Hygiene for Scala

Master thesis in Computer Science by
Denys Shabalin

Supervised by
Prof. Martin Odersky, EPFL
Jason Zaugg, Typesafe

EPFL

August 15, 2014

Contents

Chapter 1

Introduction

1.1 Background

Since the early times of Scala, there has always been a built-in way to do code reflection.

Surprisingly, even in Scala 1.x, one could find the `scala.reflect` package that featured definitions of abstract syntax trees called `Code` along with supporting data structures. In a type-directed fashion, users could request lambda expressions to be lifted to their AST representation for future introspection:

```
val x: Int = 0
val xtimesy: Code[Int => Int] = (y: Int) => x * y
```

The main goal of this functionality was to enable virtualization of Scala code, providing a facility to transform lifted code according to domain-specific rules. `Code` trees were quite incomplete, supporting only a subset of Scala syntax.

The Scala 2.2 release introduced the `Code.lift` helper utility, which provided a way to obtain ASTs of arbitrary code snippets using a high-level function call notation:

```
val x: Int = 0
val xtimes2: Code[Int] = Code.lift(x * 2)
```

`Code.lift` was in fact not a regular method but a special intrinsic whose calls were replaced with lifted AST representations of its arguments in a dedicated compiler phase.

The Scala 2.10 release, which arrived in early 2013, featured an experimental macro system [1] that provided a way to write functions that transformed ASTs at compile time. The `Code` AST used by `Code.lift` was abandoned in favor of a comprehensive Reflection API that exposed internal compiler data structures via a restricted public interface.

At the core of the newly introduced macro system was the `reify` function that could construct Reflection API trees of its argument, much like its predecessor `Code.lift` could construct `Code` trees. However, unlike `Code.lift`, `reify` didn't require a separate compiler phase and was purely macro-based [1].

In the example below, `reify` is used to define the `assert` macro that enapsulates a common debugging idiom. Note how `reify` not only creates the code that will be returned by the macro, but also lets the programmer to incorporate macro arguments into the final result.

```
package object asserts {
  def assertionsEnabled = ...
  def raise(msg: Any) = throw new AssertionError(msg)

  def assert(cond: Boolean, msg: Any): Unit = macro assertImpl
  def assertImpl(c: Context)(cond: c.Expr[Boolean],
                             msg: c.Expr[Any]): c.Expr[Unit] =
    if (assertionsEnabled)
      c.reify(if (!cond.splice) raise(msg.splice))
    else
      c.reify(())
}
```

In comparison with untyped macro systems of lisps, `reify` promised to deliver strong guarantees with respect to well-typedness of generated code and absense of inadvertent name clashes. For instance, code generated with `reify` is guaranteed to be well-typed due to the fact that only typed snippets can be spliced. Additionally, `raise` within the first `reify` block is guaranteed to bind to `asserts.raise` in accordance with the traditional intuition of lexical scoping. Both of these properties are possible due to the fact that code generated via `reify` is typechecked on snippet-by-snippet basis[1].

[1]However, additionally to being constructed via `reify`, instances of `Expr` could also be created from manually constructed, untyped trees, and once the user does that, safety is natually off the table.

Curiously, the very guarantees that `reify` aimed to provide have led to it being insufficiently flexible:

- Composition of code snippets was limited to composition of well-typed terms (instances of `Expr`) based on other well-typed terms. This might look like a good idea at first glance but ultimately it significantly restricts modularity, as it often is impossible to split snippets into small, yet still well-typed parts.

- There was no way to abstract over arity and generate code with variable amount of subtrees. For example, one couldn't generate a block that has variable amount of statements inside.

- Lastly, `reify` only provided code generation functionality. There was no support for pattern matching on existing code snippets to decompose them into smaller pieces.

The limitations of `reify` often made it necessary to use raw tree constructors which were quite verbose and easy to get wrong considering the fact that those trees were never designed for convenience of the end user but rather for simplicity of the underlying compiler.

As a response to these problems, an alternative tool for code genaration, quasiquotes, was introduced in the Scala 2.11 release [8]. The goal of quasiquotes was to emphasize flexibility of tree manipulation and completely eliminate the need to use the constructors of the data structures underlying the ASTs of the Reflection API. All aforementioned problems of `reify` were solved, albeit at the expense of safety.

With the help of quasiquotes, one might rewrite the previously demonstrated macro implementation in the following way:

```
def assertImpl(c: Context)(cond: c.Tree, msg: c.Tree): c.Tree =
  if (assertionsEnabled)
    q"if ($cond) _root_.asserts.raise($msg)"
  else
    q"()"
```

As it can be seen, quasiquotes produce untyped trees and don't require unquotees to be statically typed. As a result, one has to fully qualify referenced names to ensure that, regardless of the environment where a macro expands, they always bind to what the macro programmer intended.

While the possibility for the generated code to be ill-typed is an unpleasant downside, it is mostly compensated by the fact that the final result of macro expansion is typechecked. Also, with the help of the `Context.typecheck` API, it is also possible to pre-validate intermediate results during macro expansion whenever necessary.

Meanwhile, the possibility for inadvertent name clashes turned to be a much bigger issue in practice. The lack of connection between scoping outside of quasiquote and scoping within quasiquotes causes numerous bugs in macros which are quite hard to find and debug. Unfortunately, there is no easy solution to this problem. In scalac, name resolution is inherently tied to typechecking and one can't just perform one without the other. But at the same time one can't typecheck quasiquotes as that would lose all the flexibility advantage over `reify`.

1.2 Our contribution

As things currently stand, Scala's macro system features two ways to construct abstract syntax trees: `reify`, which provides binding integrity guarantees at the expense of flexibility, and quasiquotes, which enjoy ultimate flexibility at a cost of exposing the programmer to inadvertent name clashes.

The main contribution of this work is a formal model of hygienic name resolution and macro system that is flexible enough to provide missing safety to quasiquotes. This makes it possible to combine the best of two worlds: we get reasonable safety guarantees without sacrificing the notational convenience of quasiquotes.

Towards this goal we build a three-tier formal model:

1. In Section 2 we define a core language that mimics Scala scoping and common underlying primitives. This section roughly corresponds to state of Scala before macros (e.g. Scala 2.9)

2. Then we proceed with a model of simple unhygienic macros in Section 3. We show detailed examples that illustrate why current system works the way it does and why it's not sufficient to sustain hygienic macros. This section models Scala 2.10-2.11 typechecking and macro expansion.

3. After that in Section 4 we present an alternative name resolution and AST re-typechecking schemes with an ultimate goal of providing hygienic safety to blackbox macros written using quasiquotes.

Each section has an accompanying formalization of typechecking and runtime evaluation which serve as a foundation that makes it possible to reason about properties of a typed language with rich scoping similar to Scala. We do not formalize all of the features of quasiquotes present in Scala 2.11, only the minimal subset that is required to illustrate hygiene challenges.

We are convinced that results of this research could be incorporated into a full-fledged Scala compiler as a foundation for a hygienic macro system.

Chapter 2

Base language

2.1 Syntax

$v ::=$	Values:
$\quad l$	store location
$\quad x_i \Rightarrow t$	function value
$i,\ j,\ k ::=$	Symbols:
$\quad 0$	empty symbol
$\quad id$	globally unique id
$x,\ y,\ z ::=$	Names:
$\quad x$	name with empty symbol (x_0)
$\quad x_i$	name with any symbol
$a,\ b,\ c,\ t ::=$	Terms:
$\quad x_i$	identifier
$\quad (x_i : T) \Rightarrow t$	function
$\quad t\ t$	application
$\quad \{\ \overline{s};\ t\ \}$	block
$\quad new\ \{\ z_i \Rightarrow \overline{s}\ \}$	new
$\quad t.x$	selection
$s ::=$	Statements:
$\quad val\ x_i : T\ =\ t$	lazy val
$\quad import\ p.\_$	wildcard import
$p ::=$	Paths:
$\quad x$	identifier
$\quad p.x$	selection
$A,\ B,\ C,\ T ::=$	Types:
$\quad \{\overline{x : T}\}$	structural object type
$\quad T \Rightarrow T$	function type
$\Gamma ::= \overline{p.x_i : T}$	Environment
$\mu ::= \overline{l \mapsto \{\overline{x = t}\}}$	Store

Figure 2.1: Base syntax

Before we proceed to model hygienic quasiquotes and macro expansion we need to fully understand mechanics of current typechecking and the inherent problems it has with respect to macros and quasiquotes.

For sake of simplicity we will inspect core ideas on a simple Scala-like language with similar scoping and binding structure. This language is based on simply-typed lambda calculus with blocks, anonymous objects with self-recursion, local lazy vals and imports from objects as first-class modules. We use straightforward structural types with subtyping to model anonymous object values.

Additionally for our examples we assume that Int and Bool with corresponding constants and operations are also available as primitives in the language even though we do not define them in the base calculus to keep typing and evaluation rules small and easy to understand.

As you can see this language doesn't yet have macros or quasiquotes. In this section we'll inspect typechecking and evaluation of the base language only.

2.2 Scoping

In our language we aimed at having scoping and name resolution that are extremely close to full-blown Scala. In particular we preserved template and block scope unification that allows forward references:

```
{
  val x: Int = y
  val y: Int = 0
  x
}
```

Here x can reference y. Semantics of evaluation is lazy so when the x value is requested in the end of block it transitively requests y to be evaluated and eventually results in both x and y having value 0 with value 0 being returned from the block.

Similarly in objects any member can reference any other during its initialization:

```
new { z =>
  val div: Int = value / 10
  val rem: Int = value % 10
  val value: Int = 10
}
```

Here `div` and `rem` can reference `value` which is in the same scope but is not necessarily defined earlier than it is being used. Similarly to blocks this works due to the fact that fields are lazily initialized.

Our base language also supports wildcard imports from objects as first-class modules. Similarly to Scala this means that for each member we introduce corresponding identifiers into a scope of statements and terms below it. For sake of simplicity we do not include full-blown Scala imports because they only improve user convenience and do not bring anything new to the name resolution mechanics.

```
{
  val x: { value: Int } = new { val value: Int = 10 }
  import x._
  value // resolves to x.value
}
```

To avoid complicating the formalization we do not support import ambiguities. If two imports in the same scope imports members with the same name then latter will shadow the former.

```
{
  val x: { value: Int } = new { val value: Int = 10 }
  val y: { value: Int } = new { val value: Int = 20 }
  import x._
  import y._
  value // resolves to y.value
}
```

In Scala such code would fail to compile with import ambiguity errror.

2.3 Typechecking

To model typechecking we use two co-recursive relations for typechecking of terms ($\Gamma \vdash t \longrightarrow t' : T$) and statements ($\Gamma \vdash_{\,stats} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Gamma'$).

Similarly to Scala compiler we model expansion of identifiers to their fully-qualified paths and attribution of symbols (unique identifiers) to definitions and identifiers. Unlike in Scala symbols are attributed to identifiers only but not to member selections. This is due to the fact that we have structural type system rather than nominal one.

To model such expansions we construct lexical environment Γ as a mapping from names x to their (optional) prefix, symbol and type $\overline{p.x_i : T}$ where $p.x$ is fully-qualified version of identifier x in current scope and i is unique definition id. Path prefix is added for definitions which were brought into scope using imports and to the members within *new*. Local definitions like vals in blocks, function parameters and self definitions in *new* (z) have an empty prefix.

Symbol attribution works in a following manner: whenever we see a definition we generate a fresh id for it and add corresponding entry to Γ. Eventually when our typechecker reaches leafs of the tree we lookup identifiers in environment and replace it with their attributed versions, either with prefixes (T-PrefixedIdent) or without them (T-LocalIdent).

$$\frac{p.x : T \in \Gamma}{\Gamma \vdash x \longrightarrow p.x : T} \qquad \text{(T-PrefixedIdent)}$$

$$\frac{x_i : T \in \Gamma}{\Gamma \vdash x \longrightarrow x_i : T} \qquad \text{(T-LocalIdent)}$$

$$\frac{i = freshId \quad \Gamma,\ x_i : A \vdash t \longrightarrow t' : B}{\Gamma \vdash (x : A) \Rightarrow t \longrightarrow (x_i : A) \Rightarrow t' : A \Rightarrow B} \qquad \text{(T-Func)}$$

$$\frac{\Gamma \vdash t \longrightarrow t' : A \Rightarrow B \quad \Gamma \vdash a \longrightarrow a' : A}{\Gamma \vdash t\ a \longrightarrow t'\ a' : B} \qquad \text{(T-App)}$$

$$\frac{\Gamma \vdash t \longrightarrow t' : T \quad T <: A}{\Gamma \vdash t \longrightarrow t' : A} \qquad \text{(T-Sub)}$$

$$\frac{\Gamma \vdash t \longrightarrow t' : \{x : A,\ \overline{y : B}\}}{\Gamma \vdash t.x \longrightarrow t'.x : A} \qquad \text{(T-Sel)}$$

Figure 2.2: Term typing $\Gamma \vdash t \longrightarrow t' : T$ (Part 1)

For statements in block or object scope we first create symbols for all definitions in the scope using *attribute* meta-function to model forward references. Then we extract all definitions from the scope using *sig* meta-function and use this new custom scope to typecheck statement bodies and resolve imports within the scope. The latter is done using helper $\vdash_{stats}$ relation.

$$\frac{\begin{array}{c}\overline{s}' = attribute[\![\overline{s}]\!] \quad \overline{xs} = sig[\![\overline{s}']\!] \\ \Gamma,\ \overline{xs} \vdash_{stats} \{\ \overline{s}'\ \} \longrightarrow \{\ \overline{s}''\ \};\ \Gamma' \\ \Gamma' \vdash t \longrightarrow t' : T\end{array}}{\Gamma \vdash \{\ \overline{s};\ t\ \} \longrightarrow \{\ \overline{s}'';\ t'\ \} : T} \quad (\text{T-Block})$$

$$\frac{\begin{array}{c}\overline{s}' = attribute[\![\overline{s}]\!] \quad \overline{xs} = sig[\![\overline{s}']\!] \quad i = freshId \\ \overline{zx} = \{z_i.x : T \mid val\ x_j : T = t \in \overline{s}'\} \\ \Gamma,\ z_i : \{\overline{xs}\},\ \overline{zx} \vdash_{stats} \{\ \overline{s}'\ \} \longrightarrow \{\ \overline{s}''\ \};\ \Gamma'\end{array}}{\Gamma \vdash new\ \{\ z \Rightarrow \overline{s}\ \} \longrightarrow new\ \{\ z_i \Rightarrow \overline{s}''\ \} : \{\overline{xs}\}} \quad (\text{T-New})$$

Figure 2.3: Term typing $\Gamma \vdash t \longrightarrow t' : T$ (Part 2)

Statement typing checks that all statements are well-formed and also performs import resolution (St-Import) returning a new environment where all imports are resolved Γ'.

$$\frac{}{\Gamma \vdash_{stats} \{\} \longrightarrow \{\};\ \Gamma} \quad (\text{St-Empty})$$

$$\frac{\Gamma \vdash t \longrightarrow t' : T \quad \Gamma \vdash_{stats} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Gamma'}{\Gamma \vdash_{stats} \{\ val\ x_i : T = t;\ \overline{s}\ \} \longrightarrow \{\ val\ x_i : T = t';\ \overline{s}'\ \};\ \Gamma'} \quad (\text{St-Val})$$

$$\frac{\begin{array}{c}\Gamma \vdash p \longrightarrow p' : \{\overline{y : A}\} \\ \overline{px} = \{\ p'.x : B \mid x : B \in \overline{y : A}\ \} \\ \Gamma,\ \overline{px} \vdash_{stats} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Gamma'\end{array}}{\Gamma \vdash_{stats} \{\ import\ p.\_;\ \overline{s}\ \} \longrightarrow \{\ import\ p'.\_;\ \overline{s}'\ \};\ \Gamma'} \quad (\text{St-Import})$$

Figure 2.4: Statement typing $\Gamma \vdash_{stats} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Gamma'$

Our previous typechecking judgements also depend on trivial structural typing <: borrowed from [7] and two helper meta-functions ($attribute[\![\overline{s}]\!]$ and $sig[\![\overline{s}]\!]$).

$$\frac{}{T <: T} \quad \text{(S-Refl)}$$

$$\frac{A <: T \quad T <: B}{A <: B} \quad \text{(S-Trans)}$$

$$\frac{C <: A \quad B <: T}{A \Rightarrow B <: C \Rightarrow T} \quad \text{(S-Func)}$$

$$\frac{}{\{\overline{x : A}, \overline{y : B}\} <: \{\overline{x : A}\}} \quad \text{(S-Width)}$$

$$\frac{\forall\ (A,\ B) \in zip[\![\overline{A}, \overline{B}]\!].\ A <: B}{\{\overline{x : A}\} <: \{\overline{x : B}\}} \quad \text{(S-Depth)}$$

$$\frac{\{\overline{x : A}\} \text{ is a permutation of } \{\overline{y : B}\}}{\{\overline{x : A}\} <: \{\overline{y : B}\}} \quad \text{(S-Perm)}$$

Figure 2.5: Subtyping

$$sig[\![\overline{s}]\!] = \{\ x_i : T \mid val\ x_i : T = t \in \overline{s}\ \}$$

Figure 2.6: *sig* meta-function

$attribute[\![\emptyset]\!] = \emptyset$
$attribute[\![val\ x : T = t;\ \overline{s}]\!] = val\ x_i : T = t;\ attribute[\![\overline{s}]\!]$
  where $i = freshId$
$attribute[\![import\ p.\_;\ \overline{s}]\!] = import\ p.\_;\ attribute[\![\overline{s}]\!]$

Figure 2.7: *attribute* meta-function

2.4 Evaluation

To simplify evaluation rules we erase all information that is not needed at runtime using *erase* meta-function before actual evaluation. It removes all type annotations and imports from the tree. We can safely remove imports thanks to the fact that they have been fully resolved during typechecking.

$$\begin{array}{l}
erase[\![x_i]\!] = x_i \\
erase[\![(x_i : T) \Rightarrow t]\!] = x_i \Rightarrow t \\
erase[\![a\ b]\!] = erase[\![a]\!]\ erase[\![b]\!] \\
erase[\![\{\ \overline{s};\ t\ \}]\!] = \{\ erase[\![\overline{s}]\!];\ erase[\![t]\!]\ \} \\
erase[\![new\ \{\ z_i \Rightarrow \overline{s}\ \}]\!] = new\ \{\ z_i \Rightarrow erase[\![\overline{s}]\!]\ \} \\
erase[\![t.x]\!] = erase[\![t]\!].x \\
\\
erase[\![\emptyset]\!] = \emptyset \\
erase[\![val\ x_i : T = t;\ \overline{s}]\!] = val\ x_i = t;\ erase[\![\overline{s}]\!] \\
erase[\![import\ p.\_;\ \overline{s}]\!] = erase[\![\overline{s}]\!]
\end{array}$$

Figure 2.8: *erase* meta-function

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t_1'\ t_2 \mid \mu'} \tag{E-App1}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \tag{E-App2}$$

$$\frac{}{(x_i \Rightarrow t)\ v \mid \mu \longrightarrow [x_i \mapsto v]t \mid \mu} \tag{E-AppAbs}$$

$$\frac{}{\begin{array}{c}\{\ \overline{val\ x_i = t};\ t\ \} \mid \mu \longrightarrow \\ \{\ \overline{val\ x_i = t};\ [\overline{x_i \mapsto t}]t\ \} \mid \mu\end{array}} \tag{E-Block1}$$

$$\frac{}{\{\ \overline{val\ x_i = t};\ v\ \} \mid \mu \longrightarrow v \mid \mu} \tag{E-Block2}$$

$$\frac{l \notin dom(\mu)}{\begin{array}{c}new\ \{\ z_i \Rightarrow \overline{val\ x_j = t}\ \} \mid \mu \longrightarrow \\ l \mid \mu, l \mapsto \{\overline{x = [z_i \mapsto l]t}\}\end{array}} \tag{E-New}$$

$$\frac{t \mid \mu \longrightarrow t' \mid \mu'}{t.x \mid \mu \longrightarrow t'.x \mid \mu'} \tag{E-Sel1}$$

$$\frac{\mu(l) = \{x = a, \overline{y = b}\}}{l.x \mid \mu \longrightarrow a \mid \mu} \tag{E-Sel2}$$

Figure 2.9: Evaluation $t \mid \mu \longrightarrow t' \mid \mu'$

Chapter 3

Unhygienic macros

3.1 Syntax

$v ::=$	New values:
$\quad q"t"$	term value provided $unquotes[\![q"t"]\!] \equiv \emptyset$
$t ::=$	New terms:
$\quad q"t"$	term quasiquote
$\quad \$x_i, \$\{t\}$	term unquote
$s ::=$	New statements:
$\quad macro\ x_i(y_j : A) : B = t$	macro
$T ::=$	New types:
$\quad Term$	term type
$\Gamma ::= \overline{p.x_i : T}$	Environment (indexed by x)
$\Xi ::= \overline{macro\ x_i(y_j : A) : B = t}$	Macro environment (indexed by x)

Figure 3.1: Unhygienic macro extension

To model current state of macros in Scala we add a simple form of term quasiquotes with unquoting and corresponding data type $Term$. The combination of these features can be used to write simple unhygienic macros:

```
{
  macro plus1(x: Int): Int = q"$x + 1"
  val x: Int = 2
  plus1 x
}
```

When macro is typechecked its body is replaced by the result of the evaluation of the right-hand side and all names get resolved symbols:

```
{
  macro plus1_1(x_2: Int): Int = q"$x_2 + 1"
  val x_3: Int = 2
  x_3 + 1
}
```

If identifiers are present in macro-generated code they are resolved in the macro expansion scope rather then macro definition one. For example the following code:

```
{
  val x: Int = 1
  macro plusx(e: Int): Int = q"$e + x"
  {
    val x: Int = 4
    val y: Int = 5
    plusx y
  }
}
```

Will be expanded into:

```
{
  val x_1: Int = 1
  macro plusx_2(e_3: Int): Int = q"$e_3 + x"
  {
    val x_4: Int = 4
    val y_5: Int = 5
    y_5 + x_4
  }
}
```

Such behaviour is quite counter-intuitive and can be really confusing. At the same time it's exactly kind of scoping we have at the moment with macros and quasiquotes in Scala 2.11.

3.2 Typechecking

To typecheck code that uses macros we need to augment our existing lexical environment with compile-time knowledge about current macros in scope. To do this we add another environment Ξ of the form $\overline{macro\ x_i(y_j : A) : B = t}$. This environment is disjoint with Γ and naturally augments it.

Most existing typing rules from base language are left as-is (T-PrefixedIdent), (T-LocalIdent), (T-Func), (T-App), (T-Sub) and (T-Sel) apart from need to add Ξ to the left hand side of the typing relation.

Apart from that we need to add a new rule for macro expansion.

$$\frac{\begin{array}{c} macro\ x_i(y_j : A) : B = b \in \Xi \\ \Gamma;\ \Xi \vdash a \longrightarrow a' : A \\ q\text{"}t\text{"} = eval[\![[y_j \mapsto q\text{"}a'\text{"}]b]\!] \\ t' = reset[\![t]\!] \\ \Gamma;\ \Xi \vdash t' \longrightarrow t'' : B \end{array}}{\Gamma;\ \Xi \vdash x\ a \longrightarrow t'' : B} \quad \text{(T-MApp)}$$

Figure 3.2: Term typing extension $\Gamma;\ \Xi \vdash t \longrightarrow t' : T$ (Part 1)

This rule performs following steps to expand a macro:

1. Looks up macro signature and body in macro environment.

2. Type checks macro argument against macro argument type from the signature.

3. Evaluates macro body with macro argument replaced by the quotation of the argument.

4. Removes all symbols from the result tree using *reset* meta-function. This corresponds to *resetAllAttrs* function in Scala which tries to make tree uniform with respect to attribution. We'll discuss problems arising on the mixing of attributed and unattributed trees in Section 3.4.2.

5. Typechecks the tree after *reset* against macro result type.

6. And lastly replaces macro application with typechecked result of macro evaluation.

Additionally we also need to tweak (T-Block) and (T-New) to make them include macros in corresponding scopes.

$$\frac{\begin{array}{c}\overline{s}' = attribute[\![\overline{s}]\!] \quad \overline{xs} = sig[\![\overline{s}']\!] \\ \Xi \vdash_{macros} \{\ \overline{s}'\ \} \longrightarrow \{\ \overline{s}''\ \};\ \Xi' \\ \Gamma,\ \overline{xs};\ \Xi' \vdash_{stats} \{\ \overline{s}''\ \} \longrightarrow \{\ \overline{s}'''\ \};\ \Gamma' \\ \Gamma';\ \Xi \vdash t \longrightarrow t' : T\end{array}}{\Gamma;\ \Xi \vdash \{\ \overline{s};\ t\ \} \longrightarrow \{\ \overline{s}''';\ t'\ \} : T} \quad \text{(T-Block)}$$

$$\frac{\begin{array}{c}\overline{s}' = attribute[\![\overline{s}]\!] \quad \overline{xs} = sig[\![\overline{s}']\!] \\ \Xi \vdash_{macros} \{\ \overline{s}'\ \} \longrightarrow \{\ \overline{s}''\ \};\ \Xi' \\ i = freshId \quad \overline{zx} = \{z_i.x : T \mid val\ x_j : T = t \in \overline{s}\} \\ \Gamma,\ z_i : \{\overline{xs}\},\ \overline{zx};\ \Xi' \vdash_{stats} \{\ \overline{s}''\ \} \longrightarrow \{\ \overline{s}'''\ \};\ \Gamma'\end{array}}{\Gamma;\ \Xi \vdash new\ \{\ z \Rightarrow \overline{s}\ \} \longrightarrow new\ \{\ z_i \Rightarrow \overline{s}'''\ \} : \{\overline{xs}\}} \quad \text{(T-New)}$$

Figure 3.3: Term typing extension $\Gamma;\ \Xi \vdash t \longrightarrow t' : T$ (Part 2)

Unlike regular statements macros need to be already typechecked before they are added to the environment. Considering that fact that any definition can reference any macro we introduce another helper relation $\vdash_{macros}$ which typechecks all macros in scope before typechecking vals and imports.

$$\frac{\begin{array}{c}y_j : Term;\ \emptyset \vdash t \longrightarrow t' : Term \\ \Xi \vdash_{macros} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Xi' \\ xy = macro\ x_i(y_j : A) : B = t'\end{array}}{\Xi \vdash_{macros} \{\ macro\ x_i(y_j : A) : B = t;\ \overline{s}\ \} \longrightarrow \{\ xy;\ \overline{s}'\ \};\ \Xi', xy} \quad \text{(M-Macro)}$$

$$\frac{s \text{ is not a macro} \quad \Xi \vdash_{macros} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Xi'}{\Xi \vdash_{macros} \{\ s;\ \overline{s}\ \} \longrightarrow \{\ s;\ \overline{s}'\ \};\ \Xi'} \quad \text{(M-Other)}$$

Figure 3.4: Macro statement typing $\Xi \vdash_{macros} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \}; \Xi'$

$$\frac{\Gamma;\ \Xi \vdash_{\ stats} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Gamma'}{\Gamma;\ \Xi \vdash_{\ stats} \{\ \mathit{macro}\ x_j(y_l : A) : B = t;\ \overline{s}\ \} \longrightarrow \{\ \mathit{macro}\ x_j(y_l : A) : B = t;\ \overline{s}'\ \};\ \Gamma'} \quad \text{(St-Macro)}$$

Figure 3.5: Statement typing extension $\Gamma;\ \Xi \vdash_{\ stats} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Gamma'$

For sake of simplicity we make macros have isolated scope that can only use value of the argument and nothing else from the outer scope.

Lastly we add another typing rule for quasiquotes. A quasiquote is considered to be well formed if all its unquotes are well-formed.

$$\frac{\begin{array}{c}\forall a \in \mathit{unquotes}[\![t]\!].\ \Gamma;\ \Xi \vdash a \longrightarrow a' : \mathit{Term} \\ t' = [\overline{a \mapsto a'}]t\end{array}}{\Gamma;\ \Xi \vdash q"t" \longrightarrow q"t'" : \mathit{Term}} \quad \text{(T-Quote)}$$

Figure 3.6: Term typing extension $\Gamma;\ \Xi \vdash t \longrightarrow t' : T$ (Part 3)

Now that we've got extended typing rules the only thing left to do is to extend existing helper meta-functions (*attribute*) and introduce a few new ones (*reset* and *unquotes*).

$\mathit{attribute}[\![\mathit{macro}\ x(y : A) : B = t]\!] = \mathit{macro}\ x_i(y_j : A) : B = t$
  where $i = \mathit{freshId},\ j = \mathit{freshId}$

Figure 3.7: Extension to *attribute* meta-function

$\mathit{reset}[\![x_i]\!] = x$
$\mathit{reset}[\![(x_i : T) \Rightarrow t]\!] = (x : T) \Rightarrow t$
$\mathit{reset}[\![a\ b]\!] = \mathit{reset}[\![a]\!]\ \mathit{reset}[\![b]\!]$
$\mathit{reset}[\![\{\ \overline{s};\ t\ \}]\!] = \{\ \mathit{reset}[\![\overline{s}]\!];\ \mathit{reset}[\![t]\!]\ \}$
$\mathit{reset}[\![\mathit{new}\ \{\ z_i \Rightarrow \overline{s}\ \}]\!] = \mathit{new}\ \{\ z_i \Rightarrow \mathit{reset}[\![\overline{s}]\!]\ \}$
$\mathit{reset}[\![t.x]\!] = \mathit{reset}[\![t]\!].x$

$\mathit{reset}[\![\emptyset]\!] = \emptyset$
$\mathit{reset}[\![\mathit{val}\ x_i : T = t;\ \overline{s}]\!] = \mathit{val}\ x : T = t;\ \mathit{reset}[\![\overline{s}]\!]$
$\mathit{reset}[\![\mathit{macro}\ x_i(y_j : A) : B = t;\ \overline{s}]\!] = \mathit{macro}\ x(y : A) : B = t;\ \mathit{reset}[\![\overline{s}]\!]$
$\mathit{reset}[\![\mathit{import}\ p.\_;\ \overline{s}]\!] = \mathit{reset}[\![\overline{s}]\!]$

Figure 3.8: *reset* meta-function

$$
\begin{array}{l}
unquotes[\![q\text{"}x\text{"}]\!] = \emptyset \\
unquotes[\![q\text{"}(x : T) \Rightarrow t\text{"}]\!] = unquotes[\![q\text{"}t\text{"}]\!] \\
unquotes[\![q\text{"}a\ b\text{"}]\!] = unquotes[\![q\text{"}a\text{"}]\!] \cup unquotes[\![q\text{"}b\text{"}]\!] \\
unquotes[\![q\text{"}\{\ t\ \}\text{"})]\!] = unquotes[\![q\text{"}t\text{"}]\!] \\
unquotes[\![q\text{"}\{\ val\ x : T = a;\ \overline{s};\ t\ \}\text{"})]\!] = \\
\quad unquotes[\![q\text{"}a\text{"}]\!] \cup unquotes[\![q\text{"}\{\ \overline{s};\ t\ \}\text{"}]\!] \\
unquotes[\![q\text{"}\{\ macro\ x(y : A) : B = b;\ \overline{s};\ t\ \}\text{"})]\!] = \\
\quad unquotes[\![q\text{"}b\text{"}]\!] \cup unquotes[\![q\text{"}\{\ \overline{s};\ t\ \}\text{"}]\!] \\
unquotes[\![q\text{"}\{\ import\ p.\_;\ \overline{s};\ t\ \}\text{"})]\!] = \\
\quad unquotes[\![q\text{"}p\text{"}]\!] \cup unquotes[\![q\text{"}\{\ \overline{s};\ t\ \}\text{"}]\!] \\
unquotes[\![q\text{"}new\ \{\ z \Rightarrow \overline{s}\ \}\text{"}]\!] = unquotes[\![q\text{"}\{\ \overline{s}\ \}\text{"}]\!] \\
unquotes[\![q\text{"}t.x\text{"}]\!] = unquotes[\![q\text{"}t\text{"}]\!] \\
unquotes[\![q\text{"}q\text{"}t\text{""}]\!] = \bigcup_{t_u \in unquotes[\![q\text{"}t\text{"}]\!]} unquotes[\![q\text{"}t_u\text{"}]\!] \\
unquotes[\![q\text{"}\$\{t\}\text{"}]\!] = t
\end{array}
$$

Figure 3.9: *unquotes* meta-function

3.3 Evaluation

We augment previously defined *erase* meta-function to remove all macros because they are a purely compile-time abstraction in our model.

$$erase[\![q\text{"}t\text{"}]\!] = \bigcup_{t_u \in unquotes[\![q\text{"}t\text{"}]\!]} q\text{"}[t_u \mapsto erase[\![t_u]\!]]t\text{"}$$

$$erase[\![macro\ x(y : A) : B = t;\ \overline{s}]\!] = erase[\![\overline{s}]\!]$$

Figure 3.10: Extension to *erase* meta-function

Semantics of quasiquote evaluation is substitution of unquoted terms with their values.

$$\frac{\begin{array}{c}\forall a \in unquotes[\![t]\!].\ a \mid \mu \longrightarrow q\text{"}b\text{"} \mid \mu' \\ t' = [\overline{\$\{a\} \mapsto b}]t\end{array}}{q\text{"}t\text{"} \mid \mu \longrightarrow q\text{"}t'\text{"} \mid \mu'} \qquad \text{(E-QUOTE)}$$

Figure 3.11: Extension to evaluation $t \mid \mu \longrightarrow t' \mid \mu'$

Finally, we define a helper *eval* meta-function which erases and evaluates a term in empty context:

$$\begin{array}{l} eval[\![t]\!] = t'' \\ \quad \text{where } t' = erase[\![t]\!],\ t' \mid \mu \longrightarrow t'' \mid \mu' \end{array}$$

Figure 3.12: *eval* meta-function

3.4 Challenges

3.4.1 Hygiene

As we've seen in previous examples, unhygienic macros have serious issues which are caused by the way how name resolution works with respect to code generated in macros.

Major problems of the current approach come from the fact that identifiers are resolved in macro expansion context completely disregarding the scope where trees where created. For example in previous example:

```
{
  val x: Int = 1
  macro plusx(e: Int): Int = q"$e + x"
  {
    val x: Int = 4
    val y: Int = 5
    plusx y
  }
}
```

One would intuitively expect `x` within quasiquote to bind to the outer `x` definition, not the inner one similarly to how the variable would have been captured in an anonymous function. This intuition corresponds to principle of *referential transparency*.

Another common problem of naïve macro systems is inability to prevent incorrect name space clashes that can happen when scopes of original code and macro-generated code collapse. For example the following code:

```
{
  macro m(e: Int): Int = q"{
    val tmp: Int = 2
    $e
  }"
  val tmp: Int = 3
  m {
    tmp
  }
}
```

Will expand into:

```
{
  macro m_1(e_2: Int): Int = q"{
    val tmp: Int = 2
    $e_2
  }"
  val tmp_3: Int = 3
  {
    val tmp_4: Int = 2
    tmp_4
  }
}
```

Here one can see that expected binding between definition of outer `tmp` and its usage gets captured by definition of the inner val with the same name. This behaviour illustrates lack of *hygiene in the narrow sense* which was originally stated in [6]:

> Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step.

Two of these propetries combined together are what's commonly understood as *hygiene* of the macro system.

3.4.2 Composition of attributed and unattributed trees

Apart from the hygiene issues in Scala we also have problems of our own which are caused by the fact that unlike most macro systems we also expose type information and symbol information to the end user.

So now instead of handling one simple data structure compiler has to be able to sanely handle mix of typed and untyped trees. Additionally it also has to somehow coexist with typechecking optimizations which try to minimize amount of work needed if tree is already typechecked.

A typical example of unfortunate interaction of all of these futures is an assumption that if tree is typed then all its subtrees are typed too. This invariant was quite a reasonable optimization before macros were added to the language but now it can cause really unexpected bugs if macro decides to insert an untyped transformed tree into a typed one causing it to never being typechecked.

To counter this problems a number of helper functions were introduced that lets user to drop type information from the tree if necessary (e.g. resetAllAttrs and resetLocalAttrs). We use an equivalent of resetAllAttrs called *reset* to simplify our unhygienic macro system. It lets us to only deal with untyped trees as a result of macro expansion even though macro may return a mix of typed an untyped trees.

In practice exposing users to those intricacies of underlying compiler design only leads to overly complicated API that requires deep understanding of the internals. In our hygienic system we will strive to make such differences completely transparent and let user not care about them at all.

Chapter 4

Hygienic macros

4.1 Overview

To counter hygiene issues presented in the previous chapter we modify typing architecture by decoupling symbol resolution information (attribution of symbols to definitions and identifiers) from typing environment. Such decoupling was originally pioneered by Kent Dybvig et al [5].

Instead of resolving symbols based on current scoping using current lexical environment Γ, we split information stored in there in two parts:

1. Information needed to resolve symbols goes into lexical context σ, an additional piece of metadata attached to every name in the program. This metadata is going to be propagated down from the root of the tree to leafs during typechecking.

2. Information about symbol meanings (or denotations) is going to be stored separately in Δ and Θ, two disjoint mappings from symbol ids to the information about values and macros correspondingly. They roughly correspond to Γ and Ξ in our previous unhygienic macro system.

More concretely the data structures discussed above have the following structure:

$\Delta ::= \overline{i \mapsto p.\bullet : T}$	Value denotation table
$\Theta ::= \overline{i \mapsto macro(y_j : A) : B = t}$	Macro denotation table
$\sigma ::= \overline{x_i^\sigma \mapsto x_j}$	Lexical context

Figure 4.1: Hygienic syntax extension (Part 1)

Lexical context itself is an ordered sequence of attributions $x_i^\sigma \mapsto x_j$. They encode that identifier x with symbol i and current context σ should get resolved as x_j. Initially before tree is typed both i and σ are set to corresponding empty values and attributions look simply like $x \mapsto x_j$. Which means that unattributed x resolves to j symbol.

Additionally we also need to slightly change name syntax to include lexical context information:

$x,\ y,\ z ::=$	Names:
x	empty symbol and empty context ($x_0^\emptyset$)
x_i	any symbol and empty context ($x_i^\emptyset$)
x_i^σ	any symbol and any context

Figure 4.2: Hygienic syntax extension (Part 2)

Similarly we'll also need to update all previous uses of name from x_i to x_i^σ across other syntax definitions as every name that used to contain symbol would now contain symbol and lexical context. Full syntax definition for hygienic system can be found in A.1 appendix.

4.2 Referential transparency

Lexical context is built incrementally by propagating information down from root to the leaves of the tree during typechecking. Propagation mechanics of the lexical context can be illustrated using a simple program:

```
{
  val x: Int = 1
  (y: Int) => x + y
}
```

When typechecking enters the block scope it would create a fresh symbol for x and propagate attribution down the leafs so both x and y will get $x \mapsto x_1$ in their σ.

Later on when typer reaches function node it would similarly create a new symbol for function argument and propagate attribution down to function body. The context of the identifiers in the body of the function will become $y \mapsto y_2,\ x \mapsto x_1$.

Lastly when identifiers are typechecked they will *resolve* their names using information in the lexical context. x identifier will typecheck as x_1 and y as y_2.

Lets have a look at more complex example that involves macros and quasiquotes.

```
{
  val x: Int = 1
  macro m(e: Int) = q"$e + x"
  {
    val x: Int = 2
    m(x)
  }
}
```

Once we typecheck statements in the outer block we'll get

```
{
  val x₁: Int = 1
  macro m₂(e₃: Int): Int = q"$e₃ + x"
  ...
}
```

Which is quite similar to the end result of unhygienic typechecking with one major difference. x identifier within quasiquote will in fact contain propagated information about its enclosing scope so later on when macro is expanded it will be able to use to correctly resolve it to x_1 but not to the x identifier within the inner block.

Later on typer will reach last expression in the block and will typecheck inner block without yet expanding the macro:

```
  {
    ...
    {
      val x_4: Int = 2
      m(x_4)
    }
  }
```

Here we can see that arguments of the macro get attributed before actual expansion. This means that identifiers in arguments will get non-zero symbols and empty lexical contexts. Unlike in unhygienic system we do not need to *reset* that information.

When macro expands we'll finally get expected result:

```
{
  ...
  {
    val x_4: Int = 2
    x_4 + x_1
  }
}
```

So as you can see name resolution in our system is performed with respect to scope where trees were originally created, not the scope where macro expands. This makes it *referentially transparent.*

4.3 Hygiene in the narrow sense

An important requirement for hygienic system is to be able to avoid name clashes between definitions created by macro and definitions given by the end user. Lets see how our previous example will fare in new system:

```
{
  macro m(e: Int): Int = q"{
    val tmp: Int = 2
    m $e
  }"
  val tmp: Int = 3
  m {
    tmp
  }
}
```

Before expansion all current definitions and identifiers will get corresponding symbols:

```
{
  ...
  val tmp_3: Int = 3
  m_1 {
    tmp_3
  }
}
```

After we evaluate a macro we get a new tree that mixes attributed and unattributed parts:

```
{
  ...
  val tmp_3: Int = 3
  {
    val tmp: Int = 2
    tmp_3
  }
}
```

We start typechecking of the block by creation of fresh symbols to all definitions in there and propagation of those attributions down to leafs of the tree. So inner `tmp` will get attributed with symbol `4`. This information gets propagated to all names in the block in a form of lexical context update $tmp \mapsto tmp_4$.

Then we proceed to typechecking of the leafs of the tree: identifier `tmp`$_3$. According to information in lexical context `tmp`$_3$ stays untouched as attribution $tmp \mapsto tmp_4$ doesn't apply to it.

So finally we get following typechecked result:

```
{
  ...
  val tmp$_3$: Int = 3
  {
    val tmp$_4$: Int = 2
    tmp$_3$
  }
}
```

As you can see the bindings within argument were correctly left as-is and still bind to outer `tmp` val.

What about situations when definitions were themselves located within argument?

```
{
  macro m(e: Int): Int = q"(tmp: Int) => $e"
  m {
    val tmp: Int = 2;
    tmp
  }
}
```

Before macro expands we'll get:

```
{
  ...
  m$_1$ {
    val tmp$_3$: Int = 2;
    tmp$_3$
  }
}
```

Once new tree is returned from a macro:

```
{
  ...
  (tmp: Int) => {
    val tmp$_3$: Int = 2;
    tmp$_3$
  }
}
```

In Scalac situation like this might cause symbol ownership corruption issues due to the fact that the symbol 3 was originally meant to be located within a block, not a function, and when we substitute it into different context the information stored in symbol gets out of touch with actual tree representation and user has to fix it somehow by either resetting local symbols (which usually causes name capture) or use complex and low-level techniques to update existing symbols.

Our new system is able to robustly re-typecheck this code. Whenever we see a definition that already contains a symbol like that tmp$_3$ val we create a new symbol for it despite that fact that it already had one and propagate corresponding information down to the leafs. In this case we might assign symbol 4 to function parameter, and 5 to the val. Lexical context update will look like $tmp_3 \mapsto tmp_5$, $tmp \mapsto tmp_4$. Lastly only identifier in this scope will get resolved to new symbol id tmp_5.

```
{
  ...
  (tmp$_4$: Int) => {
    val tmp$_5$: Int = 2;
    tmp$_5$
  }
}
```

So symbol resolution in our system is not only performed to resolve unattributed identifiers but to also consistently update symbols. This allows our system to uniformly treat attributed and unattributed trees.

4.4 Typechecking

Unlike in the case of unhygienic macro language extension, we need to update most typing rules in the system with new treatment of definitions, symbols and name resolution. In this chapter we'll provide key typing rules and meta-functions that illustrate general structure of the system. Full type system definition can be found in A.2 appendix.

We've already shown the structure an intuition behind lexical context but haven't specified precise behaviour of name resolution and identifier type-checking.

$$\frac{x_j = \mathit{resolve}[\![x_i^\sigma]\!] \quad j \mapsto \bullet : T \in \Delta}{\Delta;\ \Theta \vdash x_i^\sigma \longrightarrow x_j : T} \quad \text{(T-LocalIdent)}$$

$$\frac{x_j = \mathit{resolve}[\![x_i^\sigma]\!] \quad j \mapsto p.\bullet : T \in \Delta}{\Delta;\ \Theta \vdash x_i^\sigma \longrightarrow p.x : T} \quad \text{(T-PrefixedIdent)}$$

Figure 4.3: Extract of hygienic typing $\Delta;\ \Theta \vdash t \longrightarrow t' : T$ (Part 1)

Unlike in previous system we need to perform two steps to typecheck an identifier. First we resolve a name based on lexical context stored within it using *resolve* meta-function. Once we obtain a resolved symbol we look up its denotation in corresponding denotation table and then perform either expansion to fully-attributed path (T-PrefixedIdent) or expansion into identifier with concrete symbol (T-LocalIdent). $\bullet$ sign in the definition of value denotation means that name with given symbol should be substituted instead of $\bullet$ to obtain a typechecked term.

Actual symbol resolution is performed by *resolve* meta-function:

$$\mathit{resolve}[\![x_i^{\emptyset}]\!] = x_i$$
$$\mathit{resolve}[\![x_i^{x_j^{\sigma'} \mapsto x_k, \sigma}]\!] = x_k$$
$$\text{where } \mathit{resolve}[\![x_j^{\sigma'}]\!] \equiv \mathit{resolve}[\![x_i^{\sigma}]\!],$$
$$\mathit{resolve}[\![x_i^{y_j^{\sigma'} \mapsto y_k, \sigma}]\!] = \mathit{resolve}[\![x_i^{\sigma}]\!]$$

Figure 4.4: *resolve* meta-function

resolve deconstructs lexical context from left to right (from latest attributions to earlier ones) looking for a attribution rule that applies to current name. An attribution rule applies if name on the left hand side has the same symbol and name as the current one after recursive attributions. In the end *resolve* returns either name with re-attributed symbol or the same name with the same symbol but empty context if no rule was applicable.

It's important to highlight that once name is resolved the lexical context always becomes empty. This property lets us reason about lexical contexts as purely typechecking-time abstraction that gets erased once all the names has been resolved.

Another part we've not clearly defined is how lexical context is constructed. The main ideas of attributions and propagation can be illustrated on typechecking of anonymous functions:

$$\frac{j = freshId \quad t' = propagate[\![x_i^\sigma \mapsto x_j, t]\!] \qquad \Delta,\ j \mapsto \bullet : A;\ \Theta \vdash t' \longrightarrow t'' : B}{\Delta;\ \Theta \vdash (x_i^\sigma : A) \Rightarrow t \longrightarrow (x_j : A) \Rightarrow t'' : A \Rightarrow B} \quad \text{(T-Func)}$$

Figure 4.5: Extract of hygienic typing $\Delta;\ \Theta \vdash t \longrightarrow t' : T$ (Part 2)

Whenever we typecheck a definition like function parameter we always generate a new symbol for it and propagate corresponding attribution down to the leafs (which in this case means down to the body of the anonymous function.)

All propagated changes also need to have mirrored changes to the corresponding denotation table so that it's possible not just to resolve an identifier but to also typecheck it. Value denotation table Δ contains mappings from symbols to type and (optional prefix.) In this case we add entry that maps freshly generated symbol to the type of the parameter.

Actual propagation of the context is performed by *propagate* helper metafunction. This function walks the tree and prepends new context information to the every name in the tree returning a copy of the tree where all names have been propagated. Its definition can be found on Figure A.7 which is a part of A.2 appendix.

As we've seen earlier Δ gets updated with entries whenever we typecheck a value definition in scope (e.g. function parameters, self definitions, vals) which is really close to how we constructed Γ in unhygienic system. The only difference is that every time we create a symbol we also need to propagate down the leaves of the tree.

Similarly Θ is a representation of macros that is very similar to Ξ in previous system. It also gets filled in with entries during $\vdash_{macros}$ typechecking that happens whenever one typechecks a list of statement (e.g. as part of new or block.)

Entries from Θ are looked up whenever a macro expands:

$$\frac{\begin{array}{c} x_j = resolve[\![x_i^\sigma]\!] \\ j \mapsto macro(y_k : A) : B = b \in \Theta \\ \Gamma;\ \Theta \vdash a \longrightarrow a' : A \\ q"t" = eval[\![[y_k \mapsto q"a'"]b]\!] \\ \Gamma;\ \Theta \vdash t \longrightarrow t' : B \end{array}}{\Gamma;\ \Theta \vdash x_i^\sigma\ a \longrightarrow t' : B} \qquad \text{(T-MApp)}$$

Figure 4.6: Extract of hygienic typing $\Delta;\ \Theta \vdash t \longrightarrow t' : T$ (Part 3)

Interestingly enough this is one of the parts of the typechecking that has become simpler in hygienic settings as we don't need to do extra step that resets attributes. Otherwise it's really close to previous macro application rule with a minor change to two-step name resolution similarly to have identifiers are resolved but only using Θ denotation table.

4.5 Evaluation

Evaluation rules stay exactly the same as in unhygienic extension. Final definition of *erase* meta-function and evaluation of erased terms can be found in A.3 appendix.

Chapter 5

Related work

5.1 Scheme and Racket

This work has been heavily inspired by the lexical context system which was originally presented in "Syntactic Abstraction in Scheme" [5] and is still being used as a foundation for hygienic macro expansion in Racket [3].

Attributions in our model closely correspond to renamings in Scheme. Unlike renamings they do not represent changes of names but rather represent changes of symbols associated with names. Names always stay the same in our model.

Another major difference from Scheme's model is the fact that we do not need marks to prevent clashes between macro-generated code and user-defined code. This is caused by the fact that our macros expand in different order (innermost macro expands first rather than outermost macro expands first as in Scheme) and receive already attributed trees as an input. This difference can be illustrated on a simple example:

```
(x: Int) => {
  macro m(e: Int): Int => Int =
    q"(x: Int) => $e + x"
  m(x)
}
```

By the virtue of `x` argument being pretypechecked by the time when the macro expands it's possible to correctly resolve names without marking:

```
(x₁: Int) => {
  ...
  (x: Int) => x₁ + x
}
```

On the other hand if we were to have macros with untyped arguments we would have run into name capturing problems without marks:

```
(x: Int) => {
  macro m(e): Int => Int = q"(x: Int) => $e + x"
  m(x)
}
```

This macro would expand as:

```
(x₁: Int) => {
  ...
  (x: Int) => x + x
}
```

And without marks there would be no way to tell which of those identifiers should bind to the inner parameter and which should bind to the outer one. Marks solve this problem by guiding name resolution to prefer definitions that were generated in the same transcription step whenever possible.

Another major difference between Scheme and the proposed macro system is the fact that we do not support bindings to definitions at macro expansion site from macro generated code. On one hand this might look like a weakness of our system but on the other hand intentional unhygienic bindings are not practical in our setting due to the fact that macro arguments have to be typed at macro expansion site before the actual macro expansion. This means that typical examples that give rise to the need to escape hygiene (like anaphoric macros) aren't in fact possible in the first place.

In situations when a macro in fact does want to introduce bindings with special meaning into scope one can use dummy methods that throw an exception if they were called outside of a macro and treat them specially inside a macro.

For example the popular async [4] macro uses dummy `await` method that only makes sense within `async` block. `await` calls deliminate boundaries of CPS transform performed by the macro.

```
import scala.async.Async.{async, await}

val future = async {
  val f1 = async { ...; true }
  val f2 = async { ...; 42 }
  if (await(f1)) await(f2) else 0
}
```

This technique closely resembles syntax parameters in Racket [2].

5.2 Haskell

Similarly to our system Template Haskell [9] provides macro-like metaprogramming capabilities for Haskell including quasiquotes functionality. Like in Scala quasiquotes in Haskell have multiple flavors that correspond to different syntactic kinds of trees they represent.

```
cross2 :: Expr -> Expr -> Expr
cross2 f g = [| \ (x,y) -> ($f x, $g y) |]
```

This code is analogous to the following snippet in a system like ours:

```
val cross2: (Term, Term) => Term
  (f: Term, g: Term) => q"(x, y) => ($f(x), $g(y))"
```

Despite syntactic similarities Haskell snippet has significantly different semantics. Unlike our hygienic quasiquotes which persist information about enclosing lexical context and use it to create symbols during typechecking of the tree, Haskell quasiquotes eagerly generate symbols for quoted definitions:

```
cross2 :: Expr -> Expr -> Expr
cross2 f g =
  do { x <- gensym "x"
     ; y <- gensym "y"
     ; ft <- f
     ; gt <- g
     ; return (Lam [Ptup [Pvar x, Pvar y]]
                   (Tup [App ft (Var x), App gt (Var y)]))
     }
```

Here `Expr` is not just a data type that represents syntax trees but in fact an alias to a data type which is enclosed in a `Q` monad. Therefore the desugaring of the quasiquote is a pure monad-based representation of the stateful computation.

```
type Expr = Q Exp
```

In our system quasiquotes can be safely desugared into plain AST construction data types with materialized equivalents of enclosing lexical context. All of the attribution mechanics is performed in the compiler and is not exposed to the end user.

Another problem of eager naming is approach is poor support for decomposing larger quasiquotes into smaller pieces. For example if we were to refactor some piece of quasiquote into helper function:

```
cross2 :: Expr -> Expr -> Expr
cross2 f g = [| \ (x,y) -> ($fxf, $g y) |]
  where fxf = fx f

fx :: Expr -> Expr
fx f = [| $f x |]
```

This would break the binding between an identifier `x` within `fx` and function parameter `x` within `cross2`. To fix this problem one has to resort either to generation of fresh symbols by hand or to usage of unhygienic bindings.

Our system is able to handle such situations automatically thanks to late-binding nature of the name resolution that we propose.

Chapter 6

Conclusions and future work

In this work we've built a fine-grained model of a Scala-like language with a hygienic macro system. Using this model we were able to provide solutions to three major problems of the current unhygienic macro system present in Scala. In our system:

- Identifier bindings are referentially transparent.
- It is impossible to induce inadvertent variable capture.
- Re-typechecking is robust with respect to arbitrary combinations of attributed and unattributed trees.

We strongly believe that these findings can be incorporated in a full-blown Scala compiler to address those issues and make Scala macros hygienic.

Future work in this area might include extensions to the model incorporating more advanced features of Scala like type members, overloading and implicits.

Bibliography

[1] E. Burmako and M. Odersky. Scala macros. Technical report, EPFL, 2012.

[2] R. C. Eli Barzilay. Keeping it Clean with Syntax Parameters. In *Workshop on Scheme and Functional Programming*, 2011.

[3] M. Flatt, R. Culpepper, D. Darais, and R. b. Findler. Macros that work together: Compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program.*, 22(2):181–216, Mar. 2012.

[4] P. Haller and J. Zaugg. SIP-22 - Async, 2013.

[5] R. Hieb, R. K. Dybvig, and C. Bruggeman. Syntactic abstraction in scheme. In *Lisp and Symbolic Computation*, pages 295–326, 1992.

[6] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. Technical report, New York, NY, USA, 1986.

[7] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[8] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for scala. Technical report, EPFL, 2013.

[9] T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002.

Appendices

Appendix A

Hygienic macro calculus

A.1 Syntax

$v ::=$	Values:
$\quad l$	store location
$\quad x_i \Rightarrow t$	function value
$\quad q"t"$	term value provided $unquotes[\![t]\!] \equiv \emptyset$
$i,\ j,\ k ::=$	Symbols:
$\quad 0$	empty symbol
$\quad id$	unique non-zero id
$x,\ y,\ z ::=$	Names:
$\quad x$	empty symbol and empty context $(x_0^{\emptyset})$
$\quad x_i$	any symbol and empty context $(x_i^{\emptyset})$
$\quad x_i^{\sigma}$	any symbol and any context
$a,\ b,\ c,\ t ::=$	Terms:
$\quad x_i^{\sigma}$	identifier
$\quad (x_i^{\sigma} : T) \Rightarrow t$	function
$\quad t\ t$	application
$\quad \{\ \overline{s};\ t\ \}$	block
$\quad new\ \{\ z_i^{\sigma} \Rightarrow \overline{s}\ \}$	new
$\quad t.x$	selection
$\quad q"t"$	term quotation
$\quad \$x_i^{\sigma},\ \$\{t\}$	term unquote
$s ::=$	Statements:
$\quad val\ x_i^{\sigma} : T\ =\ t$	lazy val
$\quad macro\ x_i^{\sigma}(y_i^{\sigma'} : A) : B = t$	blackbox macro
$\quad import\ p.\_$	wildcard import
$p ::=$	Paths:
$\quad x_i^{\sigma}$	identifier
$\quad p.x$	selection
$A,\ B,\ C,\ T ::=$	Types:
$\quad \{\overline{x : T}\}$	structural object type
$\quad T \Rightarrow T$	function type
$\Delta ::= \overline{i \mapsto p.\bullet : T}$	Value denotation table
$\Theta ::= \overline{i \mapsto macro(y_j : A) : B = t}$	Macro denotation table
$\sigma ::= \overline{x_i^{\sigma} \mapsto x_j}$	Lexical context
$\mu ::= \overline{l \mapsto \{\overline{x = t}\}}$	Store

A.2 Typing

$$\frac{\begin{array}{c} x_j = resolve[\![x_i^\sigma]\!] \\ j \mapsto \bullet : T \in \Delta \end{array}}{\Delta;\ \Theta \vdash x_i^\sigma \longrightarrow x_j : T} \qquad (\text{T-}\textsc{LocalIdent})$$

$$\frac{\begin{array}{c} x_j = resolve[\![x_i^\sigma]\!] \\ j \mapsto p.\bullet : T \in \Delta \end{array}}{\Delta;\ \Theta \vdash x_i^\sigma \longrightarrow p.x : T} \qquad (\text{T-}\textsc{PrefixedIdent})$$

$$\frac{\begin{array}{c} j = freshId \quad t' = propagate[\![x_i^\sigma \mapsto x_j, t]\!] \\ \Delta,\ j \mapsto \bullet : A;\ \Theta \vdash t' \longrightarrow t'' : B \end{array}}{\Delta;\ \Theta \vdash (x_i^\sigma : A) \Rightarrow t \longrightarrow (x_j : A) \Rightarrow t'' : A \Rightarrow B} \qquad (\text{T-}\textsc{Func})$$

$$\frac{\Delta;\ \Theta \vdash t \longrightarrow t' : A \Rightarrow B \quad \Delta;\ \Theta \vdash a \longrightarrow a' : A}{\Delta;\ \Theta \vdash t\ a \longrightarrow t'\ a' : B} \qquad (\text{T-}\textsc{App})$$

$$\frac{\begin{array}{c} x_j = resolve[\![x_i^\sigma]\!] \\ j \mapsto macro(y_k : A) : B = b \in \Theta \\ \Gamma;\ \Theta \vdash a \longrightarrow a' : A \\ q"t" = eval[\![[y_k \mapsto q"a'"]b]\!] \\ \Gamma;\ \Theta \vdash t \longrightarrow t' : B \end{array}}{\Gamma;\ \Theta \vdash x_i^\sigma\ a \longrightarrow t' : B} \qquad (\text{T-M}\textsc{App})$$

$$\frac{\Delta;\ \Theta \vdash t \longrightarrow t' : T \quad T <: A}{\Delta;\ \Theta \vdash t \longrightarrow t' : A} \qquad (\text{T-}\textsc{Sub})$$

$$\frac{\begin{array}{c} \forall a \in unquotes[\![t]\!].\ \Gamma;\ \Theta \vdash a \longrightarrow a' : Term \\ t' = [\overline{a \mapsto a'}]t \end{array}}{\Gamma;\ \Theta \vdash q"t" \longrightarrow q"t'" : Term} \qquad (\text{T-}\textsc{Quote})$$

Figure A.1: Hygienic term typing $\Delta;\ \Theta \vdash t \longrightarrow t' : T$ (Part 1)

$$\frac{\begin{array}{c} (\overline{s}', \sigma) = \mathit{attribute}\llbracket \overline{s} \rrbracket \quad \overline{s}'' = \mathit{propagate}\llbracket \sigma,\ s' \rrbracket \\ \overline{xj} = \{\ j \mapsto \bullet : T \mid \mathit{val}\ x_j : T = t \in \overline{s}''\ \} \\ \Theta\ \vdash_{\ \mathit{macros}}\ \{\ \overline{s}''\ \} \longrightarrow \{\ \overline{s}'''\ \};\ \Theta' \\ \Delta,\ \overline{xj};\ \Theta'\ \vdash_{\ \mathit{stats}}\ \{\ \overline{s}'''\ \} \longrightarrow \{\ \overline{s}''''\ \};\ \Delta';\ \sigma' \\ t' = \mathit{propagate}\llbracket \sigma',\ \mathit{propagate}\llbracket \sigma,\ t \rrbracket \rrbracket \\ \Delta';\ \Theta'\ \vdash\ t' \longrightarrow t'' : T \end{array}}{\Delta;\ \Theta\ \vdash\ \{\ \overline{s};\ t\ \} \longrightarrow \{\ \overline{s}'''';\ t''\ \} : T} \quad \text{(T-Block)}$$

$$\frac{\begin{array}{c} j = \mathit{freshId} \quad (\overline{s}', \sigma) = \mathit{attribute}\llbracket \overline{s} \rrbracket \quad \overline{xs} = \mathit{sig}\llbracket s' \rrbracket \\ \overline{s}'' = \mathit{propagate}\llbracket z_i^{\sigma} \mapsto z_j,\ \mathit{propagate}\llbracket \sigma,\ s' \rrbracket \rrbracket \\ \overline{xk} = \{k \mapsto z_j.\bullet : T \mid \mathit{val}\ x_k : T = t \in \overline{s}''\} \\ \Theta\ \vdash_{\ \mathit{macros}}\ \{\ \overline{s}''\ \} \longrightarrow \{\ \overline{s}'''\ \};\ \Theta' \\ \Delta,\ j \mapsto \bullet : \{\overline{xs}\},\ \overline{xk};\ \Theta'\ \vdash_{\ \mathit{stats}}\ \{\ \overline{s}'''\ \} \longrightarrow \{\ \overline{s}''''\ \};\ \Delta';\ \sigma' \end{array}}{\Delta;\ \Theta\ \vdash\ \mathit{new}\ \{\ z_i^{\sigma} \Rightarrow \overline{s}\ \} \longrightarrow \mathit{new}\ \{\ z_j \Rightarrow \overline{s}''''\ \} : \{\overline{xs}\}} \quad \text{(T-New)}$$

Figure A.2: Hygienic term typing $\Delta;\ \Theta\ \vdash\ t \longrightarrow t' : T$ (Part 2)

$$\frac{}{\Delta;\ \Theta\ \vdash_{\ \mathit{stats}}\ \{\} \longrightarrow \{\};\ \Delta;\ \emptyset} \quad \text{(St-Empty)}$$

$$\frac{\Delta;\ \Theta\ \vdash\ t \longrightarrow t' : T \quad \Delta;\ \Theta\ \vdash_{\ \mathit{stats}}\ \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Delta;\ \sigma}{\Delta;\ \Theta\ \vdash_{\ \mathit{stats}}\ \{\ \mathit{val}\ x_i : T = t;\ \overline{s}\ \} \longrightarrow \{\ \mathit{val}\ x_i : T = t';\ \overline{s}'\ \};\ \Delta;\ \sigma} \quad \text{(St-Val)}$$

$$\frac{\begin{array}{c} \Delta;\ \Theta\ \vdash\ p \longrightarrow p' : \{\overline{y : A}\} \\ \sigma = \{\ y \mapsto y_j \mid y \in \overline{y : A},\ j = \mathit{freshId}\ \} \\ \overline{jp} = \{\ j \mapsto p'.\bullet : A \mid y \mapsto y_j \in c,\ y : A \in \overline{y : A}\} \\ \overline{s'} = \mathit{propagate}\llbracket \sigma,\ \overline{s} \rrbracket \\ \Delta,\ \overline{jp};\ \Theta\ \vdash_{\ \mathit{stats}}\ \{\ \overline{s}'\ \} \longrightarrow \{\ \overline{s}''\ \};\ \Delta';\ \sigma' \end{array}}{\Delta;\ \Theta\ \vdash_{\ \mathit{stats}}\ \{\ \mathit{import}\ p.\_;\ \overline{s}\ \} \longrightarrow \{\ \overline{s}''\ \};\ \Delta';\ \sigma', \sigma} \quad \text{(St-Import)}$$

$$\frac{\Delta;\ \Theta\ \vdash_{\ \mathit{stats}}\ \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Delta';\ \sigma'}{\Delta;\ \Theta\ \vdash_{\ \mathit{stats}}\ \{\ \mathit{macro}\ x_j(y_l : A) : B = t;\ \overline{s}\ \} \longrightarrow \{\ \mathit{macro}\ x_j(y_l : A) : B = t;\ \overline{s}'\ \};\ \Delta';\ \sigma'} \quad \text{(St-Macro)}$$

Figure A.3: Hygienic statement typing $\Delta;\ \Theta\ \vdash_{\ \mathit{stats}}\ \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Delta';\ \sigma$

$$\frac{j \mapsto \bullet : Term;\ \emptyset \vdash\ t \longrightarrow t' : Term \qquad \Theta \vdash_{\ macros} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Theta'}{\begin{array}{c}\Theta \vdash_{\ macros} \{\ macro\ x_i(y_j : A) : B = t;\ \overline{s}\ \} \longrightarrow \\ \{\ macro\ x_i(y_j : A) : B = t';\ \overline{s}'\ \};\ \Theta', i \mapsto macro(y_j : A) : B = t'\end{array}} \quad \text{(M-Macro)}$$

$$\frac{s \text{ is not a macro} \qquad \Theta \vdash_{\ macros} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Theta'}{\Theta \vdash_{\ macros} \{\ s;\ \overline{s}\ \} \longrightarrow \{\ s;\ \overline{s}'\ \};\ \Theta'} \quad \text{(M-Other)}$$

Figure A.4: Hygienic macro typing $\Theta \vdash_{\ macros} \{\ \overline{s}\ \} \longrightarrow \{\ \overline{s}'\ \};\ \Theta'$

$$\frac{}{T <: T} \quad \text{(S-Refl)}$$

$$\frac{A <: T \qquad T <: B}{A <: B} \quad \text{(S-Trans)}$$

$$\frac{C <: A \qquad B <: T}{A \Rightarrow B <: C \Rightarrow T} \quad \text{(S-Func)}$$

$$\frac{}{\{\overline{x : A}, \overline{y : B}\} <: \{\overline{x : A}\}} \quad \text{(S-Width)}$$

$$\frac{\forall\ (A,\ B) \in zip[\![\overline{A}, \overline{B}]\!].\ A <: B}{\{\overline{x : A}\} <: \{\overline{x : B}\}} \quad \text{(S-Depth)}$$

$$\frac{\{\overline{x : A}\} \text{ is a permutation of } \{\overline{y : B}\}}{\{\overline{x : A}\} <: \{\overline{y : B}\}} \quad \text{(S-Perm)}$$

Figure A.5: Subtyping

$$sig[\![\overline{s}]\!] = \{\ x : T \mid val\ x_i^c : T = t \in \overline{s}\ \}$$

Figure A.6: *sig* meta-function

$$
\begin{array}{l}
propagate\llbracket\sigma,\ x_k^{\sigma'}\rrbracket = x_k^{\sigma,\sigma'} \\
propagate\llbracket\sigma,\ (x_k^{\sigma'} : T) \Rightarrow t\rrbracket = (x_k^{\sigma,\sigma'} : T) \Rightarrow propagate\llbracket\sigma, t\rrbracket \\
propagate\llbracket\sigma,\ t\ a\rrbracket = propagate\llbracket\sigma, t\rrbracket\ propagate\llbracket\sigma, a\rrbracket \\
propagate\llbracket\sigma,\ \{\ \overline{s};\ t\ \}\rrbracket = \{\ propagate\llbracket\sigma, \overline{s}\rrbracket;\ propagate\llbracket\sigma, t\rrbracket\ \} \\
propagate\llbracket\sigma,\ new\ \{\ z_k^{\sigma'} \Rightarrow \overline{s}\ \}\rrbracket = new\ \{\ z_k^{\sigma,\sigma'} \Rightarrow propagate\llbracket\sigma, \overline{s}\rrbracket\ \} \\
propagate\llbracket\sigma,\ t.x\rrbracket = propagate\llbracket\sigma, t\rrbracket.x \\
propagate\llbracket\sigma,\ q\text{''}t\text{''}\rrbracket = q\text{''}propagate\llbracket\sigma, t\rrbracket\text{''} \\
propagate\llbracket\sigma,\ \$\{t\}\rrbracket = \$\{propagate\llbracket\sigma, t\rrbracket\}
\end{array}
$$

$$
\begin{array}{l}
propagate\llbracket\sigma,\ \emptyset\rrbracket = \emptyset \\
propagate\llbracket\sigma,\ val\ x_k^{\sigma'} : T = t;\ \overline{s}\rrbracket = val\ x_k^{\sigma,\sigma'} = t;\ propagate\llbracket\sigma, \overline{s}\rrbracket \\
propagate\llbracket\sigma,\ macro\ x_i^{\sigma'}(y_j^{\sigma''} : A) : B = t;\ \overline{s}\rrbracket = \\
\quad macro\ x_i^{\sigma,\sigma'}(y_j^{\sigma,\sigma''} : A) : B = propagate\llbracket\sigma, t\rrbracket;\ propagate\llbracket\sigma, \overline{s}\rrbracket \\
propagate\llbracket\sigma,\ import\ p.\_;\ \overline{s}\rrbracket = import\ propagate\llbracket\sigma, p\rrbracket.\_;\ propagate\llbracket\sigma, \overline{s}\rrbracket
\end{array}
$$

Figure A.7: *propagate* meta-function

$$
\begin{array}{l}
resolve\llbracket x_i^{\emptyset}\rrbracket = x_i \\
resolve\llbracket x_i^{x_j^{\sigma'} \mapsto x_k, \sigma}\rrbracket = x_k \\
\quad \text{where } resolve\llbracket x_j^{\sigma'}\rrbracket \equiv resolve\llbracket x_i^{\sigma}\rrbracket, \\
resolve\llbracket x_i^{y_j^{\sigma'} \mapsto y_k, \sigma}\rrbracket = resolve\llbracket x_i^{\sigma}\rrbracket
\end{array}
$$

Figure A.8: *resolve* meta-function

$$
\begin{array}{l}
attribute\llbracket\emptyset\rrbracket = (\emptyset, \emptyset) \\
attribute\llbracket val\ x_i^{\sigma} : T = t;\ \overline{s}\rrbracket = (val\ x_j : T = t;\ \overline{s}', c') \\
\quad \text{where } j = freshId,\ (\overline{s}', c) = attribute\llbracket\overline{s}\rrbracket,\ c' = c, x_i^{\sigma} \mapsto x_j \\
attribute\llbracket import\ p.\_;\ \overline{s}\rrbracket = (import\ p.\_;\ \overline{s}', c) \\
\quad \text{where } (\overline{s}', c) = attribute\llbracket\overline{s}\rrbracket \\
attribute\llbracket macro\ x_i^{\sigma}(y_j^{\sigma'} : A) : B = t; \overline{s}\rrbracket = (macro\ x_{i'}(y_{j'} : A) : B = t'; s', c') \\
\quad \text{where } i' = freshId,\ j' = freshId, \\
\qquad\quad t' = propagate\llbracket y_j^{\sigma'} \mapsto y_{j'}, t\rrbracket, \\
\qquad\quad (s', c) = attribute\llbracket\overline{s}\rrbracket,\ c' = c, x_i^{\sigma} \mapsto x_{i'}
\end{array}
$$

Figure A.9: *attribute* meta-function

$unquotes\llbracket q\text{"}x_i^{\sigma}\text{"}\rrbracket = \emptyset$
$unquotes\llbracket q\text{"}(x_i^{\sigma} : T) \Rightarrow t\text{"}\rrbracket = unquotes\llbracket q\text{"}t\text{"}\rrbracket$
$unquotes\llbracket q\text{"}a\ b\text{"}\rrbracket = unquotes\llbracket q\text{"}a\text{"}\rrbracket \cup unquotes\llbracket q\text{"}b\text{"}\rrbracket$
$unquotes\llbracket q\text{"}\{\ t\ \}\text{"})\rrbracket = unquotes\llbracket q\text{"}t\text{"}\rrbracket$
$unquotes\llbracket q\text{"}\{\ val\ x_i^{\sigma} : T = a;\ \overline{s};\ t\ \}\text{"})\rrbracket =$
$\quad unquotes\llbracket q\text{"}a\text{"}\rrbracket \cup unquotes\llbracket q\text{"}\{\ \overline{s};\ t\ \}\text{"}\rrbracket$
$unquotes\llbracket q\text{"}\{\ macro\ x_i^{\sigma}(y_j^{\sigma'} : A) : B = b;\ \overline{s};\ t\ \}\text{"})\rrbracket =$
$\quad unquotes\llbracket q\text{"}b\text{"}\rrbracket \cup unquotes\llbracket q\text{"}\{\ \overline{s};\ t\ \}\text{"}\rrbracket$
$unquotes\llbracket q\text{"}\{\ import\ p.\_;\ \overline{s};\ t\ \}\text{"})\rrbracket =$
$\quad unquotes\llbracket q\text{"}p\text{"}\rrbracket \cup unquotes\llbracket q\text{"}\{\ \overline{s};\ t\ \}\text{"}\rrbracket$
$unquotes\llbracket q\text{"}new\ \{\ z_i^{\sigma} \Rightarrow \overline{s}\ \}\text{"}\rrbracket = unquotes\llbracket q\text{"}\{\ \overline{s};\ x\ \}\text{"}\rrbracket$
$unquotes\llbracket q\text{"}t.x\text{"}\rrbracket = unquotes\llbracket q\text{"}t\text{"}\rrbracket$
$unquotes\llbracket q\text{"}q\text{"}t\text{""}\rrbracket = \bigcup_{a \in unquotes\llbracket q\text{"}t\text{"}\rrbracket} unquotes\llbracket q\text{"}a\text{"}\rrbracket$
$unquotes\llbracket q\text{"}\$\{t\}\text{"}\rrbracket = t$

Figure A.10: *unquotes* meta-function

A.3 Evaluation

$erase\llbracket x_i^{\sigma}\rrbracket = x_i$
$erase\llbracket (x_i^{\sigma} : T) \Rightarrow t\rrbracket = x_i \Rightarrow t$
$erase\llbracket a\ b\rrbracket = erase\llbracket a\rrbracket\ erase\llbracket b\rrbracket$
$erase\llbracket \{\ \overline{s};\ t\ \}\rrbracket = \{\ erase\llbracket \overline{s}\rrbracket;\ erase\llbracket t\rrbracket\ \}$
$erase\llbracket new\ \{\ z_i^{\sigma} \Rightarrow \overline{s}\ \}\rrbracket = new\ \{\ z_i \Rightarrow erase\llbracket \overline{s}\rrbracket\ \}$
$erase\llbracket t.x\rrbracket = erase\llbracket t\rrbracket.x$
$erase\llbracket q\text{"}t\text{"}\rrbracket = \bigcup_{a \in unquotes\llbracket q\text{"}t\text{"}\rrbracket} q\text{"}[a \mapsto erase\llbracket a\rrbracket]t\text{"}$

$erase\llbracket \emptyset\rrbracket = \emptyset$
$erase\llbracket val\ x_i^{\sigma} : T = t;\ \overline{s}\rrbracket = val\ x_i = t;\ erase\llbracket \overline{s}\rrbracket$
$erase\llbracket import\ p.\_;\ \overline{s}\rrbracket = erase\llbracket \overline{s}\rrbracket$
$erase\llbracket macro\ x_i^{\sigma}(y_j^{\sigma'} : A) : B = t;\ \overline{s}\rrbracket = erase\llbracket \overline{s}\rrbracket$

Figure A.11: *erase* meta-function

$$
\begin{array}{l}
eval[\![t]\!] = t'' \\
\quad \text{where } t' = erase[\![t]\!],\ t' \mid \mu \longrightarrow t'' \mid \mu'
\end{array}
$$

Figure A.12: *eval* meta-function

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t_1'\ t_2 \mid \mu'} \qquad \text{(E-App1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \qquad \text{(E-App2)}$$

$$\frac{}{(x_i \Rightarrow t)\ v \mid \mu \longrightarrow [x_i \mapsto v]t \mid \mu} \qquad \text{(E-AppAbs)}$$

$$\frac{}{\begin{array}{c}\{\ \overline{val\ x_i = a};\ b\ \} \mid \mu \longrightarrow \\ \{\ \overline{val\ x_i = a};\ [\overline{x_i \mapsto a}]b\ \} \mid \mu\end{array}} \qquad \text{(E-Block1)}$$

$$\frac{}{\{\ \overline{val\ x_i = a};\ v\ \} \mid \mu \longrightarrow v \mid \mu} \qquad \text{(E-Block2)}$$

$$\frac{l \notin dom(\mu)}{\begin{array}{c}new\ \{\ z_i \Rightarrow \overline{val\ x_j = a}\ \} \mid \mu \longrightarrow \\ l \mid \mu, l \mapsto \{\overline{x = [z_i \mapsto l]a}\}\end{array}} \qquad \text{(E-New)}$$

$$\frac{t \mid \mu \longrightarrow t' \mid \mu'}{t.x \mid \mu \longrightarrow t'.x \mid \mu'} \qquad \text{(E-Sel1)}$$

$$\frac{\mu(l) = \{x = a, \overline{y = b}\}}{l.x \mid \mu \longrightarrow a \mid \mu} \qquad \text{(E-Sel2)}$$

$$\frac{\begin{array}{c}\forall a \in unquotes[\![t]\!].\ a \mid \mu \longrightarrow q\text{”}b\text{”} \mid \mu' \\ t' = [\$\{a\} \mapsto b]t\end{array}}{q\text{”}t\text{”} \mid \mu \longrightarrow q\text{”}t'\text{”} \mid \mu'} \qquad \text{(E-Quote)}$$

Figure A.13: Evaluation of erased terms $t \mid \mu \longrightarrow t' \mid \mu'$