x:Int}]` as it is equivalent to `code"((?x:Int)+1).toDouble"`. The free variables contained in a term that is inserted into some quotation context are correctly captured by the variables bound in said context: `code"val x = 0; $fx.toDouble"` has type `Code[Double,{}]` and evaluates to `code"val x = 0; (x+1).toDouble"` (or, equivalently, `code"val y = 0; (y+1).toDouble"`).

Quasiquote patterns may be used to match bindings, and extracted subterms will have types that reflect and track their potential dependencies to these bindings; e.g., using → to denote evaluation:

```
val f = code"(x: Int) => x + (?y:Int)"           : Code[Int=>Int,{y:Int}]  → code"(x:Int) => x + ?y"
val g = f match {case code"(z => $body)" => body}: Code[Int,{y:Int;z:Int}] → code"?z + ?y"
```

Several things should be noted here: First, `code` quasiquotes integrate well with Scala's type inference, as the type of the scrutinee `f` propagates to help type the pattern in `g`. If `f` had type `Code[Any,_]` instead, we would have to write the pattern as `case code"(z: Int) => $body: Int"`. Second, the names of bound variables do not matter, and a lambda that used `x` as the parameter name can be matched as if it were using `z` instead. Third, the type of extracted fragment `body` reflects that it may contain free variables from two different sources: by propagation from the scrutinee's type we know it

[4] We could avoid the '?' and view all unqualified names as free variables, but this would be a bad ergonomic choice: typos could easily result in confusing errors, and we do not want for e.g., `code"print(1)"` to be interpreted as `code"(?print)(1)"`.

may refer to some `y`, and because the pattern introduced a binding named `z` it may as well refer to it — what happened is that `body` was *safely extruded* from its enclosing context `{z:Int}`.

2.2 Rewrite Rules and Polymorphism

To help with the definition of safe rewritings, Squid provides a **`rewrite`** method that traverses a program and applies a transformation while checking at compile-time that the transformation preserves the type and context of each sub-terms. Additionally, in order to define type-parametric rewrite rules, Squid allows the extraction of types, along with terms. In the example below, given some `pgrm` fragment we transform calls to `foldLeft` on `List` objects into imperative `foreach` loops:

```
pgrm rewrite { case code"($ls:List[$t]).foldLeft[$r]($init)($f)"
                 => code"var cur = $init; $ls.foreach(x => cur = $f(cur, x)); cur" }
```

For example, if `pgrm = code"List(1,2).foldLeft(0)((acc,x) => acc+x) + 1"`, the rewriting returns `code"var cur = 0; List(1,2).foreach(x => cur = cur+x); cur + 1"` (the β-redex is removed by Squid's internal normalization). Note that in Scala partial functions are written `{ case ... => ... }`; they are similar to pattern matching, but they need not be exhaustive. Operator syntax '`p rewrite f`' is the same as '`p.rewrite(f)`'. Notice that we extract a type `t` that is never used explicitly — it is in fact inferred as part of the type of the quoted program on the right-hand side of the rewriting case.[5]

2.3 Fixed Point Rewritings

Rewritings can be applied over and over again until they reach a fixed point. In Squid, rewriting `Math.pow(x,2)` into `x * x` is trivially expressed, but let us here consider the generalization of that problem to arbitrary exponents. We define below a fixed point rewriting that uses binary exponentiation to transform calls to `Math.pow` with a constant integer exponent into a series of multiplications. `Const` is the constructor for constants, used to lift current-stage values as code constants and extract constant values from code fragments. For instance, pattern `code"pow($x,${Const(exp)})"` extracts `x` as a code value of type `Code[Double,_]`, but it extracts `exp` as a "bare" value of type `Double`. Method `isWhole` from class `Double` is used to query whether a floating-point number has an integral value.

```
import Math.pow
pgrm fix_rewrite {
  case code"pow($x,$exp)"  if !x.isTrivial  =>  code"val base = $x; pow(base,$exp)"
  case code"pow($x,0)"                      =>  code"1.0"
  case code"pow($x,${Const(exp)})"  if exp.isWhole && exp > 0
    => if (exp % 2 == 0) code"val tmp = pow($x,${Const(exp/2)}); tmp * tmp"
       else              code"$x * pow($x,${Const(exp-1)})"                          }
```

The role of the first **`case`** rule, which is applied first, is to let-bind the base `x` passed to `pow` if it is not "trivial" i.e., unless it is a constant or a variable reference. This avoids code duplication that would otherwise result from the following rules.[6] For example, `pow(.5,3)` is rewritten into `0.5*{val tmp_0 = 0.5*1.0; tmp_0*tmp_0}`, duplicating `0.5`, but `pow(readDouble,3)` (where `readDouble` reads a number from standard input) is rewritten into `val x_0 = readDouble; x_0*{val tmp_1= x_0*1.0; tmp_1*tmp_1}`.

[5] This named pattern variable is necessary (`$t` cannot be replaced with `$_`) because Squid needs to generate a local, named type symbol representing the extracted type. This is explained further in Section 4.1.

[6] Note that Squid can be used with different underlying intermediate representation [Parreaux et al. 2017b]; by using an appropriate representation (such as the A-normal form), such code duplication concerns disappear, as we show in [2017a].

```
def optimize[T] (pgrm: Code[T,{}]) : Code[T,{}]  =  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}($i) = ($va, $vb)"  =>  code"$a($i) = $va; $b($i) = $vb"
      case  code"${body.arr}($i)"               =>  code"($a($i), $b($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"  }
```

Fig. 1. A speculative rewrite rule for transforming any array of pairs into two separate arrays.

2.4 Free Variables and Substitution

As explained in the introduction, given some variable `x` free in `t`, we can refer to that free variable with syntax `t.x`, and we can replace all its occurrences in `t` with syntax `t.x ~> y`, as in the following:

```
val a = code"(?x: Int) + 1" : Code[Int, {x: Int}]  →  code"?x + 1"
val b = a.x                 : Code[Int, {x: Int}]  →  code"?x"
val c = a.x ~> code"27"     : Code[Int, {}]        →  code"27 + 1"
```

In fact, the `t.x` construct is equivalent to using an explicit free variable in a quasiquote: above, `a.x` is equivalent to `code"?x:Int"`. Note that the right-hand side `y` of `t.x ~> y`, is evaluated *lazily* (i.e., only if there are actual instances of `x` left in `t`), a property that proves useful in the next sections.

2.5 Speculative Rewrite Rules

The current innermost rewriting can always be aborted by calling `abort()` at any point in the right-hand side of the rewriting case. This call never returns and passes the control back to the rewriting engine.[7] We call *speculative rewrite rules* those rewrite rules that attempt to apply a transformation but abort that transformation as soon as they find something that should have prevented it from applying in the first place. In other words, speculative rewrite rules are a convenient and type-safe way to express conditional rewritings without having to define separate, error-prone analysis passes over the program one wants to transform. In the next section, we present an example of speculative rewrite rule and explain how Squid ensures the safety of such transformers. Another example, directly extracted from our work on query compilation [Shaikhha et al. 2016], is given in Section 5.1.

2.6 Motivating Example: Array of Tuples Optimization

Figure 1 presents an example of speculative rewrite rule that attempts to turn any array of 2-tuple elements into two distinct arrays. This optimization is sometimes known as "array-of-structs to struct-of-arrays," and has particular relevance in the field of databases (see Section 5.2); its goal is to streamline array accesses, making them more cache-friendly for the processor, and to avoid

[7] This mechanism is similar to delimited continuations [Danvy and Filinski 1990], where '`case pattern => ...`' acts like `(reset (shift c ...))` and `abort()` acts like a short-circuiting `(c ())`. This is implemented on the JVM using exceptions.

the performance cost of allocating tuples.[8] A trace of the successive values taken by each variable, given a dummy input `pgrm`, is shown below:

```
pgrm   = code"if (readInt > 0) { val a = new Array[(Int,String)](3); a(0) = (36,"ok"); a.size }"
size   = code"3"
body   = code"(?arr)(0) = (36,"ok"); (?arr).size"
body2  = code"(?a)(0) = 36; (?b)(0) = "ok"; (?a).size"
body3  = body2
result = code"val a = new Array[Int](3); val b = new Array[String](3); a(0) = 36; b(0) = "ok"; a.size"
optimize(pgrm)  →  code"if (readInt > 0) { $result }"
```

To understand this example, three key properties of Squid should be noted: 1. holes at the end of a list of statements can be viewed as matching the input greedily; for instance, pattern `code"print(42); $b"` will match a `print` statement and all following statements in the current block;[9] 2. as shown in the introduction, qualified references to free variables (here `body.arr`) can be used even from within a pattern unquote — remember that this is just syntax sugar, `code"${body.arr}($i)"` being equivalent to `code"(?arr:Array[T])($i)"`; and 3. free variables in patterns will match free variables in program fragments. All statements following the array binding that is matched in the original program are captured into `body`; they are extruded from their enclosing context, and their references to the bound array are transformed into references to the free variable `arr`. In the inner rewriting, we then match references to `arr` to transform usages of the array as they existed in the original program. Note that this process is hygienic: as it traverses a program, the **`rewrite`** method *only* extrudes bindings that are matched explicitly in rule patterns, and therefore there is no risk of encountering free variables also named '`arr`' that referred to different bindings than the one we matched.

Once the inner rewriting has been applied, the result `body2` contains a program fragment where patterns like `arr(i)._1` have been replaced with expressions referring to free variables `a` and `b` — in this case, `(?a)(i)`. All remaining references to `arr` are then searched for using the free variable substitution syntax (which evaluates its second argument lazily), and if any is found the rewriting is aborted. Notice that in the inner rewriting, we deliberately do not handle patterns of the form `code"${body.arr}($i) = $v"` where `v` is *not* of the form `(x,y)`. As a consequence, arrays used in such a way are not transformed. The rationale is that if the original program stored already-tupled values into the array, then perhaps it is not a good idea to do the transformation: it may lead to *more* allocation rather than less.[10] Finally, note that while the optimization in Figure 1 is defined for tuples of two elements only, applying it until it reaches a fixed point will also transform arrays of tuples of more elements, as long as an inductive encoding of tuples is used — for example, `(a,b,c,d)` could be encoded as a composition of nested 2-tuples such as `(a,(b,(c,d)))`.

To get a sense of how Squid's type system and contextual code types help us avoid runtime errors, let us look at some programming mistakes that could be made while writing the transformer:

- omitting to insert `$body3` in the result code fragment: this would give result type `Unit` (i.e., `void` in Scala) and the **`rewrite`** method would complain that the rewriting is not type-preserving;
- using the wrong array in the inner rewriting — for example writing `code"$a($i)"` instead of `code"$b($i)"`: the inner **`rewrite`** will complain that this case tries to rewrite a term of type `ta` to a term of type `tb` and reject it (fail to compile), as above;

[8] The JVM stores composite objects such as tuples using an additional level of indirection (boxing), which is removed if we store each field of the tuple in a separate array.
[9] A natural consequence of the inductive representation of program statements, i.e., `{a;b;c}` is equivalent to `{a;{b;{c}}}`.
[10] In a real-world setting, a more precise analysis with heuristics could determine whether or not to apply the rewriting.

- forgetting to define one of `a` or `b` in the final result of the main rewriting: the result type inferred for the whole rewriting will be[11] `Code[T,{a:Array[_]}]` instead of `Code[T,{}]` (the return type explicitly specified for `optimize`), resulting in a type mismatch;
- using `body2` directly instead of `body3`, without performing the variable substitution: a similar error as above will be raised, as variable `arr` is still assumed to be free in `body2`, propagating its requirement to the result type of the outer rewriting expression;
- forgetting to transform some array operations: this will simply abort the rewriting and leave arrays with unexpected usages untouched, ensuring the safety of our transformer; this case includes when the array "escapes" the current scope, being sent to unknown functions;
- trying to evaluate the `size` program fragment with `size.run`: this results in a compile-time error that reads "`cannot prove that <context @ 2:7> =:= {}`" (explained in Section 4.3);
- accessing `size` from outside of the rewriting (e.g., via a mutable variable): since the context requirement of `size` is only defined inside the right-hand side of the rewrite rule, `size`'s type becomes `Code[Int,_]` when viewed from the outside, making the term impossible to close.

3 FORMALIZATION OF THE CORE LANGUAGE

This section presents the core ideas of our design of statically-typed analytic quasiquotes, demonstrating them independently of their Scala embedding. $\lambda^{\{\}}$ is a call-by-value multi-stage λ-calculus with two types of pattern matching on code values: **`match`** simply decomposes a term against a pattern, and **`rewrite`** traverses a term bottom-up, applying some transformation on the way.

3.1 Syntax

The syntax of $\lambda^{\{\}}$ is given in Figure 2. α is a meta-meta-variable that ranges over meta-variables, and is used to parametrize the production of q. For now, we only use $q_{\lfloor u \rfloor}$, but in Section 3.3 we will use a variation. $\overline{\alpha}$ denotes 0 to n repetitions of 'α' separated by semicolons. $\mathbf{let}\ x : T = t_0\ \mathbf{in}\ t_1$ is syntactic sugar for $(\lambda x : T.\ t_1)\ t_0$. To simplify the development, we make the usual assumption that α-renaming is used whenever needed to prevent shadowing: a context never contains two distinct bindings $x : T$ and $y : S$ such that $x = y$. This allows us to equate contexts with finite partial functions from variable names to types. For example, we use $\emptyset$ for the empty context $\{\}$ and $\Gamma \cup \Gamma'$ for context extension. Type ascriptions are used to disambiguate types when necessary.

Examples. As a first example of a $\lambda^{\{\}}$ program, we give below a simple optimization that transforms an expression of the form `pow x 2` into `x * x`. We assume the existence of constants '`pow`' and '$*$' for integer power and multiplication, respectively:

$$\lambda x : \mathbf{Code\ Int}\ \emptyset.\ \ x\ \mathbf{match}\ \lceil \mathrm{pow}\ \lfloor y \rfloor\ 2 \rceil \Rightarrow \lceil \mathbf{let}\ z = \lfloor y \rfloor\ \mathbf{in}\ z * z \rceil\ \mathbf{else}\ x$$

The function above takes a code value x and pattern-matches it against the power-of-2 pattern, binding the program fragment extracted as the base to variable y. If the pattern matches, a program is returned that consists in the binding of the code value represented by y to some variable z, that is then multiplied with itself (this avoids duplicating the computations potentially contained in y). If the pattern does not match, the original code value x is simply returned unchanged.

In $\lambda^{\{\}}$, free variables present in quoted terms do not require a special syntax, so for example **`code`**`"(?x:Int)+1"` is written just $\lceil x + 1 \rceil$. The $\mathbf{close}_x$ construct makes sure that a term contains no free variable x, otherwise defaulting to the associated **`else`** branch. To illustrate the use of open terms and show a speculative rewrite rule, we take inspiration from the rewriting of Figure 1,

[11] In Scala, an underscore in type position stands for an existential: `Array[_]` stands for `Array[t] forSome { type t }`.

q_α	$::=$	*Parametrized Term*	t	$::= q_{\lfloor u \rfloor}$	*Term*
	n	*Integer Literal*	u	$::=$	*Unquotable*
	$\mid t + t$	*Addition*		x	*Variable*
	$\mid x, y, z$	*Variable*		$\mid$ **const** x	*Lifted Constant*
	$\mid t\ t$	*Application*	T, S	$::=$	*Type*
	$\mid \lambda x : T.\ t$	*Lambda Abstraction*		**Int**	*Integer*
	$\mid \lceil q_{\lfloor u \rfloor} \rceil$	*Quote*		$\mid T \to T$	*Function*
	$\mid$ **run** t	*Code Evaluation*		$\mid$ **Code** $T\ C$	*Code*
	$\mid t$ **match** $\lceil t \rceil \Rightarrow t$ **else** t	*Pattern Matching*	C, Γ	$::= \{\ \overline{c}\ \}$	*Context*
	$\mid t$ **rewrite** $\lceil t \rceil \Rightarrow t$	*Term Rewriting*	c	$::=$	*Context Member*
	$\mid$ **close**$_x$ t **else** t	*Speculative Closure*		$x : T$	*Binding*
	$\mid t : T$	*Type Ascription*		$\mid x$	*Context Brand*
	$\mid \alpha$	*Parametrized Rule*			

Fig. 2. Syntax of $\lambda^{\{\}}$.

which matches usages of an extruded variable `arr` and replaces them with usages of different free variables `a` and `b`. We assume the language is extended with types 'Array' and 'PairArray' (similar to `Array[Int]` and `Array[(Int,Int)]` in Scala), and with constants 'mkPairArray', 'mkArray', 'size', 'get', and 'first' with the expected semantics. An incomplete version of the rewriting of Figure 1, where we handle only two cases (namely *size* and *get-first*), is given below:

$$
\begin{aligned}
&\mathit{pgrm}\ \textbf{rewrite}\ \lceil \textbf{let}\ \mathit{arr} = \mathrm{mkPairArray}\ \lfloor n \rfloor\ \textbf{in}\ \lfloor \mathit{body} \rfloor \rceil \Rightarrow \\
&\quad \textbf{let}\ a = \lceil a : \mathrm{Array} \rceil\ \textbf{in} \\
&\quad \textbf{let}\ \mathit{body}_2 = \mathit{body}\ \textbf{rewrite}\ \lceil \mathrm{size}\ \mathit{arr} \rceil \Rightarrow \lceil \mathrm{size}\ \lfloor a \rfloor \rceil \\
&\qquad\qquad\qquad\qquad \textbf{rewrite}\ \lceil \mathrm{first}\ (\mathrm{get}\ \mathit{arr}\ \lfloor i \rfloor) \rceil \Rightarrow \lceil \mathrm{get}\ \lfloor a \rfloor\ \lfloor i \rfloor \rceil\ \textbf{in} \\
&\quad \textbf{close}_{\mathit{arr}}\ \lceil \textbf{let}\ a = \mathrm{mkArray}\ \lfloor n \rfloor\ \textbf{in}\ \lfloor \mathit{body}_2 \rfloor \rceil\ \textbf{else}\ \mathit{pgrm}
\end{aligned}
$$

The program above proceeds in much the same way as in Figure 1. For example, given some $\mathit{pgrm} = \lceil \textbf{let}\ x = \mathrm{mkPairArray}\ 3\ \textbf{in}\ \mathrm{first}\ (\mathrm{get}\ x\ 0) + \mathrm{second}\ (\mathrm{get}\ x\ 1) \rceil$, after the outer pattern matches, we get $\mathit{body} = \lceil \mathrm{first}\ (\mathrm{get}\ \mathit{arr}\ 0) + \mathrm{second}\ (\mathrm{get}\ \mathit{arr}\ 1) \rceil$, which is subsequently rewritten into $\mathit{body}_2 = \lceil \mathrm{get}\ a\ 0 + \mathrm{second}\ (\mathrm{get}\ \mathit{arr}\ 1) \rceil$, and then $\textbf{close}_{\mathit{arr}}$ is called on a term that still contains references to *arr* (as we are missing the rule to rewrite uses of `second`), aborting the rewriting as expected.

3.2 Type System

The typing rules of $\lambda^{\{\}}$ are presented in Figure 3. Typing judgment $\Gamma \vdash t : T \dashv \Gamma'$ is read "under inner context Γ and outer context Γ', t has type T" (see explanations below). Syntax $\vdash t : T$ is shorthand for $\emptyset \vdash t : T$, and $\Gamma \vdash t : T$ stands for $\Gamma \vdash t : T \dashv \emptyset$.

Quoting and unquoting. We use a "double-headed" typing judgment in order to type the term inside a quote as having its own, *inner* context (on the left), while remembering the outer context from outside the quote (on the right). The inner context is the usual typing context, accounting for free variables. Free variables in a quote may be bound by a lambda abstraction, or may remain free and become part of the quoted term's context requirements. The outer context is used to type unquotes, which refer to the context outside of the quotation. Since unquotes can only contain identifiers, they cannot be nested so we only need to carry a single outer context even though $\lambda^{\{\}}$ is a *multi-stage* language. This syntactic restriction does not incur a loss of generality, as a nested unquote such as $\lceil \ldots \lfloor f\ \lfloor x_0 \rfloor \rfloor \ldots \rceil$ can always be encoded by using an intermediate binding:

$\texttt{let}\ x_1 = f\ \lfloor x_0 \rfloor\ \texttt{in}\ \lceil \ldots \lfloor x_1 \rfloor \ldots \rceil$. Notice how in *T-Quote*, the outer context of t becomes the inner context of $\lceil t \rceil$ while the inner context becomes part of the **Code** type of $\lceil t \rceil$, and how in *T-Anti*, the context parameter of unquoted code has to coincide with the inner context of surrounding code.

Running Code. Rule *T-Run* requires the context of program fragment t in $\texttt{run}\ t$ to be empty. This is central to avoiding the occurrence of unbound reference errors at runtime. For example, the term $\texttt{run}\ \lceil x + 1 \rceil$ is not typeable, similar to how `code"(?x:Int)+1".run` is rejected by Squid.

Pattern Matching. Rule *T-Mat* needs to ensure two important properties:

- Unquotes in a pattern capture the local context surrounding them. For example, for some $z : \texttt{Code Int}\ \emptyset$ in program $z\ \texttt{match}\ \lceil \lambda x : \texttt{Int}.\ \lfloor y \rfloor + 1 \rceil\ \Rightarrow\ y\ \texttt{else}\ \lceil 0 \rceil$, the type of *extracted* variable y should be $\texttt{Code Int}\ \{\, x : \texttt{Int} \,\}$; indeed, y can be used to extract terms containing references to x (in particular, when $z = \lceil \lambda x : \texttt{Int}.\ x + 1 \rceil$ we get $y = \lceil x \rceil$). This is achieved by typing the pattern t_p with outer context Γ' (so that Γ' contains the extracted variables) and then typing the body t_b in the original context Γ extended with Γ'. Because *T-Anti* requires the unquoted variable's context parameter to *exactly coincide* with the local context surrounding the unquote, Γ' has to contain variables whose types reflect the exact context from which they were extracted. It is important for *T-Anti* not to allow widening of the unquoted variable's type: though it would make sense in expressions, it would also allow the types of terms extracted from patterns to "forget" about their local context requirements (in the previous example, y could be assigned type $\texttt{Code Int}\ \emptyset$).[12] As a result, expression $\lambda x : \texttt{Code Int}\ \emptyset.\ \lceil \lambda y : \texttt{Int}.\ \lfloor x \rfloor \rceil$ is not typeable, but this is not a practical limitation, since one can use an intermediate binding allowing subsumption (*T-Sub*) to widen the context of x as needed, as in: $\lambda x : \texttt{Code Int}\ \emptyset.\ \texttt{let}\ z : \texttt{Code Int}\ \{\, y : \texttt{Int} \,\} = x\ \texttt{in}\ \lceil \lambda y : \texttt{Int}.\ \lfloor z \rfloor \rceil$.
- Extracted variables propagate the scrutinee's own context requirements. For example, consider $t = \lceil x + 1 \rceil\ \texttt{match}\ \lceil \lfloor y \rfloor + 1 \rceil\ \Rightarrow\ y\ \texttt{else}\ \lceil 0 \rceil$, which extracts a subterm from a program fragment containing free variable x. Term t should have type $\texttt{Code Int}\ \{\, x : \texttt{Int} \,\}$ since x is free in the result $\lceil x \rceil$. This is achieved by adding the original context C of the scrutinee to the context of each extracted term in Γ', written Γ'/C and formally defined below. In the example above, $C = \{\, x : \texttt{Int} \,\}$, $\Gamma' = \{\, y : \texttt{Code Int}\ \emptyset \,\}$ and so $\Gamma'' = \Gamma'/C = \{\, y : \texttt{Code Int}\ \{\, x : \texttt{Int} \,\} \,\}$.

Definition 3.1 (Context predication Γ/C).

$$\Gamma/C \stackrel{\text{def}}{=} \{(x : f(T, C)) \mid (x : T) \in \Gamma\} \quad \text{where} \quad f(T, C) = \begin{cases} \texttt{Code}\ T'\ (C\ \cup\ C') & \text{if } T = \texttt{Code}\ T'\ C' \\ T & \text{otherwise} \end{cases}$$

Definition 3.2 (Canonical context $\dashv^* \Gamma'$). We write $\Gamma \vdash t : T \dashv^* \Gamma'$ instead of $\Gamma \vdash t : T \dashv \Gamma'$ to require that Γ' be a smallest context satisfying the typing judgment.[13]

$$\Gamma \vdash t : T \dashv^* \Gamma' \stackrel{\text{def}}{=} \Gamma \vdash t : T \dashv \Gamma' \ \wedge\ (\nexists \Gamma''.\ \Gamma \vdash t : T \dashv \Gamma'' \ \wedge\ |\Gamma''| < |\Gamma'|)$$

In rule *T-Mat*, we require pattern code t_p to be typed only with the smallest outer context possible, otherwise called *canonical context*. Indeed, by weakening, a pattern such as $t_p = \lfloor x \rfloor + 1$ can not only be typed with outer context $\{\, x : \texttt{Code Int}\ \emptyset \,\}$, but also with, e.g., $\{\, x : \texttt{Code Int}\ \emptyset;\ y : \texttt{Int} \,\}$. We have to reject the latter, as it would introduce a spurious variable y into the scope of body t_b, whereas no y was actually extracted from t_p. Intuitively, this is because when typing a pattern the

[12] Another approach could be to change the premise of *T-Anti* to $\Gamma' \vdash x : \texttt{Code}\ T\ \Gamma$ and to add a "flag" to the typing judgment that specifies whether we are typing an expression (where *T-Sub* is allowed to happen), or a pattern (where it is not).
[13] Notation $|\Gamma|$, based on the interpretation of contexts as sets, denotes the number of context members c in Γ.

Typing Rules

$$\text{T-Var}\ \frac{(x : T) \in \Gamma}{\Gamma \vdash x : T \dashv \Gamma_0} \qquad \text{T-Lit}\ \frac{}{\Gamma \vdash n : \texttt{Int} \dashv \Gamma_0} \qquad \text{T-Asc}\ \frac{\Gamma \vdash t : T \dashv \Gamma_0}{\Gamma \vdash (t : T) : T \dashv \Gamma_0}$$

$$\text{T-App}\ \frac{\Gamma \vdash t_f : T \to S \dashv \Gamma_0 \qquad \Gamma \vdash t_a : T \dashv \Gamma_0}{\Gamma \vdash t_f\ t_a : S \dashv \Gamma_0} \qquad \text{T-Lam}\ \frac{\Gamma \cup \{ x : T \} \vdash t : S \dashv \Gamma_0}{\Gamma \vdash (\lambda x : T.\ t) : T \to S \dashv \Gamma_0}$$

$$\text{T-Quote}\ \frac{C \vdash t : T \dashv \Gamma}{\Gamma \vdash \lceil t \rceil : \texttt{Code}\ T\ C \dashv \Gamma_0} \qquad \text{T-Anti}\ \frac{(x : \texttt{Code}\ T\ \Gamma) \in \Gamma'}{\Gamma \vdash \lfloor x \rfloor : T \dashv \Gamma'} \qquad \text{T-Anti'}\ \frac{(x : \texttt{Int}) \in \Gamma'}{\Gamma \vdash \lfloor \texttt{const}\ x \rfloor : \texttt{Int} \dashv \Gamma'}$$

$$\text{T-Plus}\ \frac{\Gamma \vdash t_0 : \texttt{Int} \dashv \Gamma_0 \qquad \Gamma \vdash t_1 : \texttt{Int} \dashv \Gamma_0}{\Gamma \vdash t_0 + t_1 : \texttt{Int} \dashv \Gamma_0} \qquad \text{T-Mat}\ \frac{\begin{array}{ccc} \Gamma \vdash t_s : \texttt{Code}\ T\ C \dashv \Gamma_0 & \Gamma'' = \Gamma'/C & \Gamma \vdash t_e : T \dashv \Gamma_0 \\ C \vdash t_p : T \dashv^* \Gamma' & & \Gamma \cup \Gamma'' \vdash t_b : T \dashv \Gamma_0 \end{array}}{\Gamma \ \vdash\ t_s\ \texttt{match}\ \lceil t_p \rceil\ \Rightarrow\ t_b\ \texttt{else}\ t_e\ :\ T \dashv \Gamma_0}$$

$$\text{T-Sub}\ \frac{\Gamma \vdash t : T_0 \dashv \Gamma_0 \qquad T_0 <: T_1}{\Gamma \vdash t : T_1 \dashv \Gamma_0} \qquad \text{T-Rwr}\ \frac{\begin{array}{ccc} \Gamma \vdash t_s : \texttt{Code}\ T\ C \dashv \Gamma_0 & \Gamma'' = \Gamma'/(C \cup \{ y \}) & y \notin C \\ C \vdash t_p : T' \dashv^* \Gamma' & \Gamma \cup \Gamma'' \vdash t_b : \texttt{Code}\ T'\ (C \cup \{ y \}) \dashv \Gamma_0 & \end{array}}{\Gamma \ \vdash\ t_s\ \texttt{rewrite}\ \lceil t_p \rceil\ \Rightarrow\ t_b\ :\ \texttt{Code}\ T\ C \dashv \Gamma_0}$$

$$\text{T-Close}\ \frac{\Gamma \vdash t : \texttt{Code}\ T\ (C \cup \{ x : S \}) \dashv \Gamma_0 \qquad \Gamma \vdash t' : \texttt{Code}\ T\ C \dashv \Gamma_0}{\Gamma \vdash \texttt{close}_x\ \ t\ \ \texttt{else}\ \ t' : \texttt{Code}\ T\ C \dashv \Gamma_0} \qquad \text{T-Run}\ \frac{\Gamma \vdash t : \texttt{Code}\ T\ \emptyset \dashv \Gamma_0}{\Gamma \vdash \texttt{run}\ t : T \dashv \Gamma_0}$$

Subtyping Rules

$$\text{T-Var}\ \frac{T_0 <: T_1 \qquad C_0 \subseteq C_1}{\texttt{Code}\ T_0\ C_0\ \ <:\ \ \texttt{Code}\ T_1\ C_1} \qquad \text{T-Fun}\ \frac{T_0 <: T_1 \qquad T_2 <: T_3}{T_1 \to T_2\ \ <:\ \ T_0 \to T_3} \qquad \text{T-Refl}\ \frac{}{T\ <:\ T}$$

Fig. 3. Typing and subtyping rules of $\lambda^{\{\}}$.

outer context serves as a binder for the extracted variables, whereas when typing an expression it is used as a normal context, where weakening is in order.

Rewriting. The rule for rewriting *T-Rwr* is similar to *T-Mat*. The differences are that: *T-Rwr* does not require the type of pattern t_p to coincide with that of scrutinee t_s, because the pattern may match any sub-term of the scrutinee; *T-Rwr* requires body t_b to be a code value with the same type as the pattern, as any matched pattern will be replaced by t_b. finally, *T-Rwr* predicates the local context Γ'' on $C \cup \{ y \}$, where y is some fresh "context brand." The effect is to introduce y into the context parameters of all terms extracted from the pattern, which will prevent them from being run: the only way to eliminate that brand from the context of a term is to use the term as body t_b of the rewriting itself. As a simple example, consider program $\lceil \lambda x : \texttt{Int}.\ x + 1 \rceil$ $\texttt{rewrite}$ $\lceil \lfloor y \rfloor + 1 \rceil \Rightarrow$ $\texttt{let}\ z : \texttt{Int} = \texttt{run}\ y\ \texttt{in}\ \lceil 0 \rceil$, which has to be ill-typed because it tries to run the open term $\lceil x \rceil$ extracted as y. Thankfully, *T-Rwr* types y not as $\texttt{Code Int}\ \emptyset$ but as $\texttt{Code Int}\ \{ y_0 \}$ where y_0 is some fresh brand preventing *T-Run* from applying. Squid uses a similar mechanism (cf., Section 4.3).

3.3 Operational Semantics

Syntax. Values v are either integer literals, lambda abstractions closing over some contexts (closures) or quoted code $\lceil t \rceil$ where t does not contain any immediate unquotes (i.e., unquotes that are not inside a quote), which we write $\lceil q_\emptyset \rceil$ where $\emptyset$ is the empty production. Value substitution

$$\text{E-Var}\ \frac{(x \mapsto v) \in \gamma}{\gamma \vdash x \to v} \qquad \text{E-App}\ \frac{\gamma \vdash t_a \to v_a \quad \gamma \vdash t_f \to \langle \lambda x : T.\ t,\ \gamma_f \rangle \quad \gamma_f \cup \{\, x \mapsto v_a \,\} \vdash t \to v}{\gamma \vdash t_f\ t_a \to v} \qquad \text{E-Lam}\ \frac{}{\gamma \vdash \lambda x : T.\ t \to \langle \lambda x : T.\ t,\ \gamma \rangle}$$

$$\text{E-Plus}\ \frac{\gamma \vdash t_0 \to n_0 \quad \gamma \vdash t_1 \to n_1 \quad n_0 + n_1 = n_3}{\gamma \vdash t_0 + t_1 \to n_3} \qquad \text{E-Match}\ \frac{\gamma \vdash t_s \to \lceil t_s' \rceil \quad t_s' \gg t_p = \gamma_b \quad \gamma \cup \gamma_b \vdash t_b \to v}{\gamma \vdash t_s\ \texttt{match}\ \lceil t_p \rceil \Rightarrow t_b\ \texttt{else}\ t_e \to v} \qquad \text{E-Lit}\ \frac{}{\vdash n \to n}$$

$$\text{E-Run}\ \frac{\gamma \vdash \lceil t \rceil \to \lceil t' \rceil \quad \gamma \vdash t' \to v}{\gamma \vdash \texttt{run}\ \lceil t \rceil \to v} \qquad \text{E-NoMatch}\ \frac{\gamma \vdash t_e \to v \quad \gamma \vdash t_s \to \lceil t_s' \rceil \quad (t_s', t_p) \notin dom(\gg)}{\gamma \vdash t_s\ \texttt{match}\ \lceil t_p \rceil \Rightarrow t_b\ \texttt{else}\ t_e \to v} \qquad \text{E-Asc}\ \frac{\gamma \vdash t \to v}{\gamma \vdash t : T \to v}$$

$$\text{E-Closed}\ \frac{\gamma \vdash t \to \lceil t' \rceil \quad x \notin FV(t')}{\gamma \vdash \texttt{close}_x\ t\ \texttt{else}\ t_e \to \lceil t' \rceil} \qquad \text{E-Open}\ \frac{\gamma \vdash t \to \lceil t' \rceil \quad x \in FV(t') \quad \gamma \vdash t_e \to v}{\gamma \vdash \texttt{close}_x\ t\ \texttt{else}\ t_e \to v}$$

$$\text{Q-Var}\ \frac{}{\gamma \vdash \lceil x \rceil \to \lceil x \rceil} \qquad \text{Q-Lit}\ \frac{}{\vdash \lceil n \rceil \to \lceil n \rceil} \qquad \text{Q-Asc}\ \frac{\gamma \vdash \lceil t \rceil \to \lceil t' \rceil}{\gamma \vdash \lceil t : T \rceil \to \lceil t' : T \rceil}$$

$$\text{Q-App}\ \frac{\gamma \vdash \lceil t_f \rceil \to \lceil t_f' \rceil \quad \gamma \vdash \lceil t_a \rceil \to \lceil t_a' \rceil}{\gamma \vdash \lceil t_f\ t_a \rceil \to \lceil t_f'\ t_a' \rceil} \qquad \text{Q-Lam}\ \frac{\gamma \vdash \lceil t \rceil \to \lceil t' \rceil}{\gamma \vdash \lceil \lambda x : T.\ t \rceil \to \lceil \lambda x : T.\ t' \rceil} \qquad \text{Q-Run}\ \frac{\gamma \vdash \lceil t \rceil \to \lceil t' \rceil}{\gamma \vdash \lceil \texttt{run}\ t \rceil \to \lceil \texttt{run}\ t' \rceil}$$

$$\text{Q-Quote}\ \frac{}{\gamma \vdash \lceil \lceil t \rceil \rceil \to \lceil \lceil t \rceil \rceil} \qquad \text{Q-Anti'}\ \frac{(x \mapsto n) \in \gamma}{\gamma \vdash \lceil \lfloor \texttt{const}\ x \rfloor \rceil \to \lceil n \rceil} \qquad \text{Q-Anti}\ \frac{(x \mapsto \lceil t \rceil) \in \gamma}{\gamma \vdash \lceil \lfloor x \rfloor \rceil \to \lceil t \rceil}$$

$$\text{Q-Mat}\ \frac{\gamma \vdash \lceil t_s \rceil \to \lceil t_s' \rceil \quad \gamma \vdash \lceil t_b \rceil \to \lceil t_b' \rceil \quad \gamma \vdash \lceil t_e \rceil \to \lceil t_e' \rceil}{\gamma \vdash \lceil t_s\ \texttt{match}\ \lceil t_p \rceil \Rightarrow t_b\ \texttt{else}\ t_e \rceil \to \lceil t_s'\ \texttt{match}\ \lceil t_p \rceil \Rightarrow t_b'\ \texttt{else}\ t_e' \rceil}$$

$$\text{Q-Rwr}\ \frac{\gamma \vdash \lceil t \rceil \to \lceil t' \rceil \quad \gamma \vdash \lceil t_b \rceil \to \lceil t_b' \rceil}{\gamma \vdash \lceil t\ \texttt{rewrite}\ \lceil t_p \rceil \Rightarrow t_b \rceil \to \lceil t'\ \texttt{rewrite}\ \lceil t_p \rceil \Rightarrow t_b' \rceil} \qquad \text{Q-Plus}\ \frac{\gamma \vdash \lceil t_0 \rceil \to \lceil t_0' \rceil \quad \gamma \vdash \lceil t_1 \rceil \to \lceil t_1' \rceil}{\gamma \vdash \lceil t_0 + t_1 \rceil \to \lceil t_0' + t_1' \rceil}$$

$$\text{Q-Close}\ \frac{\gamma \vdash \lceil t \rceil \to \lceil t' \rceil \quad \gamma \vdash \lceil t_e \rceil \to \lceil t_e' \rceil}{\gamma \vdash \lceil \texttt{close}_x\ t\ \texttt{else}\ t_e \rceil \to \lceil \texttt{close}_x\ t'\ \texttt{else}\ t_e' \rceil}$$

Fig. 4. Basic rules from the big step operational semantics of $\lambda^{\{\}}$.

contexts γ map identifiers to values,[14] and $\models$ is used to express that a value substitution context *conforms to* or is *consistent with* a typing context:

$$v ::= n \mid \langle \lambda x : T.\ t,\ \gamma \rangle \mid \lceil q_\emptyset \rceil$$
$$\gamma ::= \{\, \overline{x \mapsto v} \,\}$$

$$\frac{}{\emptyset \models \emptyset} \qquad \frac{\Gamma \models \gamma \quad \vdash v : T}{\Gamma \cup \{\, x : T \,\} \models \gamma \cup \{\, x : v \,\}}$$

$$\text{T-Clos}\ \frac{\Gamma \models \gamma \quad \Gamma \cup \{\, x : T \,\} \vdash t : S}{\Gamma \vdash \langle \lambda x : T.\ t,\ \gamma \rangle : T \to S}$$

E- and Q-Rules. Figure 4 shows the basic big step semantics rules of $\lambda^{\{\}}$. These rules are of the form $\gamma \vdash t \to v$, read "under context γ, t evaluates to v." *E-* rules are for current stage code. *Q-* rules, which are for next stage code (terms surrounded by one level of quotation), replace immediate unquotes in quoted code by the values to be unquoted; they can be seen as the β-rule(s) for quotes. For example $\{\, x \mapsto \lceil y + 1 \rceil \,\} \vdash \lceil \lambda y : \texttt{Int}.\ \lfloor x \rfloor \rceil \to \lceil \lambda y : \texttt{Int}.\ y + 1 \rceil$. *E-Run* takes code from the next stage and evaluates it as code in the current stage. Rules *E-Match* and *E-NoMatch* make use of partial function $\gg$ to match a term against a given pattern, producing a value substitution

[14] We do not use direct substitution because our syntax prevents expressions from appearing in unquotes, which means we cannot perform straightforward substitution of expressions for variables. Additionally, contexts interact more intuitively with the semantics of pattern matching, which introduces a set of bindings to be merged with the current context.

$$
\begin{array}{rcll}
(t : T) \gg (t' : S) & = & t \gg^T t' \quad \text{if } T <: S & \text{(X-Asc)} \\
x \gg^T x & = & \emptyset & \text{(X-Var)} \\
t \gg^T \lfloor x \rfloor & = & \{\, x \mapsto \lceil t : T \rceil \,\} & \text{(X-Anti)} \\
n \gg^T n & = & \emptyset & \text{(X-Lit)} \\
n \gg^T \lfloor \texttt{const}\ x \rfloor & = & \{\, x \mapsto n \,\} & \text{(X-Anti')} \\
\lceil t \rceil \gg^T \lceil t \rceil & = & \emptyset & \text{(X-Quote)} \\
(\lambda x : S.\ t) \gg^T (\lambda y : S'.\ t') & = & t[x \mapsto y] \gg t' & \text{(X-Lam)} \\
\texttt{run}\ t \gg^T \texttt{run}\ t' & = & t \gg t' & \text{(X-Run)} \\
(t_0 + t_1) \gg^T (t'_0 + t'_1) & = & (t_0 \gg t'_0) \uplus (t_1 \gg t'_1) & \text{(X-Plus)} \\
(t_0\ t_1) \gg^T (t'_0\ t'_1) & = & (t_0 \gg t'_0) \uplus (t_1 \gg t'_1) & \text{(X-App)} \\
\texttt{close}_x\ t_0\ \texttt{else}\ t_1 \gg^T \texttt{close}_x\ t'_0\ \texttt{else}\ t'_1 & = & (t_0 \gg t'_0) \uplus (t_1 \gg t'_1) & \text{(X-Close)} \\
t_s\ \texttt{match}\ \lceil t_p \rceil \Rightarrow t_b\ \texttt{else}\ t_e \gg^T t'_s\ \texttt{match}\ \lceil t_p \rceil \Rightarrow t'_b\ \texttt{else}\ t'_e & = & (t_s \gg t'_s) \uplus (t_b \gg t'_b) \uplus (t_e \gg t'_e) & \text{(X-Mat)} \\
t_s\ \texttt{rewrite}\ \lceil t_p \rceil \Rightarrow t_b \gg^T t'_s\ \texttt{rewrite}\ \lceil t_p \rceil \Rightarrow t'_b & = & (t_s \gg t'_s) \uplus (t_b \gg t'_b) & \text{(X-Rwr)}
\end{array}
$$

Fig. 5. Extraction rules for pattern matching in $\lambda^{\{\}}$.

context that contains the results of the matching. For example, $\lceil (x : \texttt{Int} \to \texttt{Int})\ (123 : \texttt{Int}) \rceil \gg^{\texttt{Int}} \lceil (\lfloor y \rfloor : \texttt{Int} \to \texttt{Int})\ (\lfloor \texttt{const}\ z \rfloor : \texttt{Int}) \rceil = \{\, y \mapsto \lceil x : \texttt{Int} \rceil;\ z \mapsto 123 \,\}$. The definitions of $\gg$ and $\gg^T$ are explained later in this section. Rule *E-Closed* searches for free occurrences of x in its argument term (where *FV* is defined as usual for multi-stage languages, for example see [Rhiger 2012b]); it evaluates to the term unchanged if there are none, and otherwise evaluates to the **else** branch.

For lack of space, we do not list all the rules for **rewrite**, as they just go through a term and transform it by reusing the semantics of pattern matching. Below we only give two examples:

$$
\text{E-R-Lit}\ \frac{t \to \lceil n \rceil \qquad \gamma \vdash \lceil n \rceil\ \texttt{match}\ \lceil t_p \rceil \Rightarrow t_b\ \texttt{else}\ \lceil n \rceil \to v}{\gamma \vdash t\ \texttt{rewrite}\ \lceil t_p \rceil \Rightarrow t_b \to v}
\qquad
\text{E-R-Plus}\ \frac{\begin{array}{c}\gamma \vdash t \to \lceil t_0 + t_1 \rceil \\ \gamma \vdash \lceil t_0 \rceil\ \texttt{rewrite}\ \lceil t_p \rceil \Rightarrow t_b \to \lceil t'_0 \rceil \\ \gamma \vdash \lceil t_1 \rceil\ \texttt{rewrite}\ \lceil t_p \rceil \Rightarrow t_b \to \lceil t'_1 \rceil \\ \gamma \vdash \lceil t'_0 + t'_1 \rceil\ \texttt{match}\ \lceil t_p \rceil \Rightarrow t_b\ \texttt{else}\ \lceil t'_0 + t'_1 \rceil \to v\end{array}}{\gamma \vdash t\ \texttt{rewrite}\ \lceil t_p \rceil \Rightarrow t_b \to v}
$$

Rule *R-Lit* simply applies pattern matching on a constant literal n (as this term has no sub-expressions); if the pattern does not match, the rule returns the term unchanged. Rule *R-Plus* first applies **rewrite** recursively inside both sides of an addition, and then applies pattern matching to transform the top-level expression made of the results of these two recursive calls.

Intensional Type Analysis. Similar to Squid (cf., Section 4.6), $\lambda^{\{\}}$ performs run-time subtyping checks to guide pattern matching. For example, pattern $\lceil (\lfloor x \rfloor : \texttt{Int} \to \texttt{Int})\ \lfloor y \rfloor \rceil$, where x and y are typed respectively as $\texttt{Code}\ (\texttt{Int} \to \texttt{Int})\ C$ and $\texttt{Code}\ \texttt{Int}\ C$, should not match a program fragment such as $\lceil (\lambda x_0 : \texttt{Int} \to \texttt{Int}.\ x_0)\ (\lambda x_1 : \texttt{Int}.\ x_1) \rceil$, because the extracted terms would not have the expected types. In order to enable those runtime checks, we actually perform evaluation not directly on a source program t, but on its translation $[\![t]\!]^{\Gamma'}_{\Gamma}$ into an explicitly-typed variant of $\lambda^{\{\}}$ — a form where every subterm is annotated with its type as assigned by the typing rules, given inner context Γ and outer context Γ'. For example, $[\![x + 1]\!]^{\emptyset}_{\{\, x : \texttt{Int} \,\}} = (x : \texttt{Int}) + (1 : \texttt{Int}) : \texttt{Int}$.

Extraction. Figure 5 shows the definitions of partial function $\gg$ and its helper $\gg^T$. We write $\gamma_0 \uplus \gamma_1$ for the disjoint union of value substitution contexts γ_0 and γ_1, which is only defined if their domains are disjoint, that is to say, $dom(\gamma_0) \cap dom(\gamma_1) = \emptyset \implies \gamma_0 \uplus \gamma_1 \stackrel{\text{def}}{=} \gamma_0 \cup \gamma_1$.

Case *X-Var* matches two variables with the same name and compatible types, producing an empty result (as nothing is extracted from this match). For example $\lceil x : \texttt{Code Int}\ \emptyset \rceil$ matches pattern $\lceil x : \texttt{Code Int}\ \{\, y : \texttt{Int} \,\} \rceil$ because we have $\texttt{Code Int}\ \emptyset <: \texttt{Code Int}\ \{\, y : \texttt{Int} \,\}$. On the other hand, case *X-Anti* matches anything with a compatible type and *extracts* it as a code value, which corresponds to the semantics of unquotes in pattern position.

Particularly interesting is the case *X-Lam*, which matches two lambda bodies by renaming variable x bound in the scrutinee to the name of the variable bound in the pattern. This way, any code extracted by $t[x \mapsto y] \gg t'$ will refer to bound variable x of the original term as y, the name used in the pattern. Remember that we assume sufficient α-renaming to avoid name collisions, which includes the assumption that y is not already present in t. To enforce hygiene in practice, Squid uses an elaborate scheme similar to the *locally named* representation [McBride and McKinna 2004; McKinna and Pollack 1993]. This schemes consists in using different syntactic constructs to distinguish free variables from bound variables, so that they can never be confused. Interestingly, this also gives us a way to compare open terms for equivalence – in Squid term equality is implemented as reciprocal matching $t_0 \approx t_1 \stackrel{\text{def}}{=} (t_0 \gg t_1) = (t_1 \gg t_0) = \emptyset$. Closed term equivalence could be implemented more efficiently using a *locally nameless* representation [Charguéraud 2012], but as it is, that representation has some drawbacks in the context of user-defined DSL compilers — for example, it forgets the original names of variables (which are helpful when debugging).

3.4 Soundness of $\lambda^{\{\}}$

Top-Level Evaluation. We write $t \Downarrow v$ the annotation and evaluation of program t down to value v, an abbreviation of $\emptyset \vdash [\![t]\!]^{\emptyset}_{\emptyset} \to v$. Note that in the proofs below, we refer to terms t with no assumptions on whether they are in an annotated form or not, because that is not a requirement for the soundness of $\lambda^{\{\}}$. Failing to annotate a program before evaluating it will *not* result in evaluation getting stuck, however it *may* result in a different evaluation result, as partial function $\gg$ is not defined on terms lacking explicit type annotations.

Canonical Forms. Since the type system admits a subtyping rule (*T-Sub*) and a reflexive subtyping relation, inverting the typing judgment always yields multiple possibilities, including the use of *T-Sub*. This leads to some bureaucracy in the proofs, forcing us to take care of the subtyping case in addition to the main case.

To help with that issue, we introduce an inversion lemma for the subtyping relation:

Lemma 3.3 (Subtyping inversion). *If $S <: T$, then S is a* pointwise-subtype *of T, define as:*

- *S has the same type constructor as T*
- *Then, depending on T:*
  - *if $T = \texttt{Int}$, then $S = \texttt{Int}$*
  - *if $T = T_1 \to T_2$, then $S = S_1 \to S_2$ with $T_1 <: S_1$ and $S_2 <: T_2$*
  - *if $T = \texttt{Code}\ T'\ C_T$, then $S = \texttt{Code}\ S'\ C_S$ with $S' <: T'$ and $C_S \subseteq C_T$*

Proof. By induction on derivations of $S <: T$. □

Notice that, because S and T are so tightly coupled, the "upward" version of lemma 3.3 is also admissible (where we do a case analysis on S to infer T's shape).

Thanks to this lemma, we know that *T-Sub* preserves the type constructor and can only replace its arguments by subtypes of theirs (or supertypes in contravariant positions). In the following, when it is clear that a property is preserved by subtyping thanks to this lemma, we may use the phrase “modulo subtyping” as a shorthand.

Remark 1 (Inversion Modulo Subtyping). *Lemma 3.3 has an important consequence. First, notice that for any term shape t, only one typing rule $\mathcal{R}$ applies aside from T-Sub — essentially, the system is syntax-directed modulo subtpying. Thus, we know precisely the structure of any type derivation for terms of that shape: it ends with $\mathcal{R}$ followed by an arbitrary number of instances of T-Sub.*

Now, applying the lemma to that observation means that all instances of T-Sub in that derivation yield pointwise-subtypes, which is a reflexive and transitive relation. Therefore, inverting the typing assumption yields the use of $\mathcal{R}$, just slightly weakened — the type of t is replaced by an arbitrary pointwise-subtype (both in the premises and the conclusion).

Lemma 3.4 (Preservation for annotation). *If $\Gamma \vdash t : T \dashv \Gamma'$, then $\Gamma \vdash [\![t]\!]_{\Gamma}^{\Gamma'} : T \dashv \Gamma'$.*

Proof. By induction on derivations of $\Gamma \vdash t : T \dashv \Gamma'$ and definition of $[\![t]\!]_{\Gamma}^{\Gamma'}$. □

Lemma 3.5 (Canonical Forms). *If $t = v$ and $\vdash t : T$, then*

- *if $T = \textbf{Int}$, then $t = n$ for some n*
- *if $T = T_1 \to T_2$, then $t = \langle \lambda x : T_1.\ t,\ \gamma \rangle$ for some x and t_1*
- *if $T = \textbf{Code}\ T'\ C$, then $t = \lceil t' \rceil$, for some t' such that $C \vdash t' : T' \dashv$*

Proof. By induction on the typing rules and the syntax of values, modulo subtyping. □

We will also need the following lemma about the type of a quote.

Lemma 3.6 (Inversion for quotes). *If $\Gamma \vdash \lceil t \rceil : T$, then there exists T_0, T_1, C_0, C_1 such that $T = \textbf{Code}\ T_1\ C_1$ and $C_0 \vdash t : T_0 \dashv \Gamma$, with $T_0 <: T_1$ and $C_0 \subseteq C_1$.*

Proof. By induction on the typing derivation. There are only 2 cases that apply: *T-Sub*, which is immediate by 3.3 and by transitivity of both $<:$ and $\subseteq$; and the base case, *T-Quote*, which allows to conclude thanks to the reflexivity of those relations. □

For the proof of preservation, we first need the following lemma.

Lemma 3.7 (Evaluation to quotes yields values). *For any value substitution context γ and term t, if $\gamma \vdash t \to \lceil t' \rceil$, then $\lceil t' \rceil \in v$. I.e., a term never evaluates to a quote containing immediate unquotes.*

Proof. By induction on the reduction. *E-Asc* and *Q-Asc* are immediate by induction on their unique premise, *E-Closed* on its first. All the other *E-*∗ rules are immediate, since none produces an unspecified quoted term; thus, all the ones which can apply (producing a value v) are obviously correct. The only interesting *Q-*∗ rule is *Q-Anti*, which is solved by the observation that γ maps identifier to values, applied to the first premise. *Q-Anti'* is trivial since the resulting quote is a value. All the other *Q-*∗ rules are solved directly because they are essentially congruence rules - under the assumption that no sub-term contains an immediate unquote after reduction, then the term itself can't contain one either (after reduction, again). □

Lemma 3.8 (Preservation — general). *Evaluation in conforming contexts preserves typing: If $\gamma \vdash t \to v$ then for all Γ, Γ', T such that $\Gamma \models \gamma$ and $\Gamma \vdash t : T \dashv \Gamma'$, one has $\Gamma \vdash v : T$.*

Proof. By induction on the evaluation derivation. (We replace t by the notations used in the typing & evaluation rules)

Case *E-Var* Since x is typable in Γ, then $\Gamma(x) = T$. We conclude by conformance of γ to Γ.

Case *E-Lit, Q-Lit, Q-Var, Q-Quote* These cases are immediate since v evaluates to itself.

Case *E-Ignore* By remark 1, inverting the judgment $\Gamma \vdash (t : T) : T \dashv \Gamma'$ yields $\Gamma \vdash t : S \dashv \Gamma'$, with $S <: T$. By induction hypothesis on that judgment, v has type S as well — and T by *T-Sub*.

Case *E-Lam* By inversion modulo subtyping, we get that $T = T_1 \to T_2$ and that there exists S_1 and S_2 such that $T_1 <: S_1$, $S_2 <: T_2$ and $\Gamma \vdash \lambda x : S_1.\ t : S_1 \to S_2$. The resulting closure v is typed via *T-Clos* and *T-Sub*; the second premise of *T-Clos* is exactly the same as the one of *T-Lam*, and the first premise is provided by the assumption that $\Gamma \models \gamma$.

Case *E-App* By remark 1, we inverse the typing judgment and obtain the judgments $\Gamma \vdash t_f : T' \to S'$ and $\Gamma \vdash t_a : T''$, with $S' <: S$ and $T'' <: T <: T'$. By induction on the first 2 premises, the closure has the same type than t_f, namely $T' \to S'$, and v_a has the same type than t_a, T'. Notice that $\Gamma \cup \{ x : T \} \models \gamma \cup \{ x \mapsto v_a \}$. The context exactly coincides with the one of rule *T-Clos* for the closure (by inversion and remark 1 again). The conclusion follows by induction on the last premise.

Case *E-Plus* By inversion of the typing judgment, we get that both t_0 and t_1 have type **Int**. By induction and the canonical forms lemma, they reduce to two constants n_0 and n_1.

Case *E-Run* By the inversion lemma for quotes, the type of t is **Code** T' C' for some T' and C'. By induction on the first premise, $\lceil t' \rceil$ (which is a value) has the same type as $\lceil t \rceil$, namely **Code** T' C'. This in turn gives us that t' has type S, with $S <: T'$. Hence, by induction on the second premise, v has type S; thus also type T' by *T-Sub* — but T' is also the type of **run** t.

Case *E-Match* By inversion modulo subtyping, the base case must be *T-Mat* with return type S, where $S <: T$. This gives us the right typing judgment on t_b to use with the corresponding induction hypothesis (on the evaluation of t_b), provided we show that $\Gamma_b \models \gamma_b$. This is obtained from the typing judgment on t_p and the definition of Γ'' in *T-Mat*, and the definition of γ_b in *E-Match* (i.e., from the matching rule). Thus, v has type S, and also T by *T-Sub*.

Case *E-NoMatch* This case is similar to the last one, even slightly simpler (no context extension)

Cases *E-Rw-∗* All these rules are handled in a similar fashion. First, by inversion modulo subtyping, we get that the base case must be *T-Rwr*. We also apply the induction hypothesis to the first premise, the one asserting the reduction of the scrutinee - invoking lemma 3.7 if necessary. This allows, after inverting (modulo subtyping) the typing hypothesis we just derived, to apply induction on any sub-rewrite premise. This ensures that the sub-terms of the scrutinee (in the last premise) have the correct type. One concludes by induction on the last premise.

Case *E-Closed* By inversion modulo subtyping, one gets $\Gamma \vdash t$: **Code** T' $C' \dashv \Gamma'$, with $T' <: T$ and $C' \subseteq C \cup \{x : S\}$. By induction on the first premise, one get that $\lceil t' \rceil$ has the same type. If $x : S \in C'$, it is easy to see that $\lceil t' \rceil$ can also be given the subtype **Code** T' $(C' \setminus \{x : S\})$, by the second premise of *E-Closed*. We conclude by applying *T-Sub* if necessary.

Case *E-Open* By inversion modulo subtyping, one gets $\Gamma \vdash t$: **Code** T' $C' \dashv \Gamma'$, with $T' <: T$ and $C' \subseteq C$. We conclude by induction hypothesis on the last premise, using *T-Sub* if necessary.

Case *Q-Anti* Again by remark 1, inverting the typing judgment on $\lceil \lfloor x \rfloor \rceil$ yields that it has type **Code** T' C', and that $C' \vdash \lfloor x \rfloor : T' \dashv \Gamma$ with $T' <: T$. Inverting that premise again gives us x : **Code** T'' $C' \in \Gamma$ with $T'' <: T'$. Since $x \mapsto \lceil t \rceil \in \gamma$ (premise of *Q-Anti*) and $\Gamma \models \gamma$, we get that $\lceil t \rceil$ is a value of type **Code** T'' C' in Γ. We conclude by applying *T-Sub* if necessary.

Case *Q-Anti'* By inversion modulo subtyping and since **Int** has itself as only subtype/supertype, we have $\Gamma \vdash \lceil \lfloor \mathbf{const}\ x \rfloor \rceil$: **Code Int** C, as well as $C \vdash \lfloor \mathbf{const}\ x \rfloor$: **Int** $\dashv \Gamma$ (*T-Quote*), and x : **Int** $\in \Gamma$ (*T-Anti'*). We conclude by recalling that $\Gamma \models \gamma$.

All the remaining *Q-∗* cases, which all apply to quoted terms, are handled the same way.

Each of these cases has premises of the form $\lceil t \rceil \to \lceil t' \rceil$. Thanks to 3.7, we show that such $\lceil t' \rceil$ terms are always values. Thus, we get an induction hypothesis for all such premises (since, by inversion modulo subtyping, the base case for all premises is always *T-Quote*), and we conclude by mirroring the input type derivation for the reduced term, applying *T-Sub* whenever necessary. □

We get preservation as an immediate corollary:

Theorem 3.9 (Type Preservation). *If* $\vdash t : T$ *and* $t \Downarrow v$*, then* $\vdash v : T$*.*

Proof. By Lemmas 3.4 and 3.8. Notice that by definition $\emptyset \models \emptyset$. □

In big step semantics, to distinguish between terms diverging and terms getting stuck, it is customary to extend the syntax with an *error* value **err** (syntax $v_e ::= v \mid \mathbf{err}$) and add rules to, on the one hand, generate errors when no original rule applies, and on the other to propagate errors. Then, *progress* is the property that if a well-typed program evaluates to a value, that value is not an error. The error-related rules for $\lambda^{\{\}}$ are standard, unsurprising, and omitted for lack of space.

For the proof of progress, we first need a version of it that only applies to quoted terms, and assert that they all reduce to quoted terms.

Lemma 3.10 (Quote Progress). *For any contexts* Γ *and* Γ'*, value substitution context* γ *such that* $\Gamma \models \gamma$*, and every term* t *such that* $\Gamma' \vdash t : T \dashv \Gamma$*, there exists* t' *such that* $\gamma \vdash \lceil t \rceil \to \lceil t' \rceil$*.*

Proof. By induction on $\Gamma \vdash t : T$. *T-Anti* and *T-Anti'* both work thanks to their corresponding *Q-_* rules and the assumption that $\Gamma \models \gamma$. The base cases *T-Var*, *T-Lit* and *T-Quote* are trivial by the associated *Q-_* rules. All the other cases are equally easy, since they don't affect the outer context, and corresponding *Q-_* rules act as congruences. Notice that this remark also apply to *T-Mat* and *T-Rwr* — the only premise where they modify the outer contexts are for the branches, but these are also left untouched by the associated *Q-_* rules. □

Finally, we will also rely implicitly on the fact that the evaluation relation is deterministic.

Lemma 3.11 (Progress — general). *Assume* $\Gamma \models \gamma$*. For any fully annotated term* t*, if* $\Gamma \vdash t : T$ *and* $\gamma \vdash t \to v_e$*, then* $v_e \neq \mathbf{err}$*.*

Proof. By induction on the typing derivation. To be precise, we use a strong induction on the size of the derivation. Most cases can be solved simply by a structural induction, and we handle them in this style; but one case (*T-Run*) requires a slightly more general induction principle.

Case *T-Var* Immediate by conformance of γ to Γ.

Case *T-Lit* Immediate since t is a non-quote value (and these never step).

Case *T-Run* From the unique premise, $\Gamma \vdash t : \mathbf{Code}\ T\ \emptyset$. By inversion modulo subtyping, t has shape $\lceil s \rceil$, for some s such that $\emptyset \vdash s : T \dashv \Gamma$. By lemma 3.10, we get s' such that $\gamma \vdash \lceil s \rceil \to \lceil s' \rceil$. It is an easy result that preservation applied to a reduction of this shape yields a derivation for $\Gamma \vdash \lceil s' \rceil : \mathbf{Code}\ T\ \emptyset$ of size *lesser or equal* to this of $\Gamma \vdash \lceil s \rceil : \mathbf{Code}\ T\ \emptyset$ (even further: one can always type s' with a sub-derivation of this of s). Hence, by induction, we get that for all $v_e^{s'}$ such that $\gamma \vdash s \to v_e^{s'}$, $v_e^{s'}$ is not an error; and $v_e = v_e^{s'}$.

Case *T-Lam* t evaluates to the corresponding closure, by rule *E-Lam* (and closures are values).

Case *T-App* By induction on the premises and preservation, we get 2 values of the corresponding types. By canonical forms on the value obtained for t_f, it is a closure with the appropriate argument type (or a supertype thereof). By induction on its typing derivation, the body reduces without error, with the argument added to the context. We conclude via *E-App*.

Case *T-Asc* Immediate from the induction hypothesis on the unique premise.

Case *T-Quote* By combining lemmas 3.7 and 3.10.

Cases *T-Anti* and *T-Anti'* Impossible since the right context is empty.

Case *T-Plus* By the induction, inversion modulo subtyping, canonical forms and *E-Plus*.

Case *T-Mat* By induction on the first premise, if the scrutinee evaluates to a value, it is not an error. By preservation and canonical forms, this value is a quoted term; we also assumed that t was fully annotated; thus the dynamic matching check doesn't return an error. Depending on the result, we keep evaluating either t_b or t_e. Importantly, the branch t_p can *not* forget the

dependency in any variable being matched, since T-Anti doesn't allow subtyping the type of the unquoted term (and since contexts can not be weakened). This is a critical requirement, or else we could "forget" variable requirements about surrounding variables, making $\gg$ unsafe and thus letting us try to run open code. By induction on the 2 corresponding premises (*T-Mat*), both cases evaluate safely.

Case *T-Rwr* Like in *T-Mat*, the scrutinee evaluates safely (if ever). The reduction is split across multiple rules, but the reasoning is essentially the same as in the pattern matching case. One may have to perform more inversion on the typing hypothesis (to match the structure of the rewriting being performed — this is proved by induction on the rewriting rules), and conclude from the associated induction hypotheses.

Case *T-Sub* Follows immediately from the induction hypothesis.

Case *T-Close* By induction (first premise), preservation and canonical forms, if $t \to v$ then v has shape $\lceil t' \rceil$. Then, check whether $x \in FV(t')$. If not, an easy result shows that $\lceil t' \rceil$ also has type $\texttt{Code}\ T\ C$ (removing the uses of *T-Sub* that add x). We conclude by *E-Closed*. If, on the other hand, x is in $FV(t')$, we conclude by induction (second premise of *T-Close*) and *E-Open*.

□

Theorem 3.12 (Progress). *If* $\vdash t : T$ *and* $t \Downarrow v_e$ *, then* $v_e \neq \texttt{err}$.

Proof. By Lemma 3.11. Notice that by definition $\emptyset \models \emptyset$. □

3.5 Extensions

In this section, we present possible extensions to $\lambda^{\{\}}$ that are left as future work.

Imperative Effects. To better mirror the capabilities of Squid, we could add imperative features to $\lambda^{\{\}}$ such as mutable references. We expect this change to be straightforward and unproblematic; effects caused problems in work such as the original MetaML because the meaning of identifiers in program fragments was derived from the lexical scoping of the quotes — i.e., code values could not safely leave their scopes at runtime — and mutable references as well as exceptions could be used to violate this lexical scoping (pulling values out into the heap). However, various works have since shown [Kameyama et al. 2014; Kiselyov et al. 2016; Rhiger 2012b] that properly reflecting scope dependencies inside the types of program fragments was sufficient to solve the problem.

Type-Parametric Matching. As seen in Section 2.2, Squid has the ability to define patterns which extract *type*, in addition to terms. Extending $\lambda^{\{\}}$ with this functionality would require the extension of the type language to allow $\lfloor T \rfloor$ and $\lceil T \rceil$, and in order to prevent mixing up distinct extracted types we would need a mechanism to prevent extracted types from "escaping" the pattern matching branch in which they are available. Below is an example use of this feature:

$$\lceil (\lambda x : \texttt{Int}.\ x + 1)\ 42 \rceil\ \texttt{match}\ \lceil (\lfloor x \rfloor : \lfloor T \rfloor \to \texttt{Int})\ \lfloor y \rfloor \rceil \Rightarrow \lceil \lfloor x \rfloor\ \lfloor y \rfloor + \lfloor x \rfloor\ \lfloor y \rfloor \rceil\ \texttt{else}\ \lceil 0 \rceil$$

Context Polymorphism. Squid's support for *context polymorphism* (also called *support polymorphism* [Nanevski 2002]) is presented in Section 4.3, and in Section 4.4 we show a technique to enforce the hygiene of that feature based on implicit evidence propagation. We believe that it would be straightforward to include an ad hoc version of this evidence propagation scheme in the type system of $\lambda^{\{\}}$, along with explicit constructs for introducing and eliminating context parameters.

4 EMBEDDING IN SCALA

This section is aimed at giving the reader a better understanding of the mechanisms underlying Squid, as well as giving prospective implementers of advanced type system techniques insights

on how Scala facilitates such endeavors. The main takeaway is that the combination of a flexible type system with an advanced type-aware macro system can go a long way towards implementing advanced statically-typed features without modifying the host language's compiler.

4.1 Compilation of Squid Quasiquotes

Squid quasiquotes are implemented as macros that perform parsing and type-checking of quoted fragments, compute and check associated types and contexts, and produce the Scala code necessary to reconstruct the program fragments at runtime encoded in Squid's intermediate representation.

Basic Expansion. To understand how Squid quasiquotes are compiled, let us start with a simple example, `code"Math.pow($x,2)"`, where some value `x` is in scope with type `Code[Int,{}]`. In Scala, this expression is conceptually equivalent to a simple invocation of the form `code(List("Math.pow(", ",2)"), x)`. `code` being a macro, it executes *at compile-time*. The first thing it does is to interpret the strings passed in its first argument as a Scala code snippet. To do this, it reconstitutes the fragment as `"Math.pow(hole[Int,{}](0),2)"` and parses it using the Scala parser. `hole[Int,{}](0)` represents the unquoted value `x`, where type arguments `Int` and `{}` were retrieved from its type in the current scope, and `0` is a unique identifier associated to the unquote. This snippet of code is then type checked using Scala's type checker, given signature `def hole[T,C]: T`. In this case, we end up with `"java.lang.Math.pow(hole[Int,{}](0).toDouble,2.0)"`, typed `Double`. Notice the insertion of `toDouble` as a result of type checking: similarly, the Scala type checker adds missing type parameters, inferred implicit arguments, fully-qualified names, etc. An analysis phase mirroring the rules presented in Section 3.2 then goes through the typed AST and infers which free variables of the inserted terms are captured, as well as the context of the resulting quoted term, here `{}`. The next step is to *lift* this typed AST into a program that reconstructs it at runtime. During this process, `hole[Int,{}](0)` is replaced with `x.impl`, where method `impl` accesses the underlying implementation of a code fragment. We give below a simplified version of the code that is produced:

```
val Math = staticObject("java.lang.Math")
val Math_pow = methodSymbol("java.lang.Math.pow")
val res = methodApp(Math, Math_pow, x.impl, Constant(2.0))
new Code[Double,{}](impl = res)
```

Where `class Code[+T,-C](impl: Impl)` is a typed wrapper that hides its internal untyped code representation `impl`. The expansion of quasiquotes in pattern position is very similar, desugaring to a call to the `extract` method that takes a pattern AST, a scrutinee AST and produces either nothing if the matching failed, or a mapping from extracted term names to extracted code fragments and extracted type names to extracted type representations.

Type-Parametric Matching. Type-parametric matching (Section 2.2) uses the ability of Scala to reason about path-dependent types, which are types that may live in arbitrary objects, including local ones (this is a similar concept to first-class modules in ML). Squid assigns to an extracted type representation `t` the module type `CodeType` that contains an abstract type member `Typ` (a type declaration without a definition). Then, Squid type checks the pattern using references to `t.Typ`. In essence, `t.Typ` — which can only be referred to within the scope of the pattern matching branch where `t` is extracted — is existentially quantified, which is the correct interpretation of type-parametric matchings. For example, assuming `pgrm` has type `Code[Any,{}]`, in the code below:

```
pgrm match { case code"List[$t]($a,$b)" => print(t) /* t is a term here */ ; code"List($b,$a)" }
```

the pattern is type checked as having type `List[t.Typ]`, and therefore the right-hand side of the match sees a scope with extracted variables `{t: CodeType; a: Code[t.Typ,{}]; b: Code[t.Typ,{}]}`. The return type of this example is `Code[List[_],{}]` — the wildcard in `List[_]` stands for an unknown type (an existential without a path). This is because the local type representation module `t` is invisible from outside the scope of pattern matching branch, which ensures that extracted types from different patterns or even from different runs of a match cannot be mixed with one another.

4.2 Required Properties of the Macro System

In order to achieve its goals of static safety, the Squid quasiquote system relies essentially on two features of the host language's macro system:

- The ability to query type information and invoke the type checker during macro expansion: it should be possible to query the type of unquoted expressions in order to properly type check the quote. Additionally but not essentially, Squid accesses the type of the scrutinee in pattern matching (cf., Section 2.1) and type-checks a quoted program fragment in the same scope as the quote itself, which is why we can write `{import Math.pow; code"pow($x,2)"}`.
- The ability to refine the type of expanded macros: since both the type and the context requirements of program fragments are computed during macro expansion, it must be possible for the compiler to expand a macro call before knowing its final type, and use the precise type of the expansion to type check the rest of the program.

4.3 Context Requirements

Contexts as Contravariant Structural Types. The type of quoted terms `Code[T,C]` is defined as `type Code[+Typ, -Ctx]`, which exhibits that it is covariant in its `Typ` parameter, and contravariant in `Ctx`. Scala applies the traditional rules for structural subtyping so that, for example, for all types X and Y where $X :> Y$, we have $\{\} :> \{a : X\} :> \{a : Y\}$. Therefore, by Scala's subtyping rules, for all T we also have $\text{Code}[T, \{\}] <: \text{Code}[T, \{a : X\}] <: \text{Code}[T, \{a : Y\}]$.

In other words, a term that requires some context C can be used in place of a term that requires some more specific context $D <: C$. In particular, a closed term, which requires no context (written $\{\}$), can be used in place of a term that requires any context. This subsumption principle is also sometimes called *weakening* [Rhiger 2012b] or *type widening*, and is important for the flexibility of the quasiquotes API. It is directly reflected in $\lambda^{\{\}}$ by typing rule *T-Var* presented in Section 3.2.

Context Polymorphism. A consequence of this encoding of contexts as Scala types means that we can abstract over contexts the same way we abstract over other types. In addition, Scala has a concept of intersection types, where $A \,\&\, B$ represents the intersection of types A and B. This means we can express *refinements* on abstract contexts by intersecting an abstract context parameter C and a structural type such as $\{x : T\}$, as in $C \,\&\, \{x : T\}$, also simply written $C\{x : T\}$. This allows Squid users to express useful context-parametric functions, such as `intro` and `outro` below:[15]

```
def intro[C](n: Code[Int, C])                 = code"(?s: String) take $n"
def outro[C](m: Code[String, C{ s: String }]) = code" (s: String) =>   $m"
```

The return type inferred for `intro` is `Code[String,C{s: String}]`, because the context requirement `C` introduced by `n` is propagated to the main term, but that context is extended with the new requirement for a free variable `s` of type `String`, introduced by the `(?s : String)` syntax. The return type inferred for `outro` is `Code[String => String,C]`, because the `s` variable in the context requirement

[15] Note that in Scala, `str.take(n)` represents the `n` first characters of a string `str`, or `str` if `str.length < n`. This method is added via an implicit conversion, but our quasiquotes allow us to ignore it completely.

of `m` is captured by the lambda abstraction `(s: String) => ...`. The following interactive session demonstrates the usage of these definitions:

```
val a = code"?x : Int"    : Code[Int, {x: Int}]              →  code"?x"
val b = intro(a)          : Code[Int, {s: String; x: Int}]   →  code"?s take ?x"
val c = outro[{x:Int}](b) : Code[String=>String,{x:Int}]     →  code"(s:String)=> s take ?x"
code"val x = 12; $c"      : Code[String=>String,{}]          →  code"x = 12; (s:String) => s take x"
```

Term Rewriting. Squid's **`rewrite`** macro, which allows the recursive transformation of all subterms of a program, has to make sure that intermediate extracted subterms are only used in the context from where they were extracted, otherwise this would result in unsafe scope extrusion. This is achieved by making the context matched by each case of the rewrite rule a refinement on some abstract context type that is only usable within the pattern matching branch. For example, in the program below the type of extracted variables `n` and `m` is `Code[Int,<context @ 2:15>{y:Int}]`. Type `<context @ 2:15>`, where `2:15` refers to the line and column of the rewrite rule **`case`**, is a local context synthesized by Squid (akin to `t.Typ` in Section 4.1) that is only valid within the pattern matching branch. This context is refined with `{y:Int}`, the context requirement of the rewritten term `pgrm`.

```
val pgrm = code"val x: Int = ?y ; println(x + ?y)"
pgrm rewrite { case code"($n:Int) + ($m:Int)"  =>  code"(-$m - $n) * (?z : Int)" }
```

The return type of the expression above is `Code[Int,{y:Int;z:Int}]` because the rewrite rule added free variable `z` to the context of its result, inferred as `<context @ 2:15>{y:Int;z:Int}`. This is similar to how rule *T-Rwr* in Figure 3 expects the body of the rewriting to contain context brand y, and removes that brand from the final result.

4.4 Type-Level Computations and Evidence

Context Disjointness. A problem with context polymorphism as presented above arises when a refined abstract context is instantiated with a concrete type that is incompatible with the refinement. An example of this would be the call `intro(code"?s : Int")`, whose result type would be the problematic structural type `{s: Int & String}`. This type is simply not a realizable context, making the result of such a call unusable. An actual, subtler problem arises when we refine a context viewed as abstract with a variable that it already contains, *and capture this variable* before the context type is concretized. An example of this can be composed with the `intro` and `outro` seen above:

```
def compose[C](x: Code[Int, C]) = outro(intro(x))
compose(code"?s : Int") : Code[String => String, {s: Int}]  →  code"(s: String) => s take s"
```

Observe that the result **`code`**`"(s: String) => s take s"` is ill-typed! The problem is that we introduce a mismatch between the static semantics of contexts, handled by quasiquotes at compilation time, and the dynamic semantics of free variables it is supposed to represent.

The solution adopted in Squid is to disallow arbitrary refinements of abstract contexts. In quasiquote macros, whenever an abstract context `C` is intersected with any other context `D`, an implicit *disjointness* evidence of type `C <> D` is searched for. If no such evidence is found, the quasiquotation fails. Type `<>[N,C]` is a simple parametric class with a private constructor, so that Squid only can create instances of it. All evidence of type `A <> B` are generated automatically for all appropriate concrete contexts by an implicit Scala macro, that checks that `A` and `B` share no common field names. Other instances are obtained by composition of implicit assumptions. As a

consequence, the definitions of `intro`, `outro` and `compose` seen above do not compile anymore. The new valid definitions follow:

```
type No_s[C] = C <> {s: Any}  // Abbreviation to avoid repetition
def intro[C](n: Code[Int, C])(implicit ev: No_s[C]) = code"(?s : String) take $n"
def outro[C](m: Code[String, C{ s: String }])(implicit ev: No_s[C]) = code"(s: String) => $m"
def compose[C](x: Code[Int, C])(implicit ev: No_s[C]) = outro(intro(x))
```

Note that the types of the variables have no importance for disjointness evidence, so `C <> {s: Any}` and `C <> {s: String}` are interchangeable. With these new definitions in place, the offending term that exposed the unsoundness, `compose(code"?s : Int")`, is now rejected with a compile-time error that reads "`Cannot prove that {s: Int} <> {s: Any}`."

Run and Closed Terms. Squid's `run` method is type-safe, as it statically rejects the evaluation of code that potentially contains free variables. This is achieved by making `run` require an implicit parameter which acts like an evidence [Oliveira et al. 2010] that the context of the term being ran is the empty context. We reproduce the signature of `run` below, as it appears as part of the `Code[+Typ,-Ctx]` class:

```
def run (implicit ev: Ctx =:= {}): Typ  =  ...
```

The implicit parameter `ev` expresses a requirement for an *evidence* that `Ctx` be the empty context `{}`. Evidence of the form `A =:= B` are generated by Scala's standard library when the subtyping relations `A <: B` and `B <: A` are satisfied. As a result, it is impossible to call `.run` on a term that is not closed. For example, `code"?x:Int".run` results in a compilation error reading "`cannot prove that {x:Int} =:= {}`," while `code"val x = 123; ${ code"println(?x)" }".run` compiles[16] and prints `123` to the console. This approach subsumes the use of environment classifiers as originally implemented in MetaOCaml [Kiselyov 2014; Taha and Nielsen 2003].

4.5 Abstracting Over Names

In the system we have seen so far, the names of free variables are not first-class. For instance, there is no way to make a function that manipulates code of type `Code[Int,{`v`:Int}]` *for all* possible names v. This is limiting, especially when we want to define recursive functions that introduce new non-conflicting names into a context on each recursive call [Nanevski 2002]. Squid provides a natural way to abstract over individual names, using context abstraction (Section 4.3) and Scala's path-dependent types [Amin et al. 2016]. Squid provides the `Variable` data type, with interface:

```
class Variable[Typ] {  type Ctx >: Fresh ; def toCode: Code[Typ,Ctx]  // >: indicates supertype bound
                       def substitute[T,C](pgrm: Code[T, Ctx & C], v: Code[Typ,C]): Code[T,C]  }
```

An instance of `Variable` represents a free variable with a unique name (internally, Squid generates a fresh name on every instantiation). This is encoded by *each* instance having a *separate* type member `Ctx` representing that name — the `Fresh` supertype bound[17] on `Ctx` indicates that `Ctx` is "fresh," in the sense that it does not conflict with any other names. We can obtain a reference to a free variable via its method `toCode`, and we can (partially) close a program fragment `pgrm` in which the variable is free by substituting every one of its occurrences using method `substitute`. Note that nothing explicitly says that type `Ctx` should contain a single free variable, but the existence

[16] Without a type annotation, free variable `x` is inferred to be of type `Any` — the type expected by `println`.

[17] `Fresh` is a phantom type used to mark fresh contexts. We cannot use *subtype* bounds for marking, because for any `C`, `C & Fresh <: Fresh`, so we could tag any open term as having a fresh context, as `Code[Int,C] <: Code[Int,C & Fresh]`.

of `substitute` indirectly constrains it to. We can now express the (path-dependent) type of a function that manipulates code open in some free variable ν without referring to that variable's name: `(ν:Variable[Int]) => (p:Code[Int,ν.Ctx]) => ...`

Having the ability to write functions that recursively build contexts made of abstract names is particularly valuable when we want to extract an arbitrary number of bindings via pattern-matching. Such a case arose in our previous work [Parreaux et al. 2017a], in the context of stream fusion: we needed to analyze the argument in calls to the stream `flatMap` function, in order to separate the main lambda abstraction passed from its captured enclosing state, so that we could make that state "resettable" by turning every bound value into a bound mutable reference that could be reset to its initial value at will. To achieve this, we had to resort to using an unsafe scope extrusion mechanism that, when misused, could create unbound variable errors at runtime (the `close` function).

We now provide a safer algorithm that achieves the same goal. The function below takes a program made of let-bindings followed by one lambda abstraction, and turns that into a program that returns a tuple made of one lambda abstraction with the same semantics, along with an effectful thunk that, when executed, *resets* the state of that lambda. For mutable references, we use a `Ref` data type with the usual `!r` and `r := v` operations for getting and setting the value, respectively:

```
def rec[T, C >: Fresh](p:Code[T,C], reset:Code[Unit,C]): Option[Code[(T,() => Unit),C]] = p match {
  case code"val x: $t = $xv; $body" =>
    val v = new Variable[Ref[xt.Typ]] ; val freshBody = body.x ~> code"!${v.toCode}"
    rec(freshBody, code"$reset; ${v.toCode} := $xv") match { case Some(r) =>
      val unfresh = v.substitute(r, code"?xr : Ref[$xt]")
      Some(code"val xr = Ref($xv); $unfresh")                case None => None }
  case code"(a: $ta) => $body" => Some(code"(((a: $ta) => $body): T, () => $reset)")
  case _ => None }
```

Function `rec` recursively analyses a program's binding structure, wrapping each bound value into a `Ref`, and accumulating a `reset` expression representing how to reset these references. Notice how we are able to recursively call `rec` on `freshBody`, which has type `Code[T, C & v.Ctx]`, without requiring a `C <> v.Ctx` disjointness evidence (Section 4.4); this is thanks to the fact that Squid sees `v.Ctx` is a super type of `Fresh`, meaning that it is disjoint from any context except itself.

As an example usage, consider the following invocation (where `"""` delimits multi-line strings):

```
rec(code"val x = readInt; val y = Ref(x); (a: Int) => {y := !y + 1; a + !y}", ir"()")
  →  Some(code""" val x_0 = Ref(readInt); val x_1 = Ref(Ref(!x_0));
                  ( (a_1: Int) => {!x_1 := !(!x_1) + 1; a_1 + !(!x_1)},
                    () => {(); x_0 := readInt; x_1 := Ref(!x_0)}      ) """)
```

4.6 Use of Runtime Reflection and Metaprogramming

Implementation of `run` and `compile`. Method `run` is implemented using Java reflection to load the classes mentioned in the program fragment and execute their methods. An alternative method `compile` invokes the Scala compiler *at runtime* to produce efficient JVM bytecode from a program fragment, and then execute it without any interpretative penalty. This enables Multi-Staged Programming [Taha and Sheard 1997] (MSP), a form of explicit partial evaluation. In MSP, the original program generates a program at runtime (first stage), which may in turn generate new programs (second stage, etc.), each time removing computations that are known at the current stage, so that an efficient implementation is finally synthesized that executes faster than its unstaged counterpart.

Subtype Checking in Code Pattern Matching. As mentioned in the context of $\lambda^{\{\}}$ in Section 3.3, Squid makes use of Scala runtime type representations, that it packages with the program fragments. This is because subtyping checks are performed at runtime to guide pattern matching on those fragments.[18] Thanks to Scala's reflection features, we perform subtyping checks at runtime, leveraging Scala's advanced type system almost for free. For example, pattern `case code"$ls: Seq[AnyVal]"` should match `code"List(1,2,3)"` because `List[Int] <: Seq[AnyVal]`, but should not match something like `code"List(4.toString)"` or it would lead to inconsistencies in reconstructed programs (cf., `String` $\not<:$ `AnyVal`). Note the necessity to annotate holes for which Scala cannot locally infer a type, like for `$ls` in the pattern example above. In contrast, pattern `code"Math.pow($x,$y)"` is fine because `Math.pow` is not overloaded nor polymorphic and only works with arguments of type `Double`.

5 A REAL USE CASE: QUERY COMPILERS

In this section, we discuss a common application of metaprogramming techniques: query compilation, which is currently an active area of database research. We have built a number of query compilers over the past years, including DBToaster [Ahmad and Koch 2009; Koch et al. 2014] and LegoBase [Klonatos et al. 2014; Shaikhha et al. 2016], which had their part in starting and accelerating this trend. Building these systems required substantial effort, due to the need for generating low-level database code with state-of-the-art performance from queries expressed in complex high-level languages (like SQL). Most existing query compilers are difficult to maintain because they work by basic template expansion, generating all the code in a single pass. To better separate the concerns of achieving advanced code optimization, one needs to design several independent transformation passes corresponding to different levels of abstraction [Shaikhha et al. 2016]. These passes should be statically type- and scope-checked to avoid potential mistakes. Squid is an answer to the metaprogramming needs discovered while iterating over the design of these compilers.

Squid is already used as part of real systems such as LegoBase, but to best explain the kinds of transformations used in our systems, we have designed a simpler, stripped down query compiler built entirely with Squid, available online.[19] In the rest of this section, we describe two of its central transformations: schema specialization and row-to-column store transformation.

5.1 Schema Specialization

Relational databases work by keeping some metadata (called the data dictionary) that represents what type of data is stored and its relation with the logical schema of the database. In a classical database system, this metadata is processed at run-time to determine how the data should be accessed and modified, given a high-level logical specification obtained from a query. This incurs high interpretive overhead, as it means that schema information has to be read over and over again, resulting in much repeated work, and that data accesses have to go through indirection.

The goal of the Schema Specialization transformer is to *optimize* or *stage away* all this overhead, specializing a query program to the current schema of the database (once it stops changing) and removing most of the indirection that would normally happen at query evaluation time. This transformer works similarly to partial evaluation, where the speculative rewrite rules of Squid are used as some form of dynamic *binding time analysis* [Jones et al. 1993], to extract the static parts from arbitrary programs. For example, we specialize query programs that use schemas expressed as lists of field information and completely remove that list data structure from the residual programs.

[18] Note that type information needs only be associated with program fragments, and not with current-stage values, which means we introduce no runtime overhead for normal computations not involving metaprogramming.
[19] This example and others can be found on the Squid open source repository: https://github.com/epfldata/squid/.

Consider the following data structures, used as the basis of a high-level query execution engine. To simplify the presentation, we assume that all columns of the relation are of type `String`. (In practice, modular abstraction with Scala path-dependent types can be used to abstract over the types, similar to `CodeType` in Section 4.1.) Instances of class `Row` internally store a list of column values, and instances of `Schema` store the list of the names associated to each of these columns:

```
class Row(values: List[String], size: Int) { ... } ;  class Schema(columnNames: List[String]) { ... }
```

We want to transform a query program such as `val s0 = new Schema("name","age"); Relation.scan ("data.csv",s0).project(Schema("age")).print` into a program where the `scan`, `project` and `print` methods are inlined to their underlying loop structures, which use methods `Row.getField` and `Schema.indicesOf`, so that we can then remove the schema data structure entirely. In the excerpt below, we show one particular rule of the schema specializer:

```
case code"val s = new Schema(List($colNames*)); $body" =>  // 'colNames' extracts a variable number
  (body fix_rewrite {                                      // of arguments as a sequence of terms
    case code"${body.s}.columnNames => code"List($colNames*)"
    case code"($r: Row).getField(${body.s}, $name)" if colNames.contains(name) =>
      val index = colNames.indexOf(name); code"$r.getValue(${Const(index)})"
    case code"${body.s}.indicesOf(List[String]($colNames2*))" =>
      val columnIndexMap = colNames.zipWithIndex.toMap
      val indices = colNames2.map(columnIndexMap).map(Const)
      code"List($indices*)" }).s ~> abort() }
```

The asterisk at the end of `$colNames*` indicates that we are extracting a variable list of arguments, giving `colNames` type `Seq[Code[String,_]]` (omitting it, we would match a single argument).

After this transformer is applied, we execute a general-purpose `List` partial evaluator (also written using Squid) to remove all schema indirections from the program. The following transformers in the pipeline of our query compiler then transform collections of rows (which are internally backed by a `List` of fields) into collections of tuples (which provide faster access to their components), with calls to `row.getValue(i)` with a constant index `i` are converted into tuple accesses.

5.2 Row-to-Column Store Transformer

Classical relational database systems such as IBM DB2, Oracle, and Microsoft SQL Server are "row-stores," meaning that they store all their data records one after the other in memory. However, many recent systems, such as Vertica, SAP HANA, and others, have experimented with a "column-store" system where, for each fields of the records of a particular table, a separate storage structure is used — column stores are a very prominent research topic in databases, starting with C-store [Stonebraker et al. 2005]. Each approach has pros and cons, but database systems are currently either developed one way or the other, with no way to reconfigure them after the fact.

In our previous work [Klonatos et al. 2014; Shaikhha et al. 2016], we showed how to automatically translate one kind of system to the other. Squid makes that transformation type-safe (i.e., more robust) and much easier to express, as we saw in Section 2.5 — in essence, the array-of-structs to struct-of-arrays optimization shown in Figure 1 corresponds to the row-store to column-store transformation of databases, when expressed in the context of in-memory query compilation.

6 RELATED WORK

We now present some related work, starting from existing real-world quasiquotation systems on to multi-stage research calculi, type-safe representations of code manipulation, and query compilation.

6.1 Existing Quasiquotation Systems

While the idea of quasiquotation is old [Quine 1940], Lisp was the language that pioneered its usage as a metaprogramming construct [Bawden et al. 1999]. Treating code as data meant that no special restrictions or mechanisms were in place to prevent common errors associated with code manipulation, such as unintended variable capture (lack of hygiene), scope extrusion and type mismatches (lack of static typing). Scheme introduced facilities to write *hygienic macros* [Abelson et al. 1991; Culpepper and Felleisen 2004; Kohlbecker et al. 1986] using a safer form of quotation that separates identifiers appearing at different compilation phases (i.e., distinguishing identifiers introduced by a macro from those present in the original program). However, within a single phase it is still possible to observe unintended variable capture. Rhiger [2012a] proposed a finer-grained hygiene system for Scheme-like *code quasiquotes*, but it does not support pattern matching on code values. The idea of code quasiquotation was picked up in a statically-typed context by Taha and Sheard [2000] with MetaML (and subsequently MetaOCaml [Taha 2004]) to enable Multi-Stage Programming (MSP). The approach was ported to compile-time macros by Ganz et al. [2001] with MacroML. In these systems, quasiquotes can only *generate* and not *inspect* code — though MacroML has some limited form of pattern–template expansion that borrows from Scheme's hygienic macro system. In these approaches, quasiquotes are a direct extension to the type system of ML which provides static guarantees about the code they generate: it is well-typed and well-scoped, except in the presence of imperative effects, which can lead to scope extrusion (cf., Section 1.4).

With Template Haskell (TH), Sheard and Jones [2002] introduced compile-time metaprogramming to Haskell using quasiquotes which had some notions 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 quasiquotes in pattern matching. However, a general quasiquote syntax (not restricted to code quasiquote) was later added to Haskell by Mainland [2007] and could in principle be used to enable quasiquote-based code pattern matching in Haskell. 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]. This is the very system used by Squid, with the difference that Squid uses type-aware macros to create an advanced extension to Scala's type system, enabling static checking that makes quasiquotation safe. 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 that system does not allow the expression of open code and does not support pattern matching, greatly limiting its usefulness.

Several other languages such as F# [Syme 2006] support different flavors of quasiquotes that fall into the categories defined above. Table 1 summarizes the features supported by quasiquotes in our paper and in several other systems. The Stratego snippet uses an example object language, but Stratego is not tied to any particular language. The asterisk (*) on the "well-scoped" and "hygienic" criteria for TTH denotes that these properties are achieved by forbidding any effects in the code generator, which can be restrictive and prevents e.g., effectful let-insertion [Kameyama et al. 2014].

6.2 Previous Squid Implementations

In its original implementation [Parreaux et al. 2017b], the Squid type-safe metaprogramming framework provided statically-typed quasiquotes, but with limited pattern matching capabilities. The only way to match bindings was to use higher-order pattern variables (similar to what was proposed by Sheard et al. [1999]), which means that matching the body of a binding construct necessarily resulted in a function term, so one could never really separate open code from its

enclosing binding. This posed problems when one wanted to change the *nature* of a binding (such as what happens in Figure 1 and in the example of Section 4.5), or when one wanted to open a binding, explore its body, and drive the reconstruction of that binding based on information gathered from the body — indeed, the reconstruction of the binding structure had to be set up before we could actually see the body. Squid was used to enable quoted staged rewriting, an approach to library-defined optimizations [Parreaux et al. 2017a]; in that work, we needed to work around these limitations and used an unsafe '`close`' function to temporarily treat some open term as closed. Misuses of that construct could lead to scope extrusion problems. In addition, users could call `.run` on arbitrary pieces of code, including open terms; this would result in runtime crashes. In contrast, Squid's new contextual quasiquote system, presented in this paper, allows for very flexible binding analysis and reconstruction, while statically preventing scope extrusion and unbound variable reference errors.

Beyond more flexible pattern matching, we also found that expressing open terms using explicit free variables was a useful metaprogramming technique in its own right. For example, it allows for more relaxed multi-stage programming patterns, as noted by Kim et al. [2006]. This technique would not be type-safe without contextual quasiquotes.

6.3 Multi-Stage Formal Calculi

Numerous multi-stage calculi based on modal logic have been developed that relate to our approach, including $\lambda^{\Box}$ [Davies and Pfenning 2001] and $\lambda^{\bigcirc}$ [Davies 1996], which inspired the design of MetaML. To prevent the evaluation of open code, Taha and Nielsen [2003] mention the possibility of reflecting context requirements in the type of terms but choose the more lightweight approach of environment classifiers, which unfortunately does not prevent imperative effects from causing scope extrusion. The systems by Nanevski [2002], Kim et al. [2006], as well as $\lambda^{[\,]}$ by Rhiger [2005, 2012b] use the contextual approach and do not have this problem. This approach was later given a foundational treatment by Nanevski et al. [2008], who presented an intuitionistic modal logic of necessity and its proof theory, and from this logic develop a contextual modal type theory, showing how modalities of necessity map to contexts. They discuss this type theory in the contexts of staging and logical frameworks. While we created our formal system by abstracting from the practical considerations of Squid pointed at throughout our paper, the type-theoretic development carried over from that line of work turns out to be strongly analogous. Most notably, the $\nu^{\Box}$ calculus by Nanevski [2002] presents code pattern matching using higher-order pattern variables (similar to what we used in [Parreaux et al. 2017a]), along with support for first-class manipulation of names (analogous to Section 4.5; but not formalized in $\lambda^{\{\}}$). In contrast, our calculus is not limited to two stages, allows for more flexible patterns that can match free variables, and lets pattern variables implicitly capture their local context. This gives us a simpler, yet more expressive account of code pattern matching. Furthermore, we allow the hygienic rewriting of all subterms of a code value at arbitrary depths[20] (the **`rewrite`** construct), unlocking the power of speculative rewrite rules.

It is worth noting that environment classifiers were eventually replaced by runtime checks in MetaOCaml because they gave "*good protection (a type error) against only rare errors, while being cumbersome always*" [Kiselyov 2017]. They also gave relatively unhelpful error messages such as "`error: 'a not generalizable in ('a, int) code`," while in Squid context errors manifest as understandable subtyping violations and disjointness proof failures such as "`Cannot prove that {s: Any} <> {s: Int}`" (see Section 4.4). Nevertheless, the problems of environment classifiers with mutable references were eventually solved via *refined* environment classifiers by Kiselyov et al. [2016], who gave a nice intuition on why using partially-ordered type variables is sufficient to solve

[20] Note that **rewrite** cannot be *encoded* with pattern matching in $\nu^{\Box}$ (or in $\lambda^{\{\}}$) as that would require polymorphic recursion.

the same problems as e.g., Rhiger [2012b]. However, whether refined environment classifiers can be extended to reason about pattern matching is an open question.

Cross-Stage Persistence (CSP) has been an important design consideration in MetaML. CSP allows a value defined in some stage to persist to a further stage. For example, `fun x →⟨ x ⟩` *lifts* an integer value into a constant code value. Complications arising from the interaction of this feature with `run` prompted the use of *explicit* CSP annotations by Taha and Nielsen [2003]. In addition, general 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). For this reason, 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 appropriately, by using static symbols and the `Const` constructor. Finally, we noticed that in some cases where MetaML requires CSP, we eschew it thanks to the use of non-lexically-scoped free variables (or explicit free variables, in Squid). For example, program `⟨fun x → ~(run ⟨⟨x⟩⟩)⟩` which in [Taha and Nielsen 2003] requires '[.]' classifier and '%' CSP annotations as in `⟨fun x → ~((run (a) ⟨%⟨x⟩⟩)[b])⟩`, can be written without any notion of CSP or classifiers in $\lambda^{\{\}}$ as $\lceil \lambda x : \mathbf{Int}.\ \lfloor \mathbf{run}\ \lceil \lceil x \rceil \rceil \rfloor \rceil$ and in Squid as `code"(x: Int) => ${ code{code"?x:Int"}.run }"` (where `code{...}` is an alternative syntax for `code"..."` that helps with nested quotations).

6.4 Safe Program Manipulation

Guarantees about manipulated programs have been encoded via the host language's type system using techniques such as Generalized Algebraic Data Types (GADTs) [Cheney and Hinze 2003; Xi et al. 2003], Higher-Order Abstract Syntax (HOAS) [Pfenning and Elliott 1988], applicative functors and monads [Kameyama et al. 2014], and De Bruijn indices [Carette et al. 2009; Sheard et al. 2005]. However, these are often heavyweight and impose a significant cost on domain experts, who have to deal with complicated type encodings, whereas they would just like to express code transformations as simple rewrite rules. In particular, we found that GADTs are hard to manipulate in systems like Haskell and Scala [Giarrusso 2013; Rompf 2016]. "Type-based embedding" systems like LMS [Rompf and Odersky 2010] use implicit conversions to compose code fragments, but this approach is complicated [Jovanovic et al. 2014] and is not applicable to code pattern-matching.

FreshML [Shinwell et al. 2003] is an extension of ML specifically designed to soundly manipulate variable bindings in metaprograms. Similarly, Cαml [Pottier 2006] allows metaprogrammers to define binding specifications that help write programs dealing with α-conversion. These systems are powerful, but they are not directly concerned with whether manipulated programs are well-typed.

Stratego [Visser 2002] is a system of composable program transformations that can express rewrite rules using the concrete object syntax, which makes it closely related to quasiquote-based approaches. The major difference with our approach is that Stratego deals with external DSLs, and that its transformations are not statically typed, so they only offer syntactic guarantees about generated programs. Several approaches base program analysis and transformation on variants of the visitor pattern [Hudak 1998; Ureche et al. 2015]. They are appropriate for a certain range of transformations that only access one level of program trees, but scale poorly to more advanced use-cases.[21] Being able to pattern-match and discover the shape of subprograms is an invaluable asset, making analyses and rewritings both concise and powerful. GHC rewrite rules [Jones et al. 2001] provide a simple interface for domain exerts to write transformations, but they are limited to simple rewritings (syntax expansion similar to hygienic macros in Scheme).

Squid focuses on manipulating *expressions*. It is not clear how a type system would capture type-safe modifications of higher-level entities such as modules and classes (where transformations

[21] Private communication with the author of [Ureche et al. 2015], March 2016.

are not usually type-preserving). Some approaches like SafeGen [Huang et al. 2011] and Morphing [Huang and Smaragdakis 2011] have attacked the problem of transforming such constructs, using custom rules and external tools such as theorem provers to make sure that some desirable properties be preserved throughout these transformations. However. these systems have provided very little in the way of transforming *expressions*, so we view them as complementary counterparts to Squid.

6.5 Query Compilation

Query compilation has been employed in database systems since the dawn of the relational database era: the very first relational database system, IBM's System R, used query compilation in its early prototypes, but this approach was quickly abandoned in favor of query *interpretation*. Chamberlin et al. [1981] explain that this was ultimately due to the impracticality of writing and maintaining code generators for query engines, rather than the query engine code itself, in this early time of databases, when architectures and algorithms were still very much in flux and subject to experimentation. What is not explicitly stated there, though very clear, is that modern metaprogramming would have helped making the construction of query compilers much more manageable and sustainable.

Recently, also thanks to advances in programming languages and technologies such as LLVM, query compilation has returned to the limelight of databases, with commercial systems such as StreamBase, IBM Spade, Microsoft's Hekaton, Cloudera Impala, and MemSQL employing it. Academic research has also intensified [Ahmad and Koch 2009; Armbrust et al. 2015; Crotty et al. 2015; Karpathiotakis et al. 2015; Klonatos et al. 2014; Koch 2010, 2014; Koch et al. 2014; Krikellas et al. 2010; Nagel et al. 2014; Neumann 2011; Rompf and Amin 2015; Viglas et al. 2014].

7 CONCLUSION

We showed how to best bring together the advantages of analytic and statically-typed quasiquotes. We formalized the approach as $\lambda^{\{\}}$, a multi-stage calculus with pattern matching on code values that allows safe scope extrusion and rewriting of open code. We have demonstrated Squid, an embedding of $\lambda^{\{\}}$ in Scala as a macro library, and shown how it enables type-safe metaprogramming. In particular, we introduced "speculative rewrite rules," an important class of optimizations based on flexible manipulations of variable bindings. We used these techniques to implement several query compiler optimizations, giving us confidence that they scale to real-world use cases.

ACKNOWLEDGMENTS

We would like to thank Samuel Grütter for his continued help on earlier versions of this paper, as well as Vlad Ureche, Georg S. Schmid, Milos Nikolic, Daniel Lupei, Dmitry Petrashko, Manohar Jonnalagedda, and Sandro Stucki. We thank the anonymous reviewers for their insightful remarks and feedback, and our shepherd Éric Tanter, for his guidance.

This work was supported by NCCR MARVEL of the Swiss National Science Foundation, and by the US National Science Foundation under Grant No. 1319880 and Grant No. 1521539. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US 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.

Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *PVLDB* 2, 2 (2009), 1566–1569.

Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *WadlerFest 2016*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer.

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1383–1394.

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 *8th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 169–178.

Alan Bawden et al. 1999. Quasiquotation in Lisp. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*. ACM, 4–12.

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.

Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1981. A History and Evaluation of System R. *Commun. ACM* 24, 10 (1981), 632–646.

Arthur Charguéraud. 2012. The locally nameless representation. *Journal of Automated Reasoning* 49, 3 (2012), 363–408.

James Cheney and Ralf Hinze. 2003. *First-class phantom types*. Technical Report. Cornell University.

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2015. Tupleware:" Big" Data, Big Analytics, Small Clusters.. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*.

Ryan Culpepper and Matthias Felleisen. 2004. Taming Macros. In *Third International Conference on Generative Programming and Component Engineering (GPCE) 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 225–243.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 151–160.

Rowan Davies. 1996. A temporal-logic approach to binding-time analysis. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*. IEEE, 184–195.

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *Journal of the ACM (JACM)* 48, 3 (2001), 555–604.

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, 5:1–5:4.

Shan Shan Huang and Yannis Smaragdakis. 2011. Morphing: Structurally Shaping a Class by Reflecting on Others. *ACM Trans. Program. Lang. Syst.* 33, 2 (Feb. 2011).

Shan Shan Huang, David Zook, and Yannis Smaragdakis. 2011. Statically Safe Program Generation with SafeGen. *Sci. Comput. Program.* 76, 5 (May 2011), 376–391.

Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)* 28, 4es (1996), 196.

Paul Hudak. 1998. Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse (ICSR '98)*. IEEE Computer Society, Washington, DC, USA, 134–.

Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.

Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop*. ACM SIGPLAN.

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.

Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. 2015. Just-in-time data virtualization: Lightweight data management with ViDa. In *CIDR*.

Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. 2006. A polymorphic modal type system for lisp-like multi-staged languages. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 257–268.

Oleg Kiselyov. 2014. The design and implementation of BER MetaOCaml. In *International Symposium on Functional and Logic Programming*. Springer, 86–102.

Oleg Kiselyov. 2017. MetaOCaml – an OCaml dialect for multi-stage programming. https://web.archive.org/web/20170725111517/http://okmij.org/ftp/ML/MetaOCaml.html

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers. In *Asian Symposium on Programming Languages and Systems*. Springer, 271–291.

Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-Level Language. *PVLDB* 7, 10 (2014), 853–864.

Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*. ACM, 87–98.

Christoph Koch. 2014. Abstraction Without Regret in Database Systems Building: a Manifesto. *IEEE Data Eng. Bull.* 37, 1 (2014), 70–79.

Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal* 23, 2 (2014), 253–278.

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.

Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *Proc. International Conference on Data Engineering (ICDE)*. 613–624.

HyoukJoong Lee, Kevin J Brown, Arvind K Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro* 31, 5 (2011), 42–53.

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.

Conor McBride and James McKinna. 2004. Functional pearl: i am not a number–i am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM, 1–9.

James McKinna and Robert Pollack. 1993. Pure type systems formalized. In *Typed Lambda Calculi and Applications*. Springer, 289–305.

Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. 2014. Code Generation for Efficient Query Processing in Managed Runtimes. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1095–1106.

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.

Aleksandar Nanevski. 2002. Meta-programming with Names and Necessity. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 206–217.

Aleksandar Nanevski and Frank Pfenning. 2005. Staged computation with names and necessity. *Journal of Functional Programming* 15, 6 (2005), 893–939.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 23.

Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE '13)*. ACM, New York, NY, USA, 125–134.

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes As Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 341–360.

Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017a. 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.

Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017b. Squid: Type-Safe, Hygienic, and Reusable Quasiquotes. In *Proceedings of the 2017 8th ACM SIGPLAN Symposium on Scala (SCALA 2017)*. ACM.

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

FranĂğois Pottier. 2006. An Overview of CÎśml. *Electronic Notes in Theoretical Computer Science* 148, 2 (2006), 27 – 52. Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005).

WV Quine. 1940. Mathematical Logic. (1940).

Morten Rhiger. 2005. First-class open and closed code fragments. In *IN PROCEEDINGS OF THE SIXTH SYMPOSIUM ON TRENDS IN FUNCTIONAL PROGRAMMING*.

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

Morten Rhiger. 2012b. *Programming Languages and Systems: 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Staged Computation with Staged Lexical Scope, 559–578.

Tiark Rompf. 2016. Reflections on LMS: exploring front-end alternatives. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. ACM, 41–50.

Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 2–9.

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.

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell (Haskell '02)*. ACM, 1–16.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. 2003. FreshML: Programming with Binders Made Simple. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*. ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/944705.944729

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, 553–564.

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

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 Michael Florentin Nielsen. 2003. Environment Classifiers. *SIGPLAN Not.* 38, 1 (Jan. 2003), 26–37.

Walid Taha and Tim Sheard. 1997. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 203–217.

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

Vlad Ureche, Aggelos Biboudis, Yannis Smaragdakis, and Martin Odersky. 2015. Automating Ad Hoc Data Representation Transformations. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 801–820.

Stratis Viglas, Gavin M. Bierman, and Fabian Nagel. 2014. Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes. *IEEE Data Eng. Bull.* 37, 1 (2014), 12–21.

Eelco Visser. 2002. Meta-programming with concrete object syntax. In *Proc. International Conference on Generative Programming and Component Engineering (GPCE)*. Springer, 299–315.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150