Report on Theory of Quoted Code Patterns

NICOLAS STUCKI, FENGYUN LIU, and AGGELOS BIBOUDIS, EPFL, Switzerland

In metaprogramming, code generation and code analysis are complementary. Traditionally, principled metaprogramming extensions for programming languages, like MetaML and BER MetaOCaml, offer strong foundations for code generation. Code analysis is also paramount for common metaprogramming scenarios. A system that supports code generation and code analysis needs to provide safety guarantees for both at the same time.

In this work, we present a system and a proposed implementation for Scala 3, that supports code analysis and transformation by *pattern matching over quoted code patterns*. The system ensures that the generated code is well-typed, hygienic and well-scoped. We provide a formalization of our system and prove its soundness.

Additional Key Words and Phrases: Metaprogramming, Macros, Quasiquotes, Quasipaterns, Optimization, Rewriting, Program transformations, Pattern matching, Scala

1 INTRODUCTION

Metaprogramming is not a new concept; it has been widely studied in academic bibliography and used in industrial-strength applications in one form or another (of variable type safety). A metaprogram–any program that manipulates, generates, and merely computes a fragment of code–treats code as *data* and our thesis is that a metaprogram should be treated with first-class support for both its construction and deconstruction in a unified system.

Generative programming [Czarnecki et al. 2002] is used in scenarios such as code configuration of libraries, code optimizations [Veldhuizen and Gannon 1998] and DSL implementations [Czarnecki et al. 2003; Tobin-Hochstadt et al. 2011]. There are various kinds of program generation systems ranging from completely syntax-based and unhygienic, to fully typed [Lilis and Savidis 2019; Smaragdakis et al. 2017]. Modern macro systems, like Racket's, can extend the syntax of the language [Flatt 2002]. On the flipside, other program generation systems may provide a fixed set of constructs offering staged evaluation [Davies and Pfenning 2001; Jørring and Scherlis 1986] like MetaML [Taha and Sheard 1997] and MetaOCaml [Calcagno et al. 2003; Kiselyov 2014, 2018; Kiselyov and Shan 2010].

Inspecting code on the other hand can be applied by reflecting over abstract-syntax trees of object code in a verbose and potentially unportable manner across different compiler versions. Scala 3 supports only generative metaprogramming [Stucki et al. 2018]—as its ancestors—and code analysis is offered via low-level reflection facilities over Scala's portable typed abstract-syntax tree format (TASTy) [Odersky et al. 2016]. In this work, we extend the work that has been done for Scala 3 and we introduce quoted pattern matching with first-class support for quoted patterns (or quasi-patterns) which are more succinct and simpler to use. We present our work through a formalization, we prove its soundness and we present case studies that exercise the feature we discuss next.

Contributions. We offer a simple 2-stage system that ensures the generated code is well-typed, hygienic and well-scoped. Our system supports pattern matching on code. We provide a formalization of the core of our system and prove its soundness. The system is implemented and integrated in the Scala 3 compiler.

Organization. Section 2 describes the existing system and introduces pattern matching via examples. Section 3 discusses formally our system. We present the syntax, semantics, type system and lay out the basic theorems we prove. In Section 4 we discuss the relation between the formalism and our implementation and in Section 5 we evaluate our system with case studies. The case studies present rewriting in three flavors: first an implementation of a string interpolator facility, second a

library that enables generic transformations/rewrites of code and third a macro that virtualizes fragments of code in Scala 3.

2 QUOTES AND SPLICES IN SCALA 3

Scala 3 offers quotation and splicing (also known as *anti-quote* or *unquote*) as a language feature [Stucki et al. 2018]. In this work, we extend that support with pattern matching on code values. Quoted expressions and patterns share the same syntactic form `'{...}`, while splicing and code extraction are expressed with `${...}`. We also offer shorthands for identifiers within a quoted block such as `$x` and `'x` instead of using the surrounding brackets. The following example constructs a code without any splices. In the subsequent lines we see a quoted pattern matching that *scrutinises* code for a `println` expression. Note that `$x` is used to *extract and bind* a code fragment that represents the argument of `println`.

```
val code = '{ println(1+2) }
code match
  case '{ println($x)) } =>
    '{ println("Result of " + ${Expr(x.show)} + ": " + $x) }
    // the generated code will print: "Result of 1+2: 3"
```

2.1 Quotes and Splices

Assuming that `e` is an expression, then `'e` represents the code value containing `e`. In MSP parlance, quoting *delays* the evaluation of an expression—also known as *staging*. Conversely, if `e` is a code value then `$e` will be replaced with the contents of `e`. Quotes and splices are duals of each other. For an expression `e1: T` we have `${'{e1}} = e1` and for an expression `e2: Expr[T]` we have `'{${e2}} = e2`. To keep track of what is staged and what is not, a level-tracking mechanism is utilized. Starting from level 0, the level within the scope of a quote is increased by one while the level within the scope of a splice is decreased by 1. References to bindings must be at *the same level* as their definition.

We demonstrate the aforementioned features via the canonical example of `power`. We start with an implementation without staging:

```
def power(x: Int, n: Int): Int =
  if (n == 0) 1
  else if (n % 2 == 1) x * power(x, n - 1)
  else { val y = x * x; power(y, n / 2) }
```

The staged version of `power` takes a staged parameter `x` (binding-time is denoted via the type `Expr[Int]`) and a non-staged parameter `n`.

```
def powerCode(x: Expr[Int], n: Int)(using QuoteContext): Expr[Int] =
  if (n == 0) '{1}
  else if (n % 2 == 1) '{ $x * ${powerCode(x, n - 1)} }
  else '{ val y = $x * $x; ${powerCode('y, n / 2)} }
```

2.2 Quoted patterns

Before generating any code in a macro, we usually need to recover some static information from the program. To this end, we introduce pattern matching over quotes by extending Scala's pattern syntax to include `'{...}`. Pattern matching tests whether a given value has the shape defined by a

pattern, and, if it does, binds the variables in the pattern to the corresponding components of the code value.

Quoted expressions carry the structure of the program which can be matched against quoted patterns. For example, matching `'{1+2}` against the pattern `'{1+2}` will succeed but it will fail against the pattern `'{3}`. Using a splice in the pattern will extract a subexpression as a code value. For example, matching `'{ println(1+2) }` against the pattern `'{ println($arg) }` will succeed and `arg` will be bound to code `'{1+2}`.

```
def powerOptimized(x: Expr[Int], n: Expr[Int])(using QuoteContext): Expr[Int] =
  x match
    case '{ power($y,  0) } => '{1}
    case '{ power($y,  1) } => y
    case '{ power($y, $m) } => '{ power($y, $n * $m ) }
    case _                  => '{ power($x, $n) }
```

The patterns rely on statically known types to infer types of the quote bindings. In the previous case, these were inferred from the type of `powerCode`. If the type cannot be inferred, extra type ascriptions can be added. For example, in the pattern `'{ ($f)($x) }` the bound variable `x` could have any type, which could work if rewritten as the pattern `'{ ($f: Int => Int)($x) }`.

When pattern matching on a subexpression that may contain bindings it is necessary to be extra attentive to not take a variable out of the scope where it is defined. We do not want to be able to extract `x + x` from `(x: Int) => x + x` and use it without the `(x: Int)` declaration as `x` would be out of its scope.

Consequently, the presence of any binding with a normal `$` will only match expressions that do not have free variables that are potentially defined in the pattern. To extract code that may contain bound variables defined in the pattern we use a special extractor `Lambda`. This form allows extracting code by lifting the expression into a function that receives the open terms as arguments.

```
  '{ (x: Int) => power(x, 2) } match
    case Lambda(fn) =>
      // fn = (x: Expr[Int]) => '{ power($x, 2) }
      fn('{42}) // returns '{ power(42, 2) }
    case _ => ...
```

2.3 Values persistance

In metaprogramming, there is often the need to use a value in the current stage in a later stage. The following example shows an attempt to do so that does not type check:

```
def powerConditionalUnroll(x: Expr[Int], n: Int)(using QuoteContext): Expr[Int] =
  if (n < 10) powerCode(x, n)
  else '{ power($x, n) } // error: n used at the incorrect level
```

The code does not type check because `n` is defined at the level 0 but used in level 1.

Using a value in a later stage is a general problem [Taha and Sheard 1997]. There are two ways to persist a value to be used in a later stage, either by serialization (*lifting*) or by keeping a reference to it (*cross-stage persistence*). To support macros that run at compile-time which generate code that will execute later we need *cross-platform portability*. This implies that the only safe way to use values in a later stage is through lifting.

```
def powerConditionalUnroll(x: Expr[Int], n: Int)(using QuoteContext): Expr[Int] =
```

```
  if (n < 10)
    powerCode(x, n)
  else {
    val lifted = Expr(n) // lift n into '{ n } using Liftable[Int]
    '{ power($x, $lifted) }
  }
```

We provide a composable `Liftable` typeclass defined in the library; which has the advantage of not requiring any extra language support. For `v:T`, the expression constuctor `Expr(v)` will lift the value into an `Expr[T]` provided an implementation of `Liftable[T]`. We provide `Liftable` for all primitive types and common collection data types from the standard library.

2.4 Literal value pattern

The simplest information metaprogrammers might want to retrieve is the value of literal constants. For this, we provide a standard extractor `Const`. This extractor will match an `Expr[T]` if it is a literal constant of type `T` and bind it to `c`. The example below shows a power function that takes two potentially dynamic values and pattern matches on their structure discovering any static part first.

```
def powerUnrollIfConstant(x: Expr[Int], n: Expr[Int])(using QuoteContext): Expr[Int] =
  n match
    case Const(n2) => powerCode(x, n2)
    case _ => '{ power($x, $n) }
```

We also provide a `ValueOfExpr` type class that provides the conversion from `Expr[T]` into an `Option[T]`. It is implemented using `Const` for all primitive values and can be composed to provide conversion for complex values. Conceptually `ValueOfExpr` is the dual of `Liftable`.

2.5 Type safety

Every quoted expression has a covariant type `Expr[T]` which guarantees by construction that the expression is of type `T` or a subtype of it. A splice of an `Expr[T]` must be used in code where a term of type `T` is valid and therefore when spliced, it will still be well-typed. On the other hand, an expression will only match if the pattern has the same types for the splices it contains. Therefore any `Expr[T]` that comes out of a pattern is guaranteed to have that type and will be safe to use.

All references within quoted code are lexically scoped and must be referenced from the same quotation level. In the code below, all references of `x` are within the scope of the outermost quote. Similarly, all references of `y` are scoped by the outermost splice. The latter is syntactically evident as the number of quotes and splices between the declaration and its reference is always the same.

```
'{
  val x = 42
  ${
    val y = 'x
    '{ $y * $y }
  }
}
```

The lexical scope constraint ensures that quotes with references to bindings in other quotes can only be created within the latter. Allowing the inner quotes to escape the scope of the splice where they are defined, could lead to inconsistent code.

Quoted pattern matching only allows patterns that safely match expressions without references that would escape their scope. The `Const` extractor does not return any code and hence is safe. The

parameterless splice only matches closed code and the splice with parameters requires the escaped references to be replaced before getting the actual expression.

3 FORMAL DISCUSSION

We formalize quoted patterns in a calculus called $\lambda^{\textcircled{\scriptsize◑}}$ which based on an extension of *simply typed lambda calculus*.

The system consists of two parts, first is an *implicitly-typed* language that represents what a user would write; and second an elaborated *explicitly-typed* language that ascribes terms with types. The types, as we will see next, are used as evidence, through the operational semantics. We present the syntax, then the elaboration of the implicitly-typed language into the explicitly-typed one, then the type system and last, the operational semantics. Finally, we state the basic soundness theorems.

3.1 Syntax

The implicitly-typed language and explicitly-typed language have the same language constructs, they only differ in their definition of terms.

3.1.1 *Syntax of implicitly-typed language.*

$$
\begin{array}{lcll}
e & ::= & n \mid x \mid \lambda x{:}T.e \mid e\ e \mid \texttt{fix}\ e \mid \square\ e \mid \$\ e \mid \texttt{lift}\ e \mid e \sim p\ ?\ e \parallel e & \textit{terms} \\
p & ::= & n \mid x \mid p\ p \mid \texttt{fix}\ p \mid \texttt{unlift}\ x \mid \texttt{bind}[T]\ x \mid \texttt{lam}[T]\ x & \textit{patterns} \\
T & ::= & \mathit{Nat} \mid T \rightarrow T \mid \square T & \textit{types}
\end{array}
$$

Terms e include standard constructs such as numbers (n), variables (x), lambdas ($\lambda x{:}T.e$), applications ($e\ e$) and a fix operator ($\texttt{fix}\ e$). In addition, we introduce four syntactic forms related to meta-programming:

$\square\ e$	*quoted expression, i.e. code value*
$\$\ e$	*code splice, i.e. insert code into other code*
$\texttt{lift}\ e$	*evaluate e to a number n and lift into* $\square\ n$
$e_1 \sim p\ ?\ e_2 \parallel e_3$	*pattern match on code* e_1 *against pattern p*

In a pattern match $e_1 \sim p\ ?\ e_2 \parallel e_3$, if the code e_1 (hereafter *scrutinee*) matches the pattern p, the evaluation continues with e_2 (*success branch*), otherwise it continues with e_3 (*failure branch*). The `lift` operator lifts a number to code representing the number. In [Stucki et al. 2018] the authors use type-classes for that reason.

The syntax of patterns echos that of terms: a pattern can be a number (n), a reference to a variable (x), a fix ($\texttt{fix}\ x$), an application ($p\ p$), a literal binding ($\texttt{unlift}\ x$), a quote binding ($\texttt{bind}[T]\ x$) and a function binding ($\texttt{lam}[T]\ x$). The difference between a number pattern n and a literal binding $\texttt{unlift}\ x$ is that the first matches a specific number, while the latter introduces a variable that matches any number in the code. The reference pattern x and quote binding pattern $\texttt{bind}[T]\ x$ are also different: the first matches a known variable in the enclosing environment, while the latter matches any expression that introduces a binding of type T. The patterns $\texttt{bind}[T]\ x$ and $\texttt{lam}[T]\ x$ are the only explicitly-typed patterns.

Types are relatively simple in the language, there are only three forms: a type of natural numbers (Nat), a type of functions ($T \rightarrow T$), and a type for code ($\square T$).

As a convention [Pierce 2002, Chapter 5.3.4], we assume that α-conversion is performed whenever necessary to avoid name clashes.

3.1.2 Syntax of explicitly-typed language. All terms of the explicitly typed language have an explicit type ascriptions in the syntax. Otherwise, they are identical in terms of language abstractions. We use the meta-variable t to denote the syntactic category of terms. Types and patterns, denoted by T and p respectively, are the same as in the implicitly-typed language, thus we omit them here.

$$\begin{array}{lcll} t & ::= & u{:}T & \textit{typed terms} \\ u & ::= & n \mid x \mid \lambda x{:}T.t \mid t\ t \mid \texttt{fix}\ t \mid \Box\ t \mid \$\ t \mid \texttt{lift}\ t \mid t \sim p\ ?\ t \parallel t & \textit{terms} \end{array}$$

Values in the language include numbers $n{:}Nat$, lambdas $(\lambda x{:}T_1.t){:}T_1 \rightarrow T_2$ and code values $\Box\ \hat{t}{:}\Box T$. Our language needs to keep track of the nesting of expressions ($\Box\ t$) and spliced ones ($\$\ e$); as is the preferred nomenclature we also use the term level for that. For simplicity, we support two-level quotation and splicing, thus we only allow only *plain terms* in quoted code (as opposed to quoted of quoted code). Plain terms do not contain any splices, quotes, lift or pattern matching.

$$\begin{array}{lcll} v & ::= & n{:}Nat \mid (\lambda x{:}T_1.t){:}T_1 \rightarrow T_2 \mid (\Box\ \hat{t}\ ){:}\Box T & \textit{values} \\ \hat{t} & ::= & \hat{u}{:}T & \textit{ascribed plain terms} \\ \hat{u} & ::= & n \mid x \mid \lambda x{:}T.\hat{t} \mid \hat{t}\ \hat{t} \mid \texttt{fix}\ \hat{t} & \textit{plain terms} \end{array}$$

For simplified exposition, we write $|u|$ to refer to an underlying term u without the top-level type annotation, i.e. $|u{:}T| = u$.

3.1.3 Environment definition.

$$\begin{array}{lcll} \Gamma & ::= & \emptyset \mid \Gamma, x^i{:}T \mid \Gamma;\Gamma & \textit{typing environment} \\ i & \in & \{0, 1\} & \textit{levels} \end{array}$$

The environment (Γ) contains bindings of the form $x^i{:}T$, where the number i indicates the level of the variable. This way, we may enforce that variables are used at the right level. Environments can be joined as $\Gamma;\Gamma$. We use the notation $\Gamma(x^i) = T$ to denote that x^i has type T in Γ.

3.2 Type System

The type system consists of two parts, one for each language. First, a set of elaboration rules that relate the implicitly-typed language with the explicitly-typed language. Second, a set of typing rules for the explicitly-typed language alone. Both parts share the same set of typing rules for the patterns as they have the same syntax and semantics.

3.2.1 Elaboration typing. The elaboration typing judgment $\Gamma \vdash^i e \in T \rightsquigarrow t$ presented in Figure 1 says that the term e can be elaborated into the typed term t (or $u{:}T$) at level i, under the typing environment Γ. We explain the type system below:

(1) The rule R-Nat checks natural numbers for both levels. Then ascribes n with Nat.
(2) The rule R-Var ensures that a variable may only be used at the level where it is introduced. This how we avoid variables to cross into another level. Then ascribes it with T.
(3) The rule R-Abs ensures that a lambda is well-typed for both levels. Note that when checking the body of a lambda, the environment is extended with $x^i{:}T_1$, i.e., the level of the bound variable x inherits the level of the typing judgment.

Elaboration Typing $\boxed{\Gamma \vdash^i e \in T \rightsquigarrow t}$

$$\Gamma \vdash^i n \in Nat \rightsquigarrow n{:}Nat \quad \text{(R-Nat)} \qquad \frac{\Gamma \vdash^i e \in T{\to}T \rightsquigarrow t}{\Gamma \vdash^i \texttt{fix}\ e \in T \rightsquigarrow (\texttt{fix}\ t){:}T}\ \text{(R-Fix)}$$

$$\frac{\Gamma(x^i) = T}{\Gamma \vdash^i x \in T \rightsquigarrow x{:}T}\ \text{(R-Var)} \qquad \frac{\Gamma \vdash^0 e \in Nat \rightsquigarrow t}{\Gamma \vdash^0 \texttt{lift}\ e \in \Box Nat \rightsquigarrow (\texttt{lift}\ t){:}\Box Nat}\ \text{(R-Lift)}$$

$$\frac{\Gamma, x^i{:}T_1 \vdash^i e \in T_2 \rightsquigarrow t}{\Gamma \vdash^i \lambda x{:}T_1.e \in T_1{\to}T_2 \rightsquigarrow (\lambda x{:}T_1.t){:}T_1{\to}T_2}\ \text{(R-Abs)} \qquad \frac{\Gamma \vdash^1 e \in T \rightsquigarrow t}{\Gamma \vdash^0 \Box\ e \in \Box T \rightsquigarrow (\Box\ t){:}\Box T}\ \text{(R-Box)}$$

$$\frac{\Gamma \vdash^i e_1 \in T_1{\to}T_2 \rightsquigarrow t_1 \qquad \Gamma \vdash^i e_2 \in T_1 \rightsquigarrow t_2}{\Gamma \vdash^i e_1\ e_2 \in T_2 \rightsquigarrow (t_1\ t_2){:}T_2}\ \text{(R-App)} \qquad \frac{\Gamma \vdash^0 e \in \Box T \rightsquigarrow t}{\Gamma \vdash^1 \$\ e \in T \rightsquigarrow (\$\ t){:}T}\ \text{(R-Unbox)}$$

$$\frac{\Gamma \vdash^0 e_1 \in \Box T_1 \rightsquigarrow t_1 \qquad \Gamma \vdash_p p \in T_1 \rightsquigarrow \Gamma_p \qquad \Gamma; \Gamma_p \vdash^0 e_2 \in T \rightsquigarrow t_2 \qquad \Gamma \vdash^0 e_3 \in T \rightsquigarrow t_3}{\Gamma \vdash^0 e_1 \sim p\ ?\ e_2 \parallel e_3\ \in\ T \quad \rightsquigarrow \quad (t_1 \sim p\ ?\ t_2 \parallel t_3){:}T}\ \text{(R-Pat)}$$

Fig. 1. Type-Based Elaboration

(4) The rule R-App ensures that application is well-typed at both levels, which is standard. Then combines the ascribed sub-terms into an ascribed application.
(5) The rule R-Fix ensures that fix is well-typed at both levels, which is standard. Then is uses the ascribed sub-term in an ascribed `fix`.
(6) The rule R-Lift ensures that an expression `lift` t is well-typed at level 0 by recursively checking that the expression t is well-typed at level 0 with the type Nat. The lifted expression takes the code type $\Box Nat$. Our system only supports two levels, thus there is no rule to type lifted expressions at level 1. Then the ascribed sub-term is used in a `lift` ascribed with Nat.
(7) The rule R-Box ensures a quoted expression $\Box\ t$ is well- typed at level 0 by recursively checking that the expression t is well-typed at level 1 with the type T. The quoted expression takes the code type $\Box T$. Again our system only supports two levels, thus there is no rule to type quoted expressions at level 1. Then the ascribed sub-term is used in a quoted term ascribed with $\Box T$.
(8) The rule R-Unbox is dual of R-Box and T-Box, which checks the spliced expression $\$\ t$ at level 1 by ensuring that the expression t is well-typed with the type $\Box T$ at level 0. For the same reason as above, there is no rule to type spliced expressions at level 0. Then the ascribed sub-term is used in a splice ascribed with T.

Explicitly-typed Terms Typing $\boxed{\Gamma \vdash^i t \in T}$

$$\Gamma \vdash^i n{:}Nat \in Nat \quad \text{(T-Nat)} \qquad \frac{\Gamma \vdash^i t \in T \to T}{\Gamma \vdash^i (\texttt{fix}\ t){:}T\ \in\ T} \quad \text{(T-Fix)}$$

$$\frac{\Gamma(x^i) = T}{\Gamma \vdash^i x{:}T \in T} \quad \text{(T-Var)} \qquad \frac{\Gamma \vdash^0 t \in Nat}{\Gamma \vdash^0 (\texttt{lift}\ t){:}\Box Nat\ \in\ \Box Nat} \quad \text{(T-Lift)}$$

$$\frac{\Gamma, x^i{:}T_1 \vdash^i t_2 \in T_2}{\Gamma \vdash^i (\lambda x{:}T_1.t_2){:}T_1 \to T_2\ \in\ T_1 \to T_2} \quad \text{(T-Abs)} \qquad \frac{\Gamma \vdash^1 t \in T}{\Gamma \vdash^0 (\Box\ t){:}\Box T\ \in\ \Box T} \quad \text{(T-Box)}$$

$$\frac{\Gamma \vdash^i t_1 \in T_1 \to T_2 \qquad \Gamma \vdash^i t_2 \in T_1}{\Gamma \vdash^i (t_1\ t_2){:}T_2\ \in\ T_2} \quad \text{(T-App)} \qquad \frac{\Gamma \vdash^0 t \in \Box T}{\Gamma \vdash^1 (\$\ t){:}T\ \in\ T} \quad \text{(T-Unbox)}$$

$$\frac{\Gamma \vdash^0 t_1 \in \Box T_1 \qquad \Gamma \vdash_p p \in T_1 \rightsquigarrow \Gamma_p \qquad \Gamma; \Gamma_p \vdash^0 t_2 \in T \qquad \Gamma \vdash^0 t_3 \in T}{\Gamma \vdash^0 (t_1 \sim p\ ?\ t_2 \parallel t_3){:}T\ \in\ T} \quad \text{(T-Pat)}$$

Fig. 2. Type system of explicitly-typed language

(9) The rule R-Pat checks pattern matching expressions at level 0. For an expression $t_1 \sim p\ ?\ t_2 \parallel t_3$, it first checks that the scrutinee t_1 is well-typed with the code type $\Box T_1$ at level 0. Then it checks that the pattern p is well-typed with the environment Γ and the unboxed scrutinee type T_1, and the pattern-bound variables have types specified in Γ_p. Finally, the environment Γ is extended with the pattern bindings Γ_p to check that the success branch t_2 has the type T; the failure branch t_3 has no access to pattern-bound variables, thus both are typed with just the enclosing environment Γ. Both t_2 and t_3 are typed at level 0, the same as the match expression. The conclusion of the rule combines the ascribed scrutinee, the ascribed success branch, the ascribed failure branch, and the original pattern into an ascribed pattern match.

3.2.2 Explicitly-typed type systems. The typing judgements $\Gamma \vdash^i t \in T$ presented in Figure 2 perform the same check presented in Section 3.2.1. The difference is that all terms already have a type that always aligns with the expected type.

3.2.3 Typing Rules for Patterns. The typing judgments for patterns, of the form $\Gamma \vdash_p p \in T \rightsquigarrow \Gamma_p$, are presented in Figure 3. The judgment, in its general form means that the pattern p is well-typed under the typing environment Γ. The pattern-bound variables that come from `unlift` x, `bind`$[T]$ x and `lam`$[T]$ x have types specified in Γ_p. We explain the typing rules for patterns below.

(1) The rule T-Pat-Nat type checks a number pattern. The scrutinee type must be Nat. The pattern does not introduce any bindings, thus resulting in an empty set of bindings.

Pattern Typing $\boxed{\Gamma \vdash_p p \in T \rightsquigarrow \Gamma_p}$

$$\Gamma \vdash_p n \in Nat \rightsquigarrow \emptyset \qquad \text{(T-Pat-Nat)}$$

$$\frac{\Gamma(x^1) = T}{\Gamma \vdash_p x \in T \rightsquigarrow \emptyset} \qquad \text{(T-Pat-Var)}$$

$$\frac{\Gamma \vdash_p p \in T \to T \rightsquigarrow \Gamma_p}{\Gamma \vdash_p \texttt{fix}\ p \in T \rightsquigarrow \Gamma_p} \qquad \text{(T-Pat-Fix)}$$

$$\frac{\Gamma \vdash_p p_1 \in T_1 \to T_2 \rightsquigarrow \Gamma_{p1} \qquad \Gamma \vdash_p p_2 \in T_1 \rightsquigarrow \Gamma_{p2}}{\Gamma \vdash_p p_1\ p_2 \in T_2 \rightsquigarrow \Gamma_{p1}; \Gamma_{p2}} \qquad \text{(T-Pat-App)}$$

$$\Gamma \vdash_p \texttt{unlift}\ x \in Nat \rightsquigarrow \{x^0{:}Nat\} \qquad \text{(T-Pat-Unlift)}$$

$$\Gamma \vdash_p \texttt{bind}[T]\ x \in T \rightsquigarrow \{x^0{:}\Box T\} \qquad \text{(T-Pat-Bind)}$$

$$\Gamma \vdash_p \texttt{lam}[T_1 \to T_2]\ x \in T_1 \to T_2 \rightsquigarrow \{x^0{:}\Box T_1 \to \Box T_2\} \qquad \text{(T-Pat-Abs)}$$

Fig. 3. Pattern Typing

(2) The rule T-Pat-Var type checks a reference pattern. It ensures that the variable has the scrutinee type at level 1. The pattern itself does not introduce any bindings, thus resulting in an empty set.
(3) The rule T-Pat-Fix type checks a fix pattern `fix` p. The pattern p may only match against code of the type $T \to T$. The checking is performed recursively on the sub-pattern p.
(4) The rule T-Pat-App type checks an application pattern $p_1\ p_2$. The pattern p_1 may only match against code of the type $T_1 \to T_2$ and p_2 against T_1. The checking is performed recursively on the sub-patterns p_1 and p_2. Finally, it combines the bindings returned from checking sub-patterns.
(5) The rule T-Pat-Unlift type checks a literal pattern. The scrutinee type must be Nat. It results in the binding $x^0{:}Nat$. Intuitively, we invite the reader to think that this rule pulls down values from level 1 to level 0.
(6) The rule T-Pat-Bind type checks a spliced binding pattern `bind`$[T]\ x$. It matches any code of type T and binds it to an x of type $\Box T$
(7) The rules T-Pat-Abs type checks a function binding pattern `lam`$[T_1 \to T_2]\ x$. It matches an explicit lambda of type $T_1 \to T_2$. The pattern-bound variable x is typed as $\Box T_1 \to \Box T_2$.

3.3 Operational Semantics

The small-step operational semantics of the language is presented in Figure 4. Intuitively, the evaluation of a top-level program starts at level 0 and follows the call-by-value semantics. However, there might be spliced code of level 0 inside a quoted expression, those level 0 splices are evaluated until a quoted expression only contains plain terms, i.e. when it reaches a value that represents plain code values.

To support reduction inside a quoted expression, the reduction rules are also defined at two levels. The reduction relation $t \longrightarrow^{i} t'$ means that the term t at level i may take one step to t'. We explain the rules below:

(1) The rules E-App-1 and E-App-2 are standard congruence rules and they need to exist on both levels.
(2) The rule E-Abs only exists at level 1, as in call-by-value semantics we don't reduce bodies of lambdas. However, at level 1, a function body may contain code that is at level 0, which may take a step.
(3) The rule E-Lift-Red lifts a natural number to the code that represents the value of the number. The rule E-Lift allows the sub-term to take a step.
(4) The rule E-Box and E-Unbox allow evaluating deeply interleaved code at level 0 in quoted expressions.
(5) The rule E-Splice is the only rule that enables reduction at level 1. This rule allows splicing of a code value into a bigger piece of code.
(6) The rule E-Beta is the standard beta-reduction, which substitutes the function parameter with the actual function argument. It only reduces at level 0.
(7) The rule E-Fix is a standard congruence rule, and it needs to exist on both levels.
(8) The E-Fix-Red is the standard fix-reduction, which substitutes the `fix` parameter with fix again. It only reduces at level 0.
(9) The rule E-Pat allows the scrutinee of a pattern match to take a step at level 0, as our system only allows pattern matches at level 0.
(10) The rule E-Pat-Succ and E-Pat-Fail performs pattern matching on a code value. The match is performed with a helper function *match*. If the match is successful, then the function returns a substitution function σ, which we use to transform the term t_1 to substitute pattern variables with the matched values. If the match fails, the evaluation continues with the term t_2.

The helper function *match* performs the actual pattern matching. If the match is successful, it returns a function that represents a substitution. The function, when called with a term, will substitute all pattern-bound variables with actual matched values. We explain the different cases below:

(1) A number pattern n matches the same number n in the code, it does not introduce any bindings, thus it returns the identity function.
(2) A reference pattern x matches the same variable x in the code. It does not introduce new bindings, thus it returns the identity function.
(3) A fix pattern `fix` p matches a fix operation `fix` t in the code if p matches t. If the sub-match is successful and returns the substitutions σ.
(4) An application pattern $p_1\ p_2$ matches an application $t_1\ t_2$ in the code if p_1 matches t_1 and p_2 matches t_2. If both the sub-matches are successful and return the substitutions σ_1 and σ_2 respectively, it returns the combined substitution $\sigma_1 \circ \sigma_2$.
(5) A literal pattern `unlift` x matches any number n in the code, it returns a function that substitutes x with the matched value n.

$$\frac{t_1 \longrightarrow^i t_1'}{(t_1\ t_2){:}T \longrightarrow^i (t_1'\ t_2){:}T}\ \text{(E-App-1)} \qquad \frac{t \longrightarrow^1 t'}{(\Box\ t){:}T \longrightarrow^0 (\Box\ t'){:}T}\ \text{(E-Box)}$$

$$\frac{t_2 \longrightarrow^i t_2'}{(t_1\ t_2){:}T \longrightarrow^i (t_1\ t_2'){:}T}\ \text{(E-App-2)} \qquad \frac{t \longrightarrow^0 t'}{(\$\ t){:}T \longrightarrow^1 (\$\ t'){:}T}\ \text{(E-Unbox)}$$

$$\frac{t \longrightarrow^1 t'}{(\lambda x{:}T_1.t){:}T \longrightarrow^1 (\lambda x{:}T_1.t'){:}T}\ \text{(E-Abs)} \qquad (\$\ \Box\ \hat{t}\,){:}T \longrightarrow^1 \hat{t}{:}T\ \text{(E-Splice)}$$

$$(\texttt{lift}\ n){:}T \longrightarrow^0 (\Box\ n){:}T\ \text{(E-Lift-Red)} \qquad \frac{t \longrightarrow^0 t'}{(\texttt{lift}\ t){:}T \longrightarrow^0 (\texttt{lift}\ t'){:}T}\ \text{(E-Lift)}$$

$$((\lambda x{:}T_1.t){:}(T_1 \to T_2)\ v){:}T_2 \longrightarrow^0 t[x \mapsto |v|]\ \text{(E-Beta)}$$

$$\frac{t \longrightarrow^i t'}{(\texttt{fix}\ t){:}T \longrightarrow^i (\texttt{fix}\ t'){:}T}\ \text{(E-Fix)}$$

$$(\texttt{fix}\ \lambda f{:}T.t){:}T \longrightarrow^0 t[f \mapsto \texttt{fix}\ \lambda f{:}T.t]\ \text{(E-Fix-Red)}$$

$$\frac{t_1 \longrightarrow^0 t_1'}{(t_1 \sim p\ ?\ t_2 \parallel t_3){:}T \longrightarrow^0 (t_1' \sim p\ ?\ t_2 \parallel t_3){:}T}\ \text{(E-Pat)}$$

$$\frac{match(\hat{t},\ p) = Some(\sigma)}{((\Box\ \hat{t}\,){:}T_s \sim p\ ?\ t_1 \parallel t_2){:}T \longrightarrow^0 \sigma(t_1)}\ \text{(E-Pat-Succ)}$$

$$\frac{match(\hat{t},\ p) = None}{((\Box\ \hat{t}\,){:}T_s \sim p\ ?\ t_1 \parallel t_2){:}T \longrightarrow^0 t_2}\ \text{(E-Pat-Fail)}$$

Fig. 4. Small-step operational semantics

(6) A quote binding pattern $\texttt{bind}[T]\ x$ matches any code, and it returns a function that substitutes x with the actually matched code value t_1.
(7) A function binding pattern $\texttt{lam}[T_1 \to T_2]\ x$ matches a lambda $\lambda x_1{:}T_1.t_1$ in the code. Suppose the lambda has the type $T_1 \to T_2$, the match returns a substitution that replaces x with the function lifted to the type $\Box T_1 \to \Box T_2$.
(8) Otherwise, the match is unsuccessful and it returns *None*.

$$
\begin{aligned}
match(n{:}Nat,\ n) &= Some(\lambda t.t) \\
match(x{:}T,\ x) &= Some(\lambda t.t) \\
match((\texttt{fix}\ \hat{t}){:}T,\ \texttt{fix}\ p) &= match(\hat{t}, p) \\
match((\hat{t}_1\ \hat{t}_2){:}T,\ p_1\ p_2) &= \sigma_1 \leftarrow match(\hat{t}_1, p_1) \\
&\quad\ \sigma_2 \leftarrow match(\hat{t}_2, p_2) \\
&\quad\ \sigma_1 \circ \sigma_2 \\
match(n{:}Nat,\ \texttt{unlift}\ x) &= Some(\lambda t.t[x \mapsto n]) \\
match(\hat{u}{:}T,\ \texttt{bind}[T]\ x) &= Some(\lambda t.t[x \mapsto \Box\ (\hat{u}{:}T)]) \\
match((\lambda x_1{:}T_1.\hat{t}_1){:}T_1 \to T_2,\ \texttt{lam}[T_1 \to T_2]\ x) &= Some(\lambda t.t[x \mapsto y]) \\
&\quad\ \texttt{where}\ y = \lambda x_3{:}\Box T_1.\ z{:}\Box T_2 \\
&\quad\ \texttt{where}\ z = \Box\ \hat{t}_1[x_1 \mapsto \$\ (x_3{:}\Box T_1)] \\
match(\_, \_) &= None
\end{aligned}
$$

Fig. 5. Pattern matching operational semantics

3.4 Soundness

We prove that the system is sound, the proof can be found in Appendix A. The soundness of the system consists of the following theorems:

Theorem (Elaboration). *If* $\Gamma \vdash^i e \in T \rightsquigarrow t$, *then* $\Gamma \vdash^i t \in T$.

Theorem (Progress). *If* $\emptyset \vdash^0 t \in T$, *then* t *is a value or there exists* t' *such* $t \longrightarrow^0 t'$.

Theorem (Preservation). *If* $\Gamma \vdash^i t \in T$ *and* $t \longrightarrow^i t'$, *then* $\Gamma \vdash^i t' \in T$.

The first theorem says that a well-typed source program elaborates to a well-typed program in the explicitly-typed language. The progress and preservation theorems ensure that a well-typed program does not get stuck at run-time.

The soundness of the system ensures that during the evaluation of a well-typed term, a variable cannot change levels, and a local variable cannot escape its definition scope to become free. Otherwise, any instance of such violation would disprove the theorems above.

The proofs of the theorems depend on some interesting lemmas, which we briefly discuss below.

Lemma (Level Progress). *For any given term* t, *we have:*

(1) *If* $\Gamma^{[1]} \vdash^0 t \in T$, *then* t *is a value or there exists* t' *such that* $t \longrightarrow^0 t'$.
(2) *If* $\Gamma^{[1]} \vdash^1 t \in T$ *and* $(\Box\ t){:}\Box T$ *is not a value, then there exists* t' *such that* $t \longrightarrow^1 t'$.

The theorem of progress depends on this lemma. It is important to prove the propositions (1) and (2) together to have a strong inductive hypothesis, as in the case $\Box\ t$ and $\$\ t$ we need to switch levels. The proposition (2) requires that $(\Box\ t){:}\Box T$ is not a value, this is the precondition for evaluation inside a quoted expression.

Lemma (Substitution). *If (1)* $\Gamma \vdash^j t_1 \in T_1$, *(2)* $\Gamma, x^j{:}T_1 \vdash^i t_2 \in T_2$ *and (3)* $j = 0$ *or* t_2 *does not contain pattern matches, then* $\Gamma \vdash^i t_2[x \mapsto |t_1|] \in T_2$.

As usual, the substitution lemma is used in the proof of the preservation theorem. Here, it gets more complex than standard substitution lemmas, as t_1 and t_2 may be at different levels, and there are four combinations in total. The subtlety in the proof is to reason that no substitution needs to be done in patterns, i.e. x may not appear in patterns.

In the case $j = 0$, the substitution lemmas reduce to the standard form. It takes advantage of the fact that x is at level 0, thus it may not appear in patterns. The typing rule T-Pat-Var ensures that only level 1 variables may appear in patterns. In the case t_2 does not contain pattern matches, it is trivial: the case for pattern match may not happen. The latter case is used in the following lemma.

Lemma (Pattern Match). *If* $\Gamma \vdash^0 (v \sim p \ ?\ t_2 \parallel t_3) : T \in T$ *and* $match(v, p) = Some(\sigma)$, *then* $\Gamma \vdash^0 \sigma(t_2) \in T$.

This lemma says that successful pattern matching preserves types of programs. It is used in proving the preservation theorem for the case of pattern matches.

One subtlety in the proof is to handle the case of pattern matching against function literals. When a function $\Box\ \lambda x_1{:}T_1.\hat{t}_1$ matches the pattern $\texttt{lam}[T_2 \to T_2]\ f$, we need to reason that the lifted function has the type $\Box T_1 \to \Box T_2$:

$$\lambda x_3{:}\Box T_1.\ (\Box\ \hat{t}_1[x_1 \mapsto \$\ (x_3{:}\Box T_1)]) : \Box T_2$$

Note that in the lifted function above, there exists the substitution $[x_1 \mapsto \$\ (x_3{:}\Box T_1)]$, where x_1 is a level 1 variable. Luckily, in this case, we know that $\hat{t}_1$ does not contain pattern matches, which satisfies the precondition of the substitution lemma.

Another subtlety with the lemma is to justify that if the pattern $p_1\ p_2$ matches the term $t_1\ t_2$, then t_1 and p_1 have the same type, t_2 and p_2 as well. This fact is supported by the following lemma:

Lemma (Correspondence). *If* $\Gamma \vdash^0 \hat{t} \in T_1, \Gamma \vdash^0 p \in T_2$ *and* $match(\hat{t}, p) = Some(\sigma)$, *then* $T_1 = T_2$.

This lemma implies that (1) patterns always have unique types; and (2) the type of a pattern corresponds to the type of the term that it matches. Though the lemma can be proved with straightforward induction, it plays a critical role in the reasoning about the soundness for pattern matches.

3.5 Discussion

In the following, we discuss several subtleties in the design of the calculus.

3.5.1 Why elaboration? We initially experimented with a calculus that does not require elaboration for the implicitly-typed language. We immediately come to a semantic difficulty: in a pattern match like $v \sim \texttt{bind}[T]\ x\ ?\ t_1 \parallel t_2$, how can we check that the value v indeed has the type T?

Ignoring the type will lead to unsoundness, as the following program shows:

$$\Box\ 5 \sim \texttt{bind}[Nat \to Nat]\ f\ ?\ \Box\ (\$\ f)\ 4 \parallel t_2$$

We may resort to the type system to figure out the type of v at run-time. However, it is non-standard to resort to typing rules in the definition of semantics, and it differs from what happens in our proposed implementation that accompanies this paper. Therefore, we find it better to define an explicitly-typed internal language, and elaborate the implicitly-typed language to it.

3.5.2 Why bindings in patterns require type annotations? Consider the following pattern match:

$$\Box\ 5 \sim \texttt{bind}[Nat]\ x\ ?\ t_1 \parallel t_2$$

It is desirable to omit the type annotation Nat, and rely on the type system to reason about it. However, care must be taken in the design of type inference, as it easily leads to unsoundness. The problem is that in a type inference system, the semantics for pattern bindings becomes unclear. Conceptually, we may distinguish two kinds of bindings: *wildcard bindings* that match any code and

type-test bindings which only match code of the specified type. A run-time type-test is mandatory for the latter to ensure soundness, while it is a waste of computation for the former.

In a type inference system, sometimes the inferred type is a wildcard binding, as the following example shows:

$$\square\ 5 \sim \texttt{bind[\_]}\ x\ ?\ \square\ (\$\ x + 4)\ \|\ t_2$$

In the code above, type inference will generate on type annotation for x, namely $\square Nat$, and no type-test is required at run-time. Sometimes, the inferred type is a type-test binding, as can be seen from the example below:

$$\square\ ((\lambda x{:}Nat.x + 3)\ 5) \sim (\texttt{bind[\_]}\ f)\ (\texttt{bind[\_]}\ x)\ ?\ f\ (\square\ \text{“}hello\text{”})\ \|\ 10$$

In the code above, a type inference system can type check the program by assigning the type $String \rightarrow Nat$ to f and $String$ to x. In this case, if a run-time type-test is omitted, the pattern match will succeed, and it gets stuck in the function call when trying to evaluate “$hello$” + 3.

To ensure soundness, a type inference system has to distinguish wildcard bindings from type-test bindings during the elaboration of binding patterns. Alternatively, it may restrict type inference only to wildcard bindings, i.e. in a pattern match $t \sim p\ ?\ t_1\ \|\ t_2$, it only constrains the type of the bound variables by the type of t and type information in p, but not t_1. Accordingly, it may introduce a different and explicit syntax for type-test bindings.

3.5.3 Scope extrusion. Given a reference to a variable, we want to make sure it never used outside the scope where it is defined. For example, we never want to start with $\square\ (\lambda x{:}T.x)$ and end up with $\square\ x$. Our design avoids scope extrusion by not adding any abstractions that could lead to it in the first place.

If a quote does not have any splice inside, then it is a value and hence will not be evaluated further. Otherwise, the quote will contain splices within which may contain references a definition in the outer quote.

$$\square\ (\lambda x{:}T.\$\ (...\square\ x...))$$

Then E-Unbox will evaluate the contents of the splice until it becomes a quoted value. In this process no rules allow this term to be moved outside the splice. Finally the E-Splice will place the contents of the splice, which may or not still contain the reference to x, and any x would be correctly bound to their definition.

Destructuring code is clearly a potential source of scope extrusion.

$$\square\ (\lambda x{:}T.x) \sim p\ ?\ t_1\ \|\ t_2$$

If we were able to get a reference to x within t_1, it would allow the construction of a term that contains x but not it's definition. The design of the system does not allow such cases to happen.

First, patterns that do not introduce any new binding are trivially safe as they do not return any part of the scrutinee. The `unlift` pattern bind a Nat and therefore cannot leak the reference.

Second, the `bind` pattern is a possible dangerous pattern if combined with the wrong pattern. The `bind` pattern nested within an `app` pattern or nested within a `fix` pattern are safe as neither of the patterns would place the `bind` within a scope defining a new binding in the quoted code. On the other hand, if we would consider the fictitious pattern $\lambda x{:}T.(\texttt{bind}[T]\ y)$, then y may match and be bound to $\square\ x$. Therefore the system simply disallows having such binding within a lambda.

Third, instead of allowing to extract the contents of the body of a lambda directly, $\texttt{lam}[T]\ x$ provides an indirection to the contents of the lambda. It extracts and lifts the lambda $\square(T \rightarrow T)$ into a $\square T \rightarrow \square T$ which when applied will replace all occurrences of the parameter with a new value or reference. With this abstraction, it is impossible to use the contents of the lambda before replacing

all the references that would get extruded. It allows the deep transformation of any lambda.

$$\square\ (\lambda x{:}T.x) \sim \texttt{lam}[T{\rightarrow}T]\ f\ ?\ \square\ (\lambda y{:}T.\$\ (f\ y))\ \|\ t2$$

as well as inspecting it's contents by replacing the references by dummy arguments that can be identified within the analysis of the code.

$$\square\ (\lambda x{:}T.x) \sim \texttt{lam}[T{\rightarrow}T]\ f\ ?\ \mathit{analyze}\ (f\ \mathit{dummyArg})\ \|\ t2$$

Therefore the set of patterns is carefully designed to disallow scope extrusions but still allow transformation and analysis of all the code.

3.5.4 α-conversion. Consider the following function (assuming arithmetic operations):

$$\begin{array}{rcl} f & = & \texttt{fix}\ \lambda \mathit{rec}{:}\square \mathit{Nat}{\rightarrow}\mathit{Nat}{\rightarrow}\square \mathit{Nat}. \\ & & \quad \lambda \mathit{acc}{:}\square \mathit{Nat}.\ \lambda n{:}\mathit{Nat}. \\ & & \quad\ \texttt{lift}\ n \sim 0\ ?\ \mathit{acc}\ \| \\ & & \quad\ \square\ ((\lambda x{:}\mathit{Nat}.\$\ (\mathit{rec}\ (\square\ (\$\ \mathit{acc}) + x)\ (n-1)))\ (\texttt{lift}\ n)) \end{array}$$

If we call this definition of power with $f\ (\square\ 7)\ 2$, we would pass through the following steps:

(1) $f\ (\square\ 7)\ 2$
(2) $\square\ ((\lambda x{:}\mathit{Nat}.\$\ (f\ (\square\ 7 + x)\ 1))\ 2)$
(3) $\square\ ((\lambda x{:}\mathit{Nat}.((\lambda x_2{:}\mathit{Nat}.\$\ (f\ (\square\ 7 + x + x_2)\ 0))\ 1))\ 2)$
(4) $\square\ ((\lambda x{:}\mathit{Nat}.((\lambda x_2{:}\mathit{Nat}.7 + x + x_2)\ 1))\ 2)$

In step (2), we can notice that the λx resulted from a step of E-Fix-Red. But then in (3) the same λx is nested within it as a second step of E-Fix-Red. To avoid any possible conflicts, all bindings within the substituted term are α-renamed. In this particular case if we did not rename, the code would have been $\square\ ((\lambda x{:}\mathit{Nat}.((\lambda x{:}\mathit{Nat}.42 + x + x)\ 1))\ 2)$ where the $x + x$ is not semantically equivalent to $x + x_2$.

In the formalization, we follow the convention [Pierce 2002, Chapter 5.3.4] to perform α-conversion whenever necessary, thus sidestep the details.

3.5.5 Pattern monomorphism. A practical limitation of the system is the lack of pattern polymorphism. Lets consider the following pattern:

$$(\texttt{bind}[T{\rightarrow}\mathit{Nat}]\ f)\ (\texttt{bind}[T]\ x)$$

If we would like to match on all such pattern, we would need to write a pattern for each T. Unfortunately, as we have the type $T{\rightarrow}\mathit{Nat}$ this would imply writing an infinite number of patterns ($\mathit{Nat}{\rightarrow}\mathit{Nat}$, $(\mathit{Nat}{\rightarrow}\mathit{Nat}){\rightarrow}\mathit{Nat}$, $(\mathit{Nat}{\rightarrow}\mathit{Nat}{\rightarrow}\mathit{Nat}){\rightarrow}\mathit{Nat}$, ...).

A possible solution, as done in the implementation, is to add a type variable X in the pattern:

$$(\texttt{bind}[X{\rightarrow}\mathit{Nat}]\ f)\ (\texttt{bind}[X]\ x)$$

Where X stands for any type that is only known at runtime. Hence the pattern would successfully match any application that has a result type Nat.

3.5.6 Cross-platform portability. The system does not assume that the code at level 0 and level 1 must execute in the same environment (or even machine). This is referred as as being *cross-platform* [Taha and Sheard 1997]. As such, a reference to a value in a different stage must always be disallowed. It is not possible to have a reference at level 1 to a value at level 0 as it might be defined in another environment that does not exist anymore. Conversely, reference at level 0 to a value at level 1 is impossible at this value does not exist yet.

Therefore the system must rely solely on lifting primitive values and combining them with quotes. The same applies for patterns, `unlift` is the only operation that allows getting a value primitive from level 1 into level 0. Then it might be combined with other patterns to extract complex values.

4 IMPLEMENTATION

The formal system specifies a core subset of the proposed implementation for Scala 3 that accompanies this paper.

4.1 Calculus and Implementation

Next, we describe the relationship between the calculus and the implementation. In the list below we provide the correspondence between the formal system and the implementation:

- the quotation $\square\ t$ is `'{t}`
- the splice $\$\ t$ is `${t}`
- the lift operation `lift` t is `Expr(t)`, which is generalized to `Liftable[T]`
- the pattern `unlift` x is `Const(x)`, which is generalized to `ValueOfExpr[T]` (whereas `Const` matches an expression of a literal value and returns the value, a complementary type, `ConstSeq`, matches an explicit sequence of literal values and returns them)
- patten matching on closed code with `bind`$[T]\ t$ is `($t:T)` or `$t`
- patten matching on open code with `lam`$[T]\ f$ is performed with `${Lambda(f)}:T` or `Lambda(f)`
- a pattern $t_1 \sim p\ ?\ t_2 \parallel t_3$ is `t1 match { case '{ p } => t2; case _ => t3 }`.
- in addition, the implementation also provides type splices `$t` as in `$xs.foreach[$t]($f)`

Consequently, we can translate the simple programs that we saw in Section 2, namely `powerOptimized`, `powerUnrollIfConstant`, and `pow` as follows:

$$
\begin{aligned}
&\texttt{powerOptimized} = \lambda x{:}\square Nat.\lambda n{:}\square Nat.\\
&\quad x \sim power\ (\texttt{bind}[Nat]\ y)\ 0\ ?\ \square\ 1 \parallel\\
&\quad x \sim power\ (\texttt{bind}[Nat]\ y)\ 1\ ?\quad y \parallel\\
&\quad x \sim power\ (\texttt{bind}[Nat]\ y)\ (\texttt{bind}[Nat]\ m)\ ?\\
&\qquad \square\ power\ (\$\ y)\ (*\ (\$\ n)\ (\$\ m)) \parallel\\
&\qquad \square\ power\ (\$\ x)\ (\$\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{powerUnrollIfConstant} = \lambda x{:}\square Nat.\lambda n{:}\square Nat.\\
&\quad n \sim \texttt{unlift}\ n_2\ ?\ pow\ x\ n_2 \parallel \square\ power\ (\$\ x)\ (\$\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{powerCode} = \texttt{fix}\ \lambda self{:}\square Nat{\to}Nat{\to}\square Nat.\\
&\quad \lambda x{:}\square Nat.\lambda n{:}Nat.\\
&\qquad \texttt{if}\ n = 0\ \texttt{then}\ \square\ 1\\
&\qquad \texttt{else if}\ n\%2 = 1\ \texttt{then}\ \square\ (\$\ x * \$\ (self\ x\ (n-1)))\\
&\qquad \texttt{else}\ \square\ (\lambda y{:}Int.\$\ (self\ y\ (n/2))\ (\$\ x * \$\ x))
\end{aligned}
$$

The scope and focus of this paper is to demonstrate quoted pattern matching formally and via a proposed implementation. Pattern matching does statically guarantee that no scope extrusions can happen within the quoted patterns. However, it is worth noting at this point that it is outside the scope of this paper to statically guarantee that all scope extrusions are caught when using impure side channels such as mutation or thrown exceptions. Consequently, following the strategy adopted by BER MetaOCaml we include *extra runtime checks* that are performed to avoid them.

4.2 Macros

We may imagine a program with macros as a quoted expression $\square\ t$ with splices inside. The expansion of a macro by the compiler is just the evaluation of one top-level (level 0) splice in the program. In this sense, the compiler is an evaluator of the input program $\square\ t$, which results in a value $\square\ \hat{t}$ that does not contain any splices inside.

In Scala, a macro is an *inline* method whose right-hand side is a splice expression. When a macro is used in a user program, its right-hand side is inlined, and the splice is evaluated and replaced by the result of the evaluation. The inlining is delayed, if the macro call is inside another inline method. Providing the macro through the `inline` method effectively hides all the `Expr` types from the caller side.

```
inline def power(x: Int, inline n: Int): Int =
  ${ powerUnrollIfConstant('x, 'n) }

def powerUnrollIfConstant(x: Expr[Int], n: Expr[Int])(using QuoteContext): Expr[Int] =
  ... // Same code as before
```

The `inline def power` is the entry point of the macro definition that will generate code by executing `powerUnrollIfConstant` at compile-time. The `inline` parameter is the annotation specifying that the parameter has by-name semantics and that it is intended for partial evaluation if possible via pattern matching.

The observant reader may notice that the method body of a definition can be arbitrarily complex. After all, a splice is just one expression. To avoid requiring a full interpreter for the Scala language when evaluating top-level splices, the right-hand side of a macro definition supports only a restricted subset of the Scala language. The right-hand side usually only calls a static method that contains the actual macro implementation. Additionally, macro definitions and usages must be located in separate files[1], so that macros are pre-compiled before their usage. The compiler evaluates the splices by interpreting the restricted subset of Scala, and reflection is used to load and execute the pre-compiled macro implementation.

5 CASE STUDIES

In this section, we present three case studies. The first shows how macros integrate with string quasiquotes to write DSLs, the second describes a general pattern to write optimizers, and the third describes a virtualization macro that transforms Scala code into a tagless format.

5.1 String interpolation macros

Scala already provides a `String` quasi-quotation system out of the box. Prefixing a string literal with some identifier will transform the string into a string interpolator. This provides a simple way to write a custom DSL system with arbitrary syntax.

It consists of a simple desugaring from `xyz"..."` into `StringContext(...).xyz(...)` where `xyz` is an extension method defined by some library. For example, the f-interpolation is transformed as follows:

```
f"Hello$x%s world$y%s"   ⟶   StringContext("Hello", "%s world", "%s").f(x, y)
```

Any such interpolator can be implemented as a macro to partially evaluate the code, check the validity of the arguments or make some arbitrary transformation. In the Scala standard library,

[1] separate files with no cyclic dependencies on their generated code

the f-interpolation is defined as a macro to check the input type validity against the format. In `f"Hello$x%s world$y%s"` it would check that `x` and `y` are indeed string. Finally, it will generate a call to `java.lang.String.format` directly. The following code sketches the implementation of the macros using an extension method `f` on `StringContext`.

```
extension on (inline self: StringContext) {
  inline def f(inline args: Any*): String = ${ fInterpol('self, 'args) }
}

def fInterpol(self: Expr[StringContext], args: Expr[Seq[Any]])(using QuoteContext): Expr[String] =
  self match
    case '{ StringContext(${ConstSeq(parts)}: _*) } =>
      ... // Check format consistency
      val stringFormat = Expr(parts.mkString)
      '{ $stringFormat.format($args: _*) }
```

Note, that since `StringContext` is defined a variable `arg`, of type `String`, we extract that list with `ConstSeq`. It is worth noting, that the `_*` operator is used to adapt a sequence so it can be used as an argument for a *varargs* field.

5.2 Optimization

Code rewriting is a fundamental operation towards implementing domain-specific languages and optimizers. Haskell RULES [Peyton Jones et al. 2001] is one of the most popular take on user-defined, rewrite rules. Library authors can write a pattern and a template of code to be spliced if the pattern is satisfied. The approach adopted by RULES can be considered non-modularly, type-safe, as the rule itself does not provide many guarantees for the generated code and relies on the optimizer and the inlining decisions of GHC.

In this case study, we gain inspiration from Squid's Quoted Staged Rewriting [Parreaux et al. 2017a] and we develop the functionality needed to create user-defined rewriters using pattern matching. We show how to create a recursive rewriter for the rules of *plus* and *power*. The rules demonstrated are obvious to the reader. What is worth noting in this specification is that our rewriter handles both expressions' rewriting with dynamic values but also with static ones. Note, that pattern matching over quotes is performed in a nested fashion.

```
val a: Int = 5
val b: Int = 6
rewrite(plus(1, 4))
rewrite(plus(plus(a, 0), plus(0, b)))
rewrite(power(4, 5))
rewrite(power(a, 5))
```

The functionality of the `Rewriter` below executes the "rules" recursively until the result does not change anymore. We demonstrate how pattern matches are nested. The first group extracts `x` and `y` from a `plus` expression and continues to examine them further to discover potential optimization opportunities. Note, that if users want to combine a pattern and `Const`, they can perform type-safe *deep pattern matching*–as in the third `case`.

```
Rewriter().withFixPoint.withPost(Transformation.safe[Int] {
  case '{ plus($x, $y) } =>
    (x, y) match
      case (Const(0), _) => y
      case (Const(a), Const(b)) => Expr(a + b)
```

```
      case (_, Const(_)) =>  '{ $y + $x }
      case _ => '{ $x + $y }
  case '{ times($x, $y) } =>
    (x, y) match
      case (Const(0), _) => '{0}
      case (Const(1), _) => y
      case (Const(a), Const(b)) => Expr(a * b)
      case (_, Const(_)) => '{ $y * $x }
      case _ => '{ $x * $y }
  case '{ power(${Const(x)}, ${Const(y)}) } =>
    Expr(power(x, y))
  case '{ power($x, ${Const(y)}) } =>
    if (y == 0) '{1}
    else '{ times($x, power($x, ${Expr(y-1)})) }
})
```

The `Rewriter` internally performs a deep-equality check to examine if two `Expr`'s (one from the current optimization pass and one from the previous recursive call) are the same. If yes, the fixpoint has been reached and the transformation stops. A `Transformation.safe[Int]` is a strict transformation that will only be applied to `Int` values and will generate `Expr[Int]` and therefore guarantees that the result will be well-typed. There is an alternative transformer that will transform `Expr[Any]` into a `Expr[Any]` as in [Parreaux et al. 2017a] which is much more flexible at the cost of requiring extra runtime checks that may fail after the transformation (`Transformation.checked[Int]`).

5.3 Virtualization

Most programming languages provide some level of extensibility to their users. C++ offers operator overloading, enabling the user to extend libraries and give syntactic sugar for operators between complex types. Monads, as a design pattern [Wadler 1990], force sequencing of operations by introducing data-dependencies with the monadic variables. Monads are by themselves not an extension of the language as such. However, we can see monads as a simple version of virtualization: a sequencing operator ; is virtualized by the monadic `bind`. F#'s Computation Expressions [Petricek and Syme 2014], Scala-virtualized [Rompf et al. 2012] for Scala and Recaf for Java [Biboudis et al. 2016] are build on top of these ideas. Based on these inspirations we create a virtualizer that transforms a block of code into its tagless representation [Carette et al. 2009]:

```
  virtualize {        //sem.Run(
    var x = 1         // sem.Bind(sem.Inject(1), ((y: Var[Int]) =>
    if (x > 0)        //   sem.Combine(sem.If(sem.Gt(sem.DeRef(y), sem.Inject(0)),
      x = x * 2       //     sem.Assign(y, sem.Mul(sem.DeRef(y), sem.Inject(2))),
    else              //
      x = 0.          //     sem.Assign(y, sem.Inject(0))),
    x                 //   sem.Return(sem.DeRef(y))))))
  }
```

The virtualizing macro uses pattern matching over quotes, the `Const` extractor and keeps track of a mapping of variables from the object of the metaprogram (the argument to `virtualize`) to the host-language's world using Higher-Order Abstract Syntax [Pfenning and Elliott 1988]. In the local scope of an expanded macro any given object of type `Symantics` (the static type of the `sem` object in the example) that implements the semantics, defines the semantics of the virtualized block (or tagless interpreter as is the preferred nomenclature in the related work). For exposition purposes, we treat all variable declarations as monadic (`Inject` injects any value in the computation). If we

assume an implementation of semantics for evaluation (merely mapping each virtualized syntax back to the original language), named `EvalSemantics` over a type `type Eval = [T] =>> () => T`, we can evaluate code that passes through this mechanism.

The `Symantics` class takes two type parameters: the return type of the macro and a higher-kinded type `M`. In this example, `Eval` is merely a type lambda to instantiate `M` (equivalent to `type Eval[T] = () => T`).

```
type Eval = [T] =>> () => T

given Symantics[Int, Eval] = new EvalSemantics[Int]
```

The `virtualize` macro invokes the recursive method `rewrite`. The latter is responsible for traversing the expression and transforming it into the tagless format. Below we show, an excerpt of this method. The pattern we chose demonstrates pattern matching with bindings. Our goal with this case is to pattern match over the binding in the source-language and introduce a new binder in the host-language. We pattern match over the structure of the variable binding and we keep track the `binder` used later in the program. A binder of type `T` in the host program is represented by a parameter to a lambda, thus we need to keep track when we need to dereference some binder in the host language later.

```
def rewrite[T: Type](e: Expr[T])(ctx: Context): Expr[M[T]] = {
  ...
  e match {
    case '{ var binder: $t = $z; ($k: `$t` => T)(binder) } => '{
     $sym.Bind[$t, T](${rewrite(z)(ctx)}, (y: Var[$t]) => ${rewrite(k)(ctx.update(binder, 'y))})
    }
  }
}
```

6 RELATED WORK

LISP has a very simple way to treat *programs as data* based on the uniform representation of programs as lists. Quotation turns fragments of (unevaluated) code into data: `'42` is a number, `'a` is a symbol, `'(+ 3 4)` is a list of the quoted constituents. Quasiquotation—with a backquote—lets us escape inside a program fragment of e.g., a whole list with a comma operator that can *unquote* and evaluate a part of the quasiquoted expression e.g., `` `(1 2 ,(+ 3 4)) ``.

Multi-Stage Programming (MSP) transfers the concepts of quotes, quasiquotes, unquotes and staged evaluation [Davies and Pfenning 2001; Jørring and Scherlis 1986] in a statically-scoped, modularly-type safe environment [Smaragdakis et al. 2017]. MSP, popularised by MetaML [Taha and Sheard 1997] and MetaOCaml [Calcagno et al. 2003; Kiselyov 2018; Kiselyov and Shan 2010], made generative programming easier [Czarnecki et al. 2002], effectively narrowing the gap of writing complex solutions of code manipulation such as: code optimizations [Veldhuizen and Gannon 1998] and DSL implementations [Czarnecki et al. 2003; Tobin-Hochstadt et al. 2011].

Fred McBride [McBride 1970] highlights the need to bridge the gap of expressing *computer-aided manipulation of symbols*. Arguing that the importance to lower the cognitive barrier of reading and writing algebraic manipulators such as algebraic simplificators and integrators, he develops the first pattern matching facility for LISP; a form that provides a *natural description* to increase the user's *problem solving potential*.

The two fundamental quasiquotation operators in Scala 3 were inspired by MetaML and BER MetaOCaml. Template Haskell [Sheard and Jones 2002] introduced Haskell to metaprogramming

using quasiquotes. Neither MetaOCaml, MetaML or Template Haskell support pattern matching with quasiquotes. MacroML [Ganz et al. 2001] used the quotation system of MetaML do define macros.

Squid [Parreaux et al. 2017a,b], a metaprogramming library for Scala, advances the state of the art of staging systems and puts quasiquotes at the center of user-defined optimizations. The user can pattern match over existing code and implement retroactive optimizations modularly. A shortcoming in Squid is that it is not lexically scoped and therefore free variables must be marked and tracked explicitly in the types.

Our calculus is closely related to λ° [Davies 1995] and $\lambda^{\Box}$ [Davies and Pfenning 2001]. These calculi capture the temporal/modal logic essence of multistage programming. The only support code generation but not code analysis.

Racket has a sophisticated macro system. On one hand, contrary to Scala, Racket is dynamically-typed and on the other, Typed Racket will type-check all expressions at the run-time phase of the given module[2]. Despite these fundamental differences, it is worth noting that Racket supports pattern matching with quasiquotes (quasipatterns). Interestingly, Racket takes one step further and much like the quasiquote expression form, unquote and unquote-splicing escape back to normal patterns which is something that we do not support.

We observe similarities between metaprogramming and proof engineering in dependent type theories. For example, Idris supports both low-level reflective capabilities and high-level pattern matching with quasiquotes similar to our work [Christiansen 2014].

7 FUTURE WORK

The system introduced in [Stucki et al. 2018] supports multistage programming with arbitrary levels and quoted types. One interesting generalization of the formalism would be to allow an arbitrary number of levels. This formalism would at least require the generalization of quotation, splicing and lift to any level. Then a further enhancement would be to generalize the pattern matching to an arbitrary level. For completeness, this step might require generalized patterns that match patterns within quotes.

Another interesting extension to the formalism would be to support type pattern variables such as in the pattern $(\texttt{lam}[X{\rightarrow}Int]\ f)\ (\texttt{bind}[X]\ x)$ which matches for an arbitrary X, which is currently implemented but not theorized. Formalization of this feature will provide a foundation for type-directed code transformations in meta-programming.

REFERENCES

Aggelos Biboudis, Pablo Inostroza, and Tijs van der Storm. 2016. Recaf: Java Dialects As Libraries. In *Proc. of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE '16)*. ACM, 2–13.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proc. of the 2nd International Conference on Generative Programming and Component Engineering* (Erfurt, Germany) *(GPCE '03)*. Springer-Verlag, Berlin, Heidelberg, 57–76.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543.

David Raymond Christiansen. 2014. Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages* (Boston, MA, USA) *(IFL '14)*. ACM, New York, NY, USA, Article 1, 9 pages.

Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Wafa Taha. 2003. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*.

Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. 2002. Generative programming. In *European Conference on Object-Oriented Programming*. Springer, 15–29.

[2]https://docs.racket-lang.org/ts-guide/caveats.html

Rowan Davies. 1995. A Temporal-Logic Approach to Binding-Time Analysis. *BRICS Report Series* 2, 51 (Jun. 1995).

Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604.

Matthew Flatt. 2002. Composable and Compilable Macros:: You Want It when?. In *Proc. of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP '02)*. ACM, New York, NY, USA, 72–83.

Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros As Multi-Stage Computations: Type-safe, Generative, Binding Macros in MacroML. In *Proc. of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy) *(ICFP '01)*. ACM, New York, NY, USA, 74–85.

Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *Proc. of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida) *(POPL '86)*. ACM, New York, NY, USA, 86–96.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Oleg Kiselyov. 2018. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Foundations and Trends® in Programming Languages* 5, 1 (2018), 1–101.

Oleg Kiselyov and Chung-chieh Shan. 2010. The MetaOCaml files - Status report and research proposal. In *ACM SIGPLAN Workshop on ML*.

Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article 113 (Oct. 2019), 39 pages. https://doi.org/10.1145/3354584

Fred McBride. 1970. *Computer Aided Manipulation of Symbols*. Ph.D. Dissertation. Queen's University of Belfast.

Martin Odersky, Eugene Burmako, and Dmytro Petrashko. 2016. A TASTY Alternative. (2016).

Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017a. Quoted Staged Rewriting: A Practical Approach to Library-defined Optimizations. In *Proc. of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Vancouver, BC, Canada) *(GPCE 2017)*. ACM, New York, NY, USA, 131–145.

Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2017b. Unifying Analytic and Statically-typed Quasiquotes. *Proc. ACM Program. Lang.* 2, POPL, Article 13 (Dec. 2017), 33 pages.

Tomas Petricek and Don Syme. 2014. The F# Computation Expression Zoo. In *Proc. of the 16th International Symposium on Practical Aspects of Declarative Languages* (San Diego, CA, USA) *(PADL '14)*. Springer-Verlag New York, Inc., 33–48.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. In *Haskell workshop*, Vol. 1. 203–233.

F. Pfenning and C. Elliott. 1988. Higher-Order Abstract Syntax. In *Proc. of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. ACM, 199–208.

Benjamin C. Pierce. 2002. Types and programming languages.

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *Higher Order Symbol. Comput.* 25, 1 (March 2012), 165–207.

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proc. of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. ACM, New York, NY, USA, 1–16.

Yannis Smaragdakis, Aggelos Biboudis, and George Fourtounis. 2017. Structured Program Generation Techniques. In *Grand Timely Topics in Software Engineering*, Jácome Cunha, João P. Fernandes, Ralf Lämmel, João Saraiva, and Vadim Zaytsev (Eds.). Springer International Publishing, Cham, 154–178.

Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A Practical Unification of Multi-stage Programming and Macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA) *(GPCE 2018)*. ACM, New York, NY, USA, 14–27.

Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *In Proc. of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. ACM, New York, NY, USA, 203–217.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages As Libraries. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 132–141.

T Veldhuizen and E Gannon. 1998. Active libraries: Rethinking the roles of compilers and libraries. In *Proc. of the 1998 SIAM Workshop: Object Oriented Methods for Interoperable Scientific and Engineering Computing*. 286–295.

Philip Wadler. 1990. Comprehending Monads. In *Proc. of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. ACM, 61–78.