Explicitly Heterogeneous Metaprogramming with MetaHaskell

Geoffrey Mainland
Microsoft Research
gmainlan@microsoft.com

Abstract

Languages with support for metaprogramming, like MetaOCaml, offer a principled approach to code generation by guaranteeing that well-typed metaprograms produce well-typed programs. However, many problem domains where metaprogramming can fruitfully be applied require generating code in languages like C, CUDA, or assembly. Rather than resorting to add-hoc code generation techniques, these applications should be directly supported by *explicitly heterogeneous* metaprogramming languages.

We present MetaHaskell, an extension of Haskell 98 that provides modular syntactic and type system support for type safe metaprogramming with multiple object languages. Adding a new object language to MetaHaskell requires only minor modifications to the host language to support type-level quantification over object language types and propagation of type equality constraints. We demonstrate the flexibility of our approach through three object languages: a core ML language, a linear variant of the core ML language, and a subset of C. All three languages support metaprogramming with open terms and guarantee that well-typed MetaHaskell programs will only produce closed object terms that are well-typed. The essence of MetaHaskell is captured in a type system for a simplified metalanguage. MetaHaskell, as well as all three object languages, are fully implemented in the `mhc` bytecode compiler.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages

General Terms Languages, Design

Keywords Metaprogramming, open terms, type systems, quasiquotation, linear languages

1. Introduction

Large bodies of widely-used scientific code, such as FFTW3 [14], ATLAS [40], and SPIRAL [29], are built using ad-hoc, custom code generators. Though code generators provide a certain kind of abstraction, allowing one program to express many different, specialized versions of a function, the implementer is typically consigned to a "`printf` purgatory" in which program fragments are represented as arrays of characters and the primary form of composition is string splicing.

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

Languages such as MetaML [37] and MetaOCaml [35] provide metaprogramming environments on the other end of the spectrum—they guarantee that well-typed metaprograms produce well-typed object programs. However, these languages focus on homogeneous metaprogramming, in which the metalanguage and object language are identical. This makes them less suited for applications that must generate code in some other object language. Ideally programmers could write type safe metaprograms and have flexibility in choosing an object language.

We take a step towards the goal of type safe heterogenous metaprogramming with MetaHaskell. We make the following contributions:

- A type system for an idealized metalanguage/object language pair that guarantees that well-typed metaprograms only ever produce closed object language terms that are well-typed. Object terms may contain free variables, and these free variables may be used polymorphically. To our knowledge, no other metaprogramming language allows free variables to be used polymorphically.
- MetaHaskell, a extension of Haskell 98 implemented in the `mhc` compiler that provides modular type system support for metaprogramming with multiple object languages.
- Three object language "plug-ins;" a core ML language, MiniML, a linear variant of the core ML language, Linear MiniML, and C♭. C♭ is a subset of C that is expressive enough to serve as a target for type safe, heterogeneous, run-time code generation which we demonstrate with a compiler for regular expression matchers.
- A methodology for converting a base language and type system into an object language and type system suitable for metaprogramming.

The rest of the paper is organized as follows. We give an overview of MetaHaskell through several small examples in Section 2. In Section 3 we outline our goals for a heterogeneous metaprogramming language. A simplified type system that captures the features of MetaHaskell that are relevant to metaprogramming is described in Section 4 along with the type system of MiniML, a small ML-like language. In Section 5 we show that our framework can accommodate even a substructural object language with little trouble. C♭, a C-like object language, is described in Section 6 in the context of a regular expression compiler that performs run-time code generation. We give some further details of our implementation in Section 7. Section 8 describes related work, and we conclude and describe future work in Section 9. The `mhc` compiler, which implements MetaHaskell and the object languages described in this paper, is publicly available.

2. MetaHaskell Basics

Syntactically, MetaHaskell builds on the quasiquotation feature of GHC [22]. Quasiquotation provides syntactic convenience when working with fragments of abstract syntax—instead of writing an inscrutable mess of constructor applications to build the abstract syntax tree representation of a term, the programmer can write the same term using concrete syntax. Given a quasiquoter, exp, that parses a "core" Haskell language, we can write the classic staged power function as follows.

```
power :: Int → Exp → Exp
power n x
    | n ≡ 0      = ⟦exp|1⟧
    | n ≡ 1      = ⟦exp| $x⟧
    | even n     = square (power (n `div` 2) x)
    | otherwise = ⟦exp| $x * $(power (n − 1) x)⟧
  where
    square :: Exp → Exp
    square x = ⟦exp| $x * $x⟧
```

The power function takes an integer n and an abstract syntax tree x and returns a new abstract syntax tree representing x raised to the power n. The syntax $[\![\mathbf{exp}|\cdot]\!]$ is a quasiquote. At compile time, the argument between the brackets is passed as a string constant to the quasiquoter exp which returns a Haskell expression represented using GHC's internal Haskell AST data type. The effect is just as if the programmer had written the equivalent Haskell term directly, but the syntax is usually much more pleasant. The quasiquoter will often support antiquotation, written here using the syntax $\$(\ldots)$. Internally, exp parses the antiquotation as a Haskell expression and splices the result into the abstract syntax tree in place of the antiquotation. Antiquotation support is purely the responsibility of the quasiquoter, but its inclusion makes the mechanism vastly more useful.

We have used quasiquotation for the purposes of code generation in past work in the context of sensor networks [25] and GPUs [24]. It is also used to support quasiquotation of perl and Ruby-style interpolated strings, regular expressions, parser grammars [15], and JavaScript [3] among other applications. Though quasiquotation provides syntactic convenience and some type safety since it supports terms represented using algebraic data types instead of strings, the terms it produces are still fundamentally untyped. There is nothing to stop the programmer from passing the power function a value of type Exp that represents a *string* expression. A Haskell program containing this call to power will happily type check only to generate an ill-typed core Haskell term at runtime.

We would like to do better. Rather than building a Haskell term of type Exp, our quasiquoter should build a Haskell term with a more accurate type. Ideally, it would build a term with a type that is somehow isomorphic to the type the quoted term has *in the type system of the quoted language*. That is, when we quote a term in core Haskell, it should receive a core Haskell type suitably lifted to Haskell. MetaHaskell allows quasiquoters to reflect object language types into the Haskell type system. We can write the same power function, now with more accurate types.

```
power :: ∀γ.Int → ⟦exp|γ ▷ Int⟧ → ⟦exp|γ ▷ Int⟧
power n x
    | n ≡ 0      = ⟦exp|1⟧
    | n ≡ 1      = ⟦exp| $x⟧
    | even n     = square (power (n `div` 2) x)
    | otherwise = ⟦exp| $x * $(power (n − 1) x)⟧
  where
    square :: ∀γ.⟦exp|γ ▷ Int⟧ → ⟦exp|γ ▷ Int⟧
    square x = ⟦exp| $x * $x⟧
```

Only the type signatures have changed. Although we have written them out explicitly here, these types appear exactly as they would be inferred by MetaHaskell. As well as expressions, object language types can be quoted. The type $[\![\mathbf{exp}|\gamma \triangleright \mathsf{Int}]\!]$ is the type of code that, in any type environment γ, has type Int. That is, object terms can be polymorphic in their typing environment. Note that we quantify over γ at the *metalanguage* level. We discuss this further in Section 4.

MetaHaskell quasiquoters provide strong typing of object language terms by plugging in to the metalanguage's type system. In all object languages we describe, object language types are refinements of Haskell types. For example, in the case of power, we refined the Haskell type Exp of abstract syntax trees. The run-time representation of a quasiquoted object term is just the corresponding abstract syntax term with an unrefined type. This means that, at run-time, we can convert an object language term to its abstract syntax representation via a safe erase function defined in terms of unsafeCoerce.

```
erase :: ∀γ, α.⟦exp|γ ▷ α⟧ → Exp
erase = unsafeCoerce
```

2.1 Safe run-time code generation

Strongly typed quasiquotation guarantees that only well-typed object language terms will be constructed, at least up until the point of erasure, but we can use it for more than just generating abstract syntax. Consider again our power function, but imagine instead that we wish to generate a specialized version of the power function at run-time as MetaOCaml would allow. MetaHaskell includes support for the C♭ object language, a restricted form of C, as well as a quasiquoter for C♭ functions that instead of producing a value whose run-time representation is an abstract syntax tree, produces a pointer to an actual compiled function. A code-generating version of the power function, cpower, can be written as follows. As with power, the type signatures for both go and cpower are not necessary as they would be inferred by MetaHaskell as shown.

```
cpower :: Int → FunPtr (CDouble → IO CDouble)
cpower n
    ⟦cfun|double pown (double x)
          {double r;
            $stms:(reverse (go n))
            return r;
          }⟧
    )
  where
    go :: ∀γ.Int → [⟦cstm|{double x; double r} γ ▷ void⟧]
    go n | n ≡ 0      = [⟦cstm|r = 1.0⟧]
         | n ≡ 1      = [⟦cstm|r = x;⟧]
         | even n     = ⟦cstm|r = r * r;⟧:go (n `div` 2)
         | otherwise = ⟦cstm|r*= x;⟧   :go (n − 1)
```

We note several aspects of MetaHaskell that are newly illustrated by cpower. First of all, $[\![\mathbf{cfun}|\cdot]\!]$ takes a quoted function and returns a FunPtr—the type of foreign function pointers in Haskell's foreign function interface—indexed by the Haskell translation of the quoted function's type. That is, because the quoted function `pown` has the C♭ type `double (*)(double)`, the index to FunPtr is its translation, CDouble → IO CDouble. This is in contrast to power, which returned a value with an object language type. Because $[\![\mathbf{cfun}|\cdot]\!]$ performs run-time code generation and therefore requires the quoted function to be closed, attempting to quote an open C♭ function using $[\![\mathbf{cfun}|\cdot]\!]$ results in a compile-time error.

Also in contrast to the core Haskell quasiquoter, C♭ provides quasiquoters for *multiple* syntactic categories. We use the $[\![\mathbf{cstm}|\cdot]\!]$ quasiquoter to build up the body of the `pown` function via recursive calls to go. The $[\![\mathbf{cstm}|\cdot]\!]$ quasiquoter *does* return a value with an

object language type—the type of the quoted C♭ statement. The type of go reflects the fact that it returns a list of statements, each of which is valid in a type environment where the variables x and r have type double. The implementation of cpower uses C♭'s ability to antiquote lists of statements to build the body of pown.

2.2 Why not GADT's?

This seems like a lot of trouble to go to when GADTs [41] suffice to write the power function. A quasiquoter has three basic jobs. First, it provides concrete syntax for an object language. Second, it embodies a decision procedure that provides a type—which may be an object language type—for the quoted term (and its antiquotations). Third, it must reflect the type provided by the decision procedure into the host language.

The first job is no different from that performed by quasiquoters as implemented in GHC—it is a purely syntactic task. We argue that the second job—performing type inference on object terms—is useful even when object language types have a strong encoding in the metalanguage's type system, i.e., when the encoding is injective. For example, although GADTs can encode open lambda terms [1], dealing with the structural rules required to work with such terms involves a great deal of bookkeeping—exactly the sort of bookkeeping that a type inference procedure automates. Furthermore, it is not clear how often an object language's type system has a strong encoding in Haskell with GADTs. Consider the following MetaHaskell term.

$$\mathsf{polyopen} = [\![\textbf{exp}|(\mathsf{f}\ 1, \mathsf{f}\ \mathsf{True})]\!]$$

This MetaHaskell term, which uses the free variable f polymorphically, has the following type.

$$\forall \alpha, \beta, \rho, \gamma. [\![\textbf{exp}|\{\mathsf{f} :: [\mathsf{Int} \to \alpha; \mathsf{Bool} \to \beta]\,\rho\}\gamma \rhd (\alpha, \beta)]\!]$$

This type states that the quoted term is valid in any type environment that contains at least a binding for f, that the binding for f must have at least *both* the types $\mathsf{Int} \to \alpha$ and $\mathsf{Bool} \to \beta$, and that in such an environment the quoted expression has type (α, β). We give further details in Section 4, but suffice to say that inference is tricky and the appropriate GADT encoding for such a type is not immediately obvious. However, even if our host language were a language like Coq, which in this case would ensure a strong embedding of the object language type system, we would still like to have access to the decision procedure embodied in the object language quasiquoter and a framework for object language integration.

3. Design Goals

Before proceeding, we outline our design goals for a heterogeneous metaprogramming languages. In what follows we primarily use small, idealized object and metalanguages. Rather than utilizing the concrete MetaHaskell syntax seen in Section 2, we use MetaML-like syntax for quotation and antiquotation. For example, in the following expression, the body of the lambda is a quotation containing the antiquoted term $f\ 1$.

$$\lambda f \to \langle\!\langle (\mathtt{g}\ 1, \tilde{}(f\ 1)) \rangle\!\rangle$$

- **Syntactic support:** The programmer should be able to write object terms using the object language's syntax.
- **Antiquotation:** The metalanguage should allow abstraction over object language sub-terms. That is, object terms should be able to contain antiquoted sub-terms.
- **Heterogeneity:** The metalanguage should provide support for multiple object languages in the same overall framework. Ideally, support for new object languages can be added to the metalanguage in a modular way that require little or no modification of the metalanguage implementation.
- **Type soundness:** Well-typed metaprograms should only generate well-typed object terms.
- **Type inference:** The metalanguage should be able to infer the types of object terms without excessive programmer burden. Annotations on the level of those that Haskell requires for, e.g., impredicative instantiation, are reasonable. If the object language's type checker can infer the type of an object term, then the metalanguage's type inference engine should be able to infer the type of the same object term when it appears in a metaprogram.
- **Open object terms:** The metalanguage should permit object terms with free variables. Ideally, inference for open object terms would not require additional programmer annotations. This would allow us to quote object language fragments like the C expression $\langle\!\langle \mathsf{sin}\ (\mathsf{x}) \rangle\!\rangle$, where both sin and x are free.
- **Subterm typability:** If an object term is well-typed, then any of its sub-terms, when appearing in isolation, should also be well-typed.
- **Subterm abstractability:** We want the property that, at the metalanguage level, we can perform β-abstraction over object language sub-terms. That is, we want to be able to abstract over any sub-term of an object term, apply the abstraction to the abstracted sub-term, and have the new application be well-typed. Obviously we want preservation to hold, so if this application is well-typed, then it has the same type as the original term, which we would recover via β-reduction. For example, we want it to be the case that the metalanguage term

  $$\langle\!\langle \underline{\mathtt{let}}\ \mathtt{f} = \lambda \mathtt{x} \to \mathtt{x}\ \underline{\mathtt{in}}\ (\mathtt{f}\ 1, \mathtt{f}\ \underline{\mathtt{true}}) \rangle\!\rangle$$

  is equivalent to the metalanguage term

  $$(\lambda e \to \langle\!\langle \underline{\mathtt{let}}\ \mathtt{f} = \lambda \mathtt{x} \to \mathtt{x}\ \underline{\mathtt{in}}\ \tilde{}e \rangle\!\rangle) \langle\!\langle (\mathtt{f}\ 1, \mathtt{f}\ \underline{\mathtt{true}}) \rangle\!\rangle$$

  This is slightly different from *subterm typability* which only requires that any object language sub-term be typeable in isolation, not that an object language term be β-abstractable. In particular, there is difficulty with abstracting over a sub-term that appears in the right-hand side of a recursive let binding which we discuss in Section 4.1.
- **Fresh name generation:** The metalanguage should provide facilities for generating fresh names for use in object terms. That is, we want to be able to gensym names that are guaranteed not to occur in any object language term so we can avoid unintended variable capture when generating code.
- **Hygiene:** The programmer should have the ability to require that free variables appearing in an antiquote be used hygienically. For example, consider the term:

  $$\langle\!\langle \underline{\mathtt{let}}\ \mathtt{x} = \ldots\ \underline{\mathtt{in}}\ \tilde{}(f\ \langle\!\langle \mathtt{x} \rangle\!\rangle) \rangle\!\rangle$$

  Here x appears free in the argument to f, which appears in an antiquotation and is some metalanguage function that manipulates object language fragments. We would like the programmer to be able to reason about this fragment without knowing the implementation details of f. In particular, the programmer should not have to know what object language context f might place its argument in in the process of building a new object language term—even if that context might bind a variable named x. We want hygiene—the x appearing in $\langle\!\langle \mathtt{x} \rangle\!\rangle$ should always refer to the x bound by the top-level let, not to any x that might be bound in the term constructed by f in which $\langle\!\langle \mathtt{x} \rangle\!\rangle$ may find itself.

- **Object term elimination:** The programmer should be able to eliminate as well as introduce object language terms. That is, we want to be able to use the metalanguage to perform intensional analysis of object terms while retaining full object term typing.

MetaHaskell make substantial progress in providing these desirable features: it provides syntactic support, antiquotation, heterogeneity, type soundness, inference, open terms, and subterm typability. We have not addressed hygiene, fresh name generation, or object term elimination. Ideas for making further progress are outlined in Section Section 9.

4. A Type System for Heterogeneous Metaprogramming

MetaHaskell consists of Haskell 98 plus extensions for heterogeneous metaprogramming—it is a rather large language. As such, in this section we describe the type system not of full MetaHaskell, but of MiniMeta, a simplified version of MetaHaskell that nonetheless contains all the type system features essential to MetaHaskell's support for metapgrogramming. Jointly, we present the type system for MiniML, an ML-like object language.

4.1 MiniML: Object Language Essentials

Two of our object language design goals are to support open terms and antiquotation, e.g., object language terms in which we have abstracted over a sub-term, as in the term $\lambda x \to \langle\!\langle \tilde{} x + 1 \rangle\!\rangle$. Both requirements present difficulties.

The presence of free variables means that we no longer face a type inference problem, but a *typing* inference problem. The difference between the two is that given a term, *typing* inference must produce a typing context as well as a type, whereas type inference need only produce a type. Although ML does have the principal type property, it *does not* have principal typings. We can see this by considering the (open) object language term $\mathtt{x}\ \mathtt{x}$ [1]. We could give this term one of the following two typings

$$\{x : \forall a.a\} \vdash \mathtt{x}\ \mathtt{x} : \forall a.a$$
$$\{x : \forall a.a \to a\} \vdash \mathtt{x}\ \mathtt{x} : \forall a.a \to a$$

The former derivation provides more because the term has type $\forall a.a$, but it requires more than the latter derivation because it can only occur in a typing environment where x has type $\forall a.a$. Neither derivation is more general than the other.

Shao and Appel [33] partly addressed the issue of typing open terms in solving the *smartest compilation* problem. They present an algorithm to infer the minimal import interface required by a compilation unit. In essence they are inferring a minimal context, i.e., a typing. Jim [16] connects smartest compilation to the typing problem and gives a more explicit presentation of a typing inference algorithm. ML's lack of principal typings will not hinder us because we will allow contexts to include more types than can appear in the expression language. The algorithm presented by Shao and Appel [33] collects constraints on variable instantiations and then "matches" them against the poltype inferred at the variable's binding site. The algorithm presented by Jim [16] uses intersection types to represent what are morally the same constraints, a technique we will reuse for MiniML.

Open terms present difficulties, but at least it is syntactically apparent which variables are free in an open term. An object language term containing an antiquotation is not so well-behaved. After all, what are the free variables in the term $\langle\!\langle \tilde{} x + 1 \rangle\!\rangle$? It is apparent that we must somehow express joint constraints on the context of a quotation and the contexts of its constituent antiquotations.

[1] This example is taken from Jim [16].

Term variables		$\in$	$\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{f}, \mathtt{g}, \mathtt{h}$
Type variables		$\in$	$\mathtt{a}, \mathtt{b}, \mathtt{c}$
Extension variables	ρ		
Row variables	γ		
Expressions	$\mathtt{e}$	::=	$\mathtt{i} \mid \underline{\mathtt{true}} \mid \underline{\mathtt{false}} \mid \mathtt{x} \mid \bullet_i \mid \lambda \mathtt{x}.\mathtt{e} \mid \mathtt{e}_1\ \mathtt{e}_2 \mid \underline{\mathtt{let}}\ \mathtt{x} = \mathtt{e}_1\ \underline{\mathtt{in}}\ \mathtt{e}_2$
Types	τ	::=	$\underline{\mathtt{int}} \mid \underline{\mathtt{bool}} \mid \tau_1 \to \tau_2 \mid \mathtt{a}$
Type schemes	σ	::=	$\forall \overline{\mathtt{a}}.\tau$
Type environment schemes	ξ	::=	$\bigwedge [\tau_i] \xi \mid \sigma \mid \rho$
Type environments	Γ	::=	$\{\mathtt{x_i} : \xi_i\}\,\Gamma \mid \bot \mid \gamma_{\#\overline{\mathtt{x}}}$
Context environments	Δ	::=	$\{\bullet_i : \Gamma_i \rhd \sigma_i\}$

Figure 1: Object language syntax

The syntax of MiniML is given in Figure 1. We use a typewriter font to distinguish MiniML terms from metalanguage terms. Its differences with respect to a standard ML-like language are highlighted. The only change in the syntax of expressions is the presence of *contexts* for antiquotations, $\bullet_i$. Although our concrete syntax inlines antiquotations into quotations, antiquotation is really an abstraction/application pair. For example, the quasiquote $\lambda x \to \langle\!\langle \tilde{} x + 1 \rangle\!\rangle$ desugars into $\lambda x \to \langle\!\langle \bullet_1 + 1 \rangle\!\rangle\ x$. A syntactically valid MiniML term will only contain sequentially and distinctly numbered contexts, although there is no constraint on which permutation of context numbering is chosen—in practice the parser numbers the contexts sequentially in parse order.

We represent contexts using Rémy's extensible records [30, 31]. This is a natural way to express the joint constraints between quasiquotation contexts and their antiquotations. Extensible records use *row variables* to represents "the rest" of the fields in a record, allowing expressions to be polymorphic in the records they manipulate—only the fields that are accessed are required to be present, but the record may also contain additional unreferenced fields that are represented by the row variable. Similarly, row variables allow our object language terms to be polymorphic in "the rest of" their context. This form of polymorphism is permissible because our object language allows weakening. MiniML's type environments, Γ, use Rémy's record type, including row variables with lacks constraints of the form $\gamma_{\#\overline{\mathtt{x}}}$. A lacks constraint specifies which variables (record labels) may not appear in the record extension associated with the row variable that it annotates.

Type environments do not bind variables to type schemes, but to *type environment schemes*. A type environment scheme is either a polytype or an *extensible* intersection type. Extensible intersection types type free variables in open terms. For example, the quotation

$$\langle\!\langle (\mathtt{f}\ 1, \mathtt{f}\ \underline{\mathtt{true}}) \rangle\!\rangle$$

has the type

$$\forall \mathtt{a}, \mathtt{b}, \rho, \gamma. \boxed{\{\mathtt{f} : \bigwedge [\underline{\mathtt{int}} \to \mathtt{a}; \underline{\mathtt{bool}} \to \mathtt{b}]\,\rho\}\,\gamma \rhd (\mathtt{a}, \mathtt{b})}$$

Note that quoted terms are metalanguage terms, so this type is a metalanguage type. In this type we have quantified over the object language typing variables $\mathtt{a}$, $\mathtt{b}$, ρ, and γ, but the quantification is

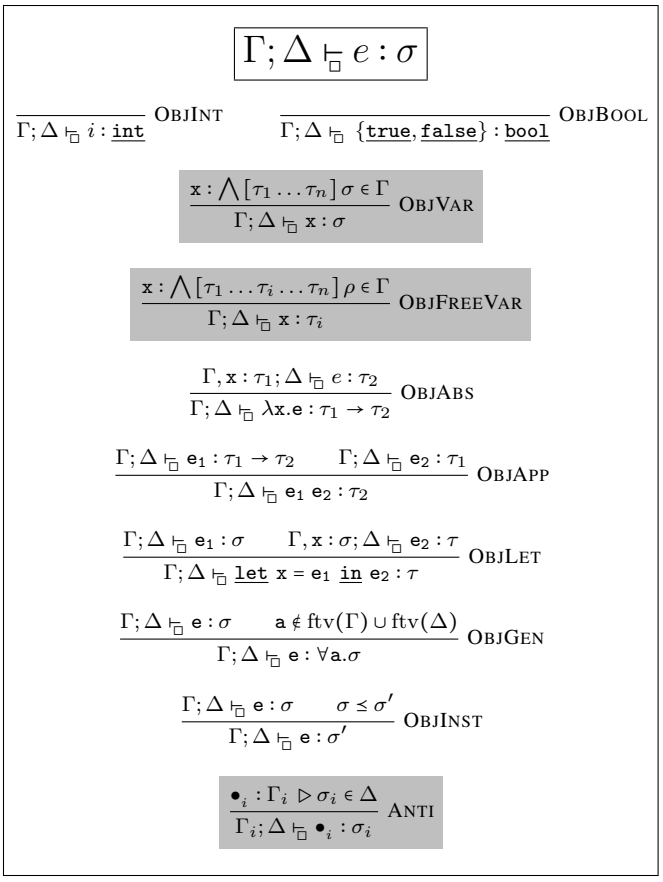

Figure 2: Declarative typing rules for MiniML

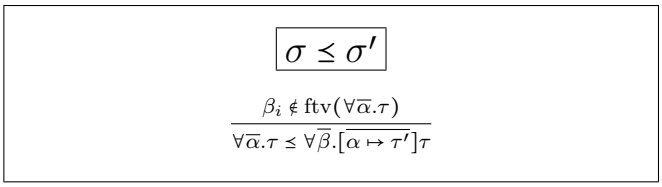

Figure 3: MiniML type subsumption relation

done at the metalanguage level. The type of the object language term is enclosed in a box when it appears in a metalanguage type to distinguish it as an object language type. We further discuss these details in Section 4.2, but for now the germane aspect of this type is the extensible intersection type assigned to `f` in the typing environment, $\bigwedge[\underline{\texttt{int}} \to \texttt{a}; \underline{\texttt{bool}} \to \texttt{b}]\,\rho$. This type says that `f` must have both type $\underline{\texttt{int}} \to \texttt{a}$ and type $\underline{\texttt{bool}} \to \texttt{b}$, but that it may also have additional types in the typing environment, indicated by the extension variable ρ. Extension variables serve the same function in extensible intersection types as row variables serve for extensible records.

Context environments, Δ, provide typings for contexts, of the form $\Gamma \rhd \sigma$, that specify both an environment Γ and a polytype σ. The declarative type system for MiniML is given in Figure 2 along with the corresponding subsumption relation in Figure 3; we again highlight the differences with respect to the standard declarative rules for an ML-like language. The rule for typing contexts, Anti, requires that the environment component Γ_i of a context's typing in the context environment Δ must exactly match the Γ_i in the derivation where the context $\bullet_i$ occurs. This directly implies that any bound variable that scopes over $\bullet_i$, as well as any variable that occurs free *anywhere* in the quotation in which $\bullet_i$ appears, must be present in Γ_i. These free variables may be used at additional types in the expression "plugged in" to the context due to the extension variable ρ, and this expression may also use additional free variables that do not appear explicitly in Γ due to the row variable γ. As we explain in Section 4.2, the type variables ρ and γ are instantiated at the metalanguage level whenever a quotation is applied to an antiquotation.

Type environment schemes have the form $\bigwedge[\tau_i]\,\sigma$, where σ is a polytype, or $\bigwedge[\tau_i]\,\rho$, where ρ is an extension variable, and in both cases the intersection may be empty. Though the polytype form of type environment schemes is not strictly necessary for the declarative rules, it is a technical device for the benefit of the algorithmic rules, which we do not present here, allowing them to substitute a polytype σ for an extension variable ρ in the typing rules for binders. A type environment scheme $\bigwedge[\tau_i]\,\sigma$ is therefore only well-formed when σ can be instantiated to each of the τ_i in the intersection. A variable `x` with a type environment scheme $\bigwedge[\tau_1 \ldots \tau_n]\,\sigma$ must therefore have type σ according to ObjVar. A variable that occurs free is expected to have a type environment scheme $\bigwedge[\tau_1 \ldots \tau_n]\,\rho$ and may have any type τ_i, $1 \le i \le n$, according to ObjFreeVar.

There is a subtlety with generalization: the rule ObjGen must look for free type variables in both Γ and Δ. Consider the following term

$$\lambda x \to \langle\!\langle \underline{\texttt{let}}\ \texttt{y} = \lambda\texttt{z} \to \texttt{z}\ \tilde{}x\ \underline{\texttt{in}}\ (\texttt{y}\ (\lambda\texttt{u} \to 1), \texttt{y}\ (\lambda\texttt{v} \to \underline{\texttt{true}})) \rangle\!\rangle$$

Because `z` will appear in the Δ used to type the context $\bullet_1$ (where $\tilde{}x$ occurs), we cannot generalize `y`, so this term cannot be typed. To see why this must be the case, imagine applying this lambda to the expression $\langle\!\langle \texttt{z}\ 2 \rangle\!\rangle$

In general, the presence of an antiquotation in the right-hand-side of a binding prevents generalization. This means that our extended language does not satisfy the *subterm abstractability* goal defined in Section 3. If we also added type ascription, then a type signature would suffice in this case to assign `y` a polymorphic type.

4.2 MiniMeta: Object Language Type System Integration

MiniMeta integrates support for using MiniML as an object language. Though we present it here in a setting where MiniML is the only supported object language, we will shortly point out the type system features that are necessary to support MiniML and show that they are general enough to also support a variety of other object languages. In Section 4.4 we give a more detailed qualitative description of the process for transforming a base language's syntax and type system into a suitable object language syntax and type system.

The syntax of MiniMeta is given in Figure 4. Quotations, of the form $\langle\!\langle \texttt{e} \rangle\!\rangle$, are the only non-standard expression syntax. We have also already seen object language typings, of the form $\boxed{\Gamma \rhd \sigma}$. The third novel aspect of MiniMeta is that it allows quantification over types by object language type variables (`a`), object language row variables (γ), and object language extension variables (ρ).

Quantification must be done at the metalanguage level if there is any hope of connecting typing constraints on quotations to constraints on their constituent antiquotations. Consider the term

$$\lambda x \to \langle\!\langle (\texttt{f}\ \texttt{1}, \tilde{}x) \rangle\!\rangle$$

The metalanguage binds x, which must have an object language typing (we will show the typing rules shortly). There are no constraints on the type of x, so we will simply use the type variable

Term variables		$\in$	x, y, z, f, g, h
Type variables		$\in$	a, b, c
Quantification variables	ω	::=	$a \mid \mathtt{a} \mid \rho \mid \gamma$
Expressions	e	::=	$i \mid \mathsf{true} \mid \mathsf{false} \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \langle\!\langle \mathtt{e} \rangle\!\rangle$
Types	ν	::=	$\mathsf{int} \mid \mathsf{bool} \mid \nu_1 \to \nu_2 \mid a \mid \boxed{\Gamma \rhd \sigma}$
Type schemes	φ	::=	$\forall \overline{\omega}.\nu$
Type environments	Θ	::=	$\{x_i : \varphi_i\}$

Figure 4: MiniMeta syntax

$$\boxed{\Theta \vdash e : \varphi}$$

$$\frac{}{\Theta \vdash i : \mathsf{int}}\ \textsc{Int} \qquad \frac{}{\Theta \vdash \{\mathsf{true}, \mathsf{false}\} : \mathsf{bool}}\ \textsc{Bool}$$

$$\frac{x : \varphi \in \Theta}{\Theta \vdash x : \varphi}\ \textsc{Var}$$

$$\frac{\Theta, x : \nu_1 \vdash e : \nu_2}{\Theta \vdash \lambda x.e : \nu_1 \to \nu_2}\ \textsc{Abs} \qquad \frac{\Theta \vdash e_1 : \nu_1 \to \nu_2 \quad \Theta \vdash e_2 : \nu_1}{\Theta \vdash e_1\ e_2 : \nu_2}\ \textsc{App}$$

$$\frac{\Theta \vdash e_1 : \varphi \quad \Theta, x : \varphi \vdash e_2 : \nu}{\Theta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \nu}\ \textsc{Let}$$

$$\frac{\Theta \vdash e : \varphi \qquad \omega \notin \mathrm{ftv}(\Theta)}{\Theta \vdash e : \forall \omega.\varphi}\ \textsc{Gen} \qquad \frac{\Theta \vdash e : \varphi \qquad \varphi \leq \varphi'}{\Theta \vdash e : \varphi'}\ \textsc{Inst}$$

$$\frac{\Gamma; \{\bullet_i : \Gamma_i \rhd \sigma_i\} \vdash_{\square} \mathtt{e} : \sigma}{\Theta \vdash \langle\!\langle \mathtt{e} \rangle\!\rangle : \overline{\boxed{\Gamma_i \rhd \sigma_i}} \to \boxed{\Gamma \rhd \sigma}}\ \textsc{Quote}$$

Figure 5: Declarative typing rules for MiniMeta

$\mathtt{b}$ to represent this type. However, we do know that the typing environment of x's object language typing must include $\mathtt{f}$ at least at the type $\underline{\mathtt{int}} \to \mathtt{a}$ for some $\mathtt{a}$. Furthermore, x may also use free variables other than $\mathtt{f}$, and in fact may use $\mathtt{f}$ at other types, but whatever these uses may be, they will *also* occur in the quotation $\langle\!\langle (\mathtt{f}\ \mathtt{1}, \tilde{}x) \rangle\!\rangle$ where x occurs as a sub-expression. We use type variables ρ and γ to represent these shared uses of the environment. The full expression therefore has the type

$$\forall \mathtt{a}, \mathtt{b}, \rho, \gamma. \boxed{\{\mathtt{f} : \bigwedge [\underline{\mathtt{int}} \to \mathtt{a}]\, \rho\}\, \gamma \rhd \mathtt{b}} \to \boxed{\{\mathtt{f} : \bigwedge [\underline{\mathtt{int}} \to \mathtt{a}]\, \rho\}\, \gamma \rhd (\mathtt{a}, \mathtt{b})}$$

MiniMeta's typing rules appear in Figure 5. Although we have not explicitly kinded the four varieties of type variables in our presentation, morally they do have different kinds as shown by the subsumption relation given in Figure 6.

Object language types are just types in our metalanguage. A term with an object language type, e.g., a quasiquotation, can appear in a program, and an object language type can be used to in-

$$\boxed{\varphi \leq \varphi'}$$

$$\frac{b_i, \mathtt{b}_j, \rho_k, \gamma_l \notin \mathrm{ftv}(\forall \overline{\omega}.\nu)}{\forall \overline{\omega}.\nu \leq \forall \overline{b}\ \overline{\mathtt{b}}\ \overline{\rho}\ \overline{\gamma}.[\overline{a \mapsto \nu'}, \overline{\mathtt{a} \mapsto \nu''}, \overline{\rho \mapsto \xi}, \overline{\gamma \mapsto \Theta}]\nu}$$

Figure 6: MiniMeta type subsumption relation

stantiate a type variable a. However, object language type variables that appear in metalanguage types, name $\mathtt{a}$, ρ, and γ, serve only as *evidence*—no values will ever have a type with an object language kind.

Quotations are typed with the QUOTE rule. This rules makes use of the object language judgment to type the quoted object language term and results in an arrow type with as many arguments as there are contexts in the quotation. Recall that the typing environment for contexts must "match up" exactly with the typing environment in the derivation where the context occurs. This means that for the abstraction introduced by a quotation to be applied, the i^{th} applicand—an antiquotation—must have precisely the type $\boxed{\Gamma_i \rhd \sigma_i}$. This may be unsettling because it seems to place too many restrictions on Γ_i. However, it is only unsettling because we are not used to encountering polymorphic environments. Just as a classic polymorphic function like cons allows us to build a list at many types, row and extension variables allow us to use a typing polymorphically and therefore use an object term *at many typings*. We use an object term at many typings the same way we would use cons at many types—by instantiating its type variables within the metalanguage. That is, metalanguage instantiation lets us "match up" the Γ's in object language types.

4.3 Hygiene

Although we listed hygiene as one of our goals in Section 3, MiniML and MiniMeta are unhygienic, a rather undesirable feature. Kim et al. [20] allow programmer control over hygiene by introducing hygienic lambda abstraction, $\lambda^{\star}$, that performs capture-avoiding substitution on the abstraction before it is applied. We believe this extension could be incorporated easily into our system: instead of elaborating to abstract syntax trees, quasiquoted expressions would elaborate to computations in a name-generating monad. Assuming that computations in our name-generating monad were run via runQ, the erase function would then be defined as

```
erase :: ∀γ, α.⟦exp|γ ▷ α⟧ → Exp
erase = runQ ∘ unsafeCoerce
```

We note that our desire for subterm abstractability is incompatible with hygiene. However, providing both hygienic and unhygienic binding forms also allows the programmer to choose between subterm abstractability and hygiene on a case-by-case basis.

4.4 Crafting an Object Language

Our metalanguage and object language seem to be tied together intimately. This raises two related questions: how easily can a new object language be integrated into MiniMeta, and how does one transform a base language—a language, like core ML, without support for antiquotation and without a typing inference procedure—into an object language suitable for integration.

There are four points of integration between MiniMeta and MiniML.

1. Syntactic support for quoting object language terms and object language types.

2. Type system support for object language types appearing as terms in metalanguage types.
3. Quantification over object language type variables in metalanguage types.
4. Passing type equality constraints from the metalanguage's type checker to the object language.

Importantly, MiniMeta knows nothing about the term structure of either MiniML types or MiniML terms—object language expressions and object language types are both completely opaque to the metalanguage. Syntactic support for quasiquotation is straightforward, as the object language parser need only produce an object language term and a list of antiquotations which the metalanguage parser then handles. The object language must provide hooks for working with object language type variables, but this can be done in a generic way. Similarly, the metalanguage type checker can simply pass equality constraints between object language types to the object language type checker without requiring any knowledge of the structure of these types. We claim that the metalanguage is largely agnostic with respect to the form of the object language and its types and that integrating new object languages requires few changes to the metalanguage and its type system, and we back up this claim in Section 5.

More difficult is the question of how, in general, to convert a base language and its type system into an object language. The syntactic portion of the problem is easy—just add support for antiquotation to the object language. Even the type "inference" problem is not too hard if we simply require all free variables and contexts to be fully annotated. However, this seems somewhat draconian and we have three examples of object languages that do not require such an annotation burden but still provide inference—MiniML, Linear MiniML, and C♭—so we expect there to be a general method for providing inference even in the presence of free variables and antiquotation.

Given a base language to convert to an object language, it is clear that we must first solve the *typing* inference problem for the base language. During *type* inference, bindings are known, and uses of a binding generate constraints that are immediately resolved using the binding's definition. For languages based on Hindley-Milner, this immediate resolution is performed using unification. To move from type inference to typing inference, the type system must be extended so that it can capture these constraints *in a type*. For MiniML, these constraints are captured by an extensible intersection type. A typing is then a pair of an environment, which maps free variables to types that accurately reflect the constraints imposed by the uses of the free variables, and a type.

Typings are not quite enough to get us where we want to go because we must also address the issue of antiquotation. It is useful to think of an object language quotation as representing an entire class of base language syntax trees formed by substituting a base language expression for each of the contexts, $\bullet_i$, in the quotation. But each object language quotation must be paired with a typing derivation that parameterizes the base language derivations for the same class of base language expressions that the quotation represents. Furthermore, these term/derivation pairs must be constructed in such a way that when they are composed—that is, when an object language term is "plugged in" to a context and its corresponding derivation is "plugged in" to the parent derivation—the resulting object language term/derivation pair still consists of a valid representative of an entire class of base language term/typing derivations.

Consider a variant of MiniML that did not allow antiquotation. For such a language, intersection types alone would be enough to provide typings—there would be no need for either extension variables or row variables. This wouldn't be a particularly useful language—after all, how would one ever construct a closed term from an open term without antiquotation. However, the point of proposing such an object language is to make the observation that we could represent entire classes of base language derivations using object language types that do not make use of row or extension variables. The need for row and extension variables arises in MiniML *because of antiquotation*. Unlike derivations for our hypothetical antiquotation-free variant, MiniML derivations have "holes" where the derivations for antiquoted terms must eventually be plugged in.

The process for constructing the object language MiniML from a core ML base language consisted of the following steps

- Modify the term structure of expressions to allow object language expressions to represent entire classes of base language expressions by adding support for antiquotation.
- Extend the base language type system so that types can represent the constraints a term's use of free variables imposes on those variables' types.
- Extend the base language type system so that object language derivations represent classes of derivations where subderivations, corresponding to antiquotations, are left free. For MiniML this required adding row and extension variables to represent typing environment constraints shared by a derivation and its free sub-derivations.

We conjecture that only type systems that can be written in a syntax-directed form are suitable for use in an object language as quotations consists of terms and derivations whose holes "line up" in a one-to-one correspondence.

In the following two sections we support our claim that MiniMeta represents a general metaprogramming framework that can support many object languages by describing two additional object languages, a linear variant of MiniML and a C-like language.

5. A linear object language

The type system we have presented is expressed for a pair of languages: a metalanguage and a *specific* object language. It is natural to ask whether or not the type system fragment associated with the metalanguage can be adapted to other object languages, and if so, how difficult this might be. In particular, one might expect that because we expend so much effort on context manipulation, supporting an object language with a substructural type system would be particularly challenging. In this section we demonstrate that although implementing the type checker for a linear object language is non-trivial and requires some novel techniques, integrating it into the metalanguage takes almost zero effort—essentially the only change to the metalanguage that is required is to allow quantification over a newly-kinded type variable.

Our linear language, Linear MiniML, is a variant of MiniML that adds two new binding constructs: a linear **let** binding and a linear λ binding. Its syntax is shown in Figure 7 with the differences with respect to MiniML highlighted.

Many existing presentations of substructural type systems annotate types with use notations that capture, e.g., when variables are used in a linear [38] or unique (in the sense of uniqueness typing) [12] way. While one could imagine using a version of our previous object language's type system extended with such annotations to type open terms, there is a complication that makes this approach a non-starter: antiquotation. The difficulty with antiquotation is that it is not syntactically obvious which variables will be used by a term that is substituted into a context $\bullet_i$, much less which variables it will use linearly!

Term variables		$\in$	$\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{f}, \mathtt{g}, \mathtt{h}$
Type variables		$\in$	$\mathtt{a}, \mathtt{b}, \mathtt{c}$
Extension variables	ρ		
Row variables	γ		
Use variables	υ		
Expressions	e	::=	$\underline{\mathtt{true}} \mid \underline{\mathtt{false}} \mid \mathtt{x} \mid \bullet_i \mid$
			$\lambda \mathtt{x}.\mathtt{e} \mid \lambda^{\mathbf{1}} \mathtt{x}.\mathtt{e} \mid \mathtt{e}_1\ \mathtt{e}_2 \mid$
			$\underline{\mathtt{let}}\ \mathtt{x} = \mathtt{e}_1\ \underline{\mathtt{in}}\ \mathtt{e}_2 \mid$
			$\underline{\mathtt{let}}^{\mathbf{1}}\ \mathtt{x} = \mathtt{e}_1\ \underline{\mathtt{in}}\ \mathtt{e}_2 \mid$
			$(\mathtt{e}_1, \mathtt{e}_2) \mid$
			$\underline{\mathtt{case}}\ \mathtt{e}_1\ \underline{\mathtt{of}}\ \{\ (\mathtt{u}, \mathtt{v}) \to \mathtt{e}_2\ \}$
Types	τ	::=	$\underline{\mathtt{int}} \mid \underline{\mathtt{bool}} \mid \tau_1 \to \tau_2 \mid$
			$\tau_1 \multimap \tau_2 \mid \mathtt{a} \mid (\tau_1, \tau_2)$
Variable use sets	ζ	::=	$\upsilon \mid \{\overline{\mathtt{x}}\} \mid \mathbb{1} \mid \mathbb{0} \mid$
			$\zeta_1 \cap \zeta_2 \mid \zeta_1 \oplus \zeta_2$
Type schemes	σ	::=	$\forall \overline{\mathtt{a}}.\tau$
Environment schemes	ξ	::=	$\bigwedge [\tau_i] \xi \mid \sigma \mid \rho$
Type environments	Γ	::=	$\{\mathtt{x_i} : \xi_i\}\, \Gamma \mid \bot \mid \gamma_{\#\overline{\mathtt{x}}}$
Context environments	Δ	::=	$\{\bullet_i : \Gamma_i \rhd \sigma_i\}$

Figure 7: Linear object language syntax

Consider the following term, in which $\mathtt{z}$ is a *linear* binding, i.e., it must be used linearly in the body of the **let**

$$\lambda x \to \langle\!\langle \underline{\mathtt{let}}^{\mathbf{1}}\ \mathtt{z} = \underline{\mathtt{true}}\ \underline{\mathtt{in}}\ \tilde{}x \rangle\!\rangle$$

Unfortunately we have abstracted over the body of the let, so we must somehow figure out how to assign x a type that reflects the fact that it must use the (free) variable $\mathtt{z}$ linearly.

Our approach to solving this problem is to construct a typing judgment that tracks two sets of disjoint variables that are free in the expression under judgment: those that are use linearly and those that are not used linearly. These sets of variables must contain not just object language variables, such as $\mathtt{z}$, but also *metavariables* υ, that themselves represent sets of variables. These metavariables will allow us to quantify over sets of variables. Our typing judgment has the following form, where $\zeta^{\mathbf{1}}$ is the set of variables used linearly by $\mathtt{e}$ and ζ^{ω} is the set of variables that $\mathtt{e}$ does not use linearly.

$$\Gamma; \Delta \vdash_{\square} \mathtt{e} : \sigma|_{(\zeta^{\mathbf{1}}, \zeta^{\omega})}$$

We claim that our previous quotation of a linear term has the type

$$\forall a, \gamma, \upsilon^{\mathbf{1}}, \upsilon^{\omega}. \boxed{\{\mathtt{z} : \underline{\mathtt{bool}}\}\, \gamma \rhd a|_{(\upsilon^{\mathbf{1}} \oplus \upsilon^{\mathbf{1}} \cap \{z\} \oplus \{z\}, \upsilon^{\omega})}} \to \boxed{\gamma \rhd a|_{(\upsilon^{\mathbf{1}} \oplus \upsilon^{\mathbf{1}} \cap \{z\}, \upsilon^{\omega})}}$$

Though we will shortly show that we can infer this type automatically, we make no claims regarding its readability or programmer-friendliness, so we will walk through our claim carefully. We use the set operations intersection, $\cap$, and symmetric difference, $\oplus$, to represent sets of variables. Our type quantifies over two *variable use metavariables*, $\upsilon^{\mathbf{1}}$ and υ^{ω}. The type of the binder should not look entirely foreign; it says that the binder is an object language term that is only valid in a type environment that must at least contain a binding for $\mathtt{z}$ at type $\underline{\mathtt{bool}}$ and that in such an environment it has the type α. However, its type also places a constraint on the set of variables that the binder must use linearly—we claim that the constraint requires that $\mathtt{z}$ is a member of this set. To see this, let us calculate the intersection of $\upsilon^{\mathbf{1}} \oplus \upsilon^{\mathbf{1}} \cap \{\mathtt{z}\} \oplus \{\mathtt{z}\}$ with the set containing just the variable $\mathtt{z}$

$$\begin{aligned}\left(\upsilon^{\mathbf{1}} \oplus \upsilon^{\mathbf{1}} \cap \{\mathtt{z}\} \oplus \{\mathtt{z}\}\right) \cap \{\mathtt{z}\} &= \upsilon^{\mathbf{1}} \cap \{\mathtt{z}\} \oplus \upsilon^{\mathbf{1}} \cap \{\mathtt{z}\} \oplus \{\mathtt{z}\} \\ &= \{\mathtt{z}\}\end{aligned}$$

A similar calculation shows that the result of applying the lambda is an object expression that is guaranteed not to use a free variable $\mathtt{z}$ linearly.

The typing rules for our linear language are given in Figure 8. They utilize join ($\sqcup$) and meet ($\sqcap$) operators, defined in Figure 9, for combining the two variable use sets according to whether two subexpressions execute in sequence ($\sqcup$) or as alternatives ($\sqcap$) (note that set union and set difference can be expressed in terms of $\cap$ and $\oplus$). We see both mixing operators at work in rule LINIFTHENELSE. The variable uses of e_2 and e_3, the **then** and **else** branch, respectively, of an **if** expression, must be combined so that only those variables that are used linearly in both branches are judged to be used linearly in the body of the **if** expression; this is the job of the meet ($\sqcap$) operator. In contrast, only those variables that are used linearly in *either* the scrutinee of the **if** expression, e_1, or the body, may be judged to be used linearly by the **if** expression as a whole; this is the job of the join ($\sqcup$) operator.

The standard abstraction rule, LINABS, is unsurprising. The application rule, LINAPP, must take care to recognize that because the λ being applied is not guaranteed to use the variable it binds linearly, a variable used linearly by the argument cannot be guaranteed to be used linearly by the application as a whole. Linear abstraction, LINABS$^{\mathbf{1}}$, is like LINABS but with an additional side condition requiring that the variable it binds is present in the set of variables used linearly by the body of the $\lambda^{\mathbf{1}}$. Application of a $\lambda^{\mathbf{1}}$-binding is judged by LINAPP$^{\mathbf{1}}$ to use linearly any variables that are used linearly by the argument since the $\lambda^{\mathbf{1}}$-binding is guaranteed to only use its argument linearly. The other rules follow similarly.

As Kennedy [19] observed in the setting of labels for extensible records, variables sets with the union and symmetric difference operators form a Boolean ring with $\mathbb{0}$ being the empty set and $\mathbb{1}$ the set of all labels. Amazingly, unification over Boolean rings is decidable and unitary [2, 26]. MetaHaskell includes the linear language in Figure 7 as an object language, and the object language's type checker uses Boolean unification to provide type inference.

Coming up with a type system and corresponding inference procedure for our linear language required some thought—we could not simply reuse an existing language as-is. Although we provided a qualitative description of how to go about converting a language and type system to a form suitable for use as an object language in Section 4.4, one has to expect that adding support for *typing* inference (in contrast to just type inference) as well as antiquotation to an existing language will require some non-trivial amount of work. However, adding support for our linear object language required only a few, small changes to the metalanguage in three areas:

1. The MetaHaskell parser was changed to recognize quoted Linear MiniML expressions and types.
2. The algebraic data type representing object language type metavariables was extend with a constructor for the Linear MiniML metavariables that can be quantified at the metalanguage level, i.e., ρ, γ, and υ.

$$\boxed{\Gamma; \Delta \vdash_{\square} e : \sigma|_{(\zeta^{\mathbf{1}}, \zeta^{\omega})}}$$

$$\frac{}{\Gamma; \Delta \vdash_{\square} \{\underline{\texttt{true}}, \underline{\texttt{false}}\} : \underline{\texttt{bool}}|_{(\varnothing,\varnothing)}}\ \textsc{LinBool}$$

$$\frac{\mathtt{x} : \bigwedge [\tau_1 \ldots \tau_n] \, \sigma \in \Gamma}{\Gamma; \Delta \vdash_{\square} \mathtt{x} : \sigma|_{(\mathtt{x},\varnothing)}}\ \textsc{LinVar}$$

$$\frac{\mathtt{x} : \bigwedge [\tau_1 \ldots \tau_i \ldots \tau_n] \, \rho \in \Gamma}{\Gamma; \Delta \vdash_{\square} \mathtt{x} : \tau_i|_{(\mathtt{x},\varnothing)}}\ \textsc{LinFreeVar}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash_{\square} \mathtt{e}_1 : \underline{\texttt{bool}}|_{\overline{\zeta}_1} \\ \Gamma; \Delta \vdash_{\square} \mathtt{e}_2 : \tau|_{\overline{\zeta}_2} \qquad \Gamma; \Delta \vdash_{\square} \mathtt{e}_3 : \tau|_{\overline{\zeta}_3}\end{array}}{\Gamma; \Delta \vdash_{\square} \underline{\texttt{if}}\ \mathtt{e}_1\ \underline{\texttt{then}}\ \mathtt{e}_2\ \underline{\texttt{else}}\ \mathtt{e}_3 : \sigma|_{\overline{\zeta}_1 \sqcup (\overline{\zeta}_2 \sqcap \overline{\zeta}_3)}}\ \textsc{LinIfThenElse}$$

$$\frac{\Gamma, \mathtt{x} : \tau_1; \Delta \vdash_{\square} e : \tau_2|_{(v^{\mathbf{1}}, v^{\omega})}}{\Gamma; \Delta \vdash_{\square} \lambda \mathtt{x}.\mathtt{e} : \tau_1 \to \tau_2|_{(v^{\mathbf{1}} \setminus \mathtt{x}, v^{\omega} \setminus \mathtt{x})}}\ \textsc{LinAbs}$$

$$\frac{\Gamma; \Delta \vdash_{\square} \mathtt{e}_1 : \tau_1 \to \tau_2|_{(u^{\mathbf{1}}, u^{\omega})} \qquad \Gamma; \Delta \vdash_{\square} \mathtt{e}_2 : \tau_1|_{(v^{\mathbf{1}}, v^{\omega})}}{\Gamma; \Delta \vdash_{\square} \mathtt{e}_1\ \mathtt{e}_2 : \tau_2|_{(u^{\mathbf{1}}, u^{\omega}) \sqcup (\varnothing, v^{\mathbf{1}} \cup v^{\omega})}}\ \textsc{LinApp}$$

$$\frac{\Gamma, \mathtt{x} : \tau_1; \Delta \vdash_{\square} e : \tau_2|_{(v^{\mathbf{1}}, v^{\omega})} \qquad \mathtt{x} \in v^{\mathbf{1}}}{\Gamma; \Delta \vdash_{\square} \lambda^{\mathbf{1}} \mathtt{x}.\mathtt{e} : \tau_1 \multimap \tau_2|_{(v^{\mathbf{1}} \setminus \mathtt{x}, v^{\omega})}}\ \textsc{LinAbs}^{\mathbf{1}}$$

$$\frac{\Gamma; \Delta \vdash_{\square} \mathtt{e}_1 : \tau_1 \multimap \tau_2|_{(u^{\mathbf{1}}, u^{\omega})} \qquad \Gamma; \Delta \vdash_{\square} \mathtt{e}_2 : \tau_1|_{(v^{\mathbf{1}}, v^{\omega})}}{\Gamma; \Delta \vdash_{\square} \mathtt{e}_1\ \mathtt{e}_2 : \tau_2|_{(u^{\mathbf{1}}, u^{\omega}) \sqcup (v^{\mathbf{1}}, v^{\omega})}}\ \textsc{LinApp}^{\mathbf{1}}$$

$$\frac{\Gamma; \Delta \vdash_{\square} \mathtt{e}_1 : \sigma|_{(u^{\mathbf{1}}, u^{\omega})} \qquad \Gamma, \mathtt{x} : \sigma; \Delta \vdash_{\square} \mathtt{e}_2 : \tau|_{(v^{\mathbf{1}}, v^{\omega})}}{\Gamma; \Delta \vdash_{\square} \underline{\texttt{let}}\ \mathtt{x} = \mathtt{e}_1\ \underline{\texttt{in}}\ \mathtt{e}_2 : \tau|_{(\varnothing, u^{\mathbf{1}} \cup u^{\omega}) \sqcup (v^{\mathbf{1}} \setminus \mathtt{x}, v^{\omega} \setminus \mathtt{x})}}\ \textsc{LinLet}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash_{\square} \mathtt{e}_1 : \sigma|_{(u^{\mathbf{1}}, u^{\omega})} \\ \Gamma, \mathtt{x} : \sigma; \Delta \vdash_{\square} \mathtt{e}_2 : \tau|_{(v^{\mathbf{1}}, v^{\omega})} \qquad \mathtt{x} \in v^{\mathbf{1}}\end{array}}{\Gamma; \Delta \vdash_{\square} \underline{\texttt{let}}^{\mathbf{1}}\ \mathtt{x} = \mathtt{e}_1\ \underline{\texttt{in}}\ \mathtt{e}_2 : \tau|_{(u^{\mathbf{1}}, u^{\omega}) \sqcup (v^{\mathbf{1}} \setminus \mathtt{x}, v^{\omega})}}\ \textsc{LinLet}^{\mathbf{1}}$$

$$\frac{\Gamma; \Delta \vdash_{\square} \mathtt{e} : \sigma|_{\overline{\zeta}} \qquad \mathtt{a} \notin \mathrm{ftv}(\Gamma) \cup \mathrm{ftv}(\Delta)}{\Gamma; \Delta \vdash_{\square} \mathtt{e} : \forall \mathtt{a}.\sigma|_{\overline{\zeta}}}\ \textsc{LinGen}$$

$$\frac{\Gamma; \Delta \vdash_{\square} \mathtt{e} : \sigma|_{\overline{\zeta}} \qquad \sigma \preceq \sigma'}{\Gamma; \Delta \vdash_{\square} \mathtt{e} : \sigma'|_{\overline{\zeta}}}\ \textsc{LinInst}$$

$$\frac{\bullet_i : \Gamma \rhd \sigma|_{\overline{\zeta}} \in \Delta}{\Gamma; \Delta \vdash_{\square} \bullet_i : \sigma|_{\overline{\zeta}}}\ \textsc{LinAnti}$$

Figure 8: Declarative typing rules for the linear object language

3. The algebraic data type representing object languages types was extended with a construct for Linear MiniML typings, and the metalanguage unification procedure was modified to pass equalities between Linear MiniML types to the Linear MiniML solver.

In total, these changes amounted to a few tens-of-lines of code. Both the metalanguage type system presented in Section 4 and its implementation in the context of MetaHaskell are flexible enough to easily incorporate a variety of object languages, even substructural languages.

$$(u^{\mathbf{1}}, u^{\omega}) \sqcup (v^{\mathbf{1}}, v^{\omega}) = \left((u^{\mathbf{1}} \setminus v^{\omega}) \oplus (v^{\mathbf{1}} \setminus u^{\omega}), u^{\omega} \cup v^{\omega} \cup (u^{\mathbf{1}} \cap v^{\mathbf{1}})\right)$$
$$(u^{\mathbf{1}}, u^{\omega}) \sqcap (v^{\mathbf{1}}, v^{\omega}) = \left(v^{\mathbf{1}} \cap u^{\mathbf{1}}, u^{\omega} \cup v^{\omega} \cup (u^{\mathbf{1}} \oplus v^{\mathbf{1}})\right)$$

Figure 9: Variable use operators

6. Regular Expression Compilation in C♭

One of the original motivations for this work grew out of our experiences using Haskell to generate code for C-like languages in Flask [25] and Nikola [24]. Projects like FFTW [14], ATLAS [40], and SPIRAL [29] show that there are many practical applications for code generation, but that these applications require targeting C-like languages. In this section we describe C♭, a pared-down version of C meant to demonstrate that the MetaHaskell approach to heterogeneous metaprogramming is flexible enough to support these languages.

Although the code generating applications we have cited, with the exception of Nikola, are used in an off-line fashion to generate libraries for later linking with application code, we believe that on-line code generation is also useful. In general, settings where profile-guided optimization [7] is beneficial may also be candidates for run-time code generation. A more concrete application that we believe would benefit from run-time code generation is network packet inspection and processing which often includes a component that performs some kind of evaluation of a decision tree constructed from pattern matching rules. The organization of branches in such a decision tree could easily be optimized given knowledge about the distribution of packets. In practice, this distribution will change over time, and we would like to recompile our pattern matcher periodically to adapt it to the new packet distribution. If rules are added or removed as the packet inspection is executing, there is even more reason to desire run-time code generation.

We present a simple example in this vein: a regular expression compiler that generates executable binary code from a regular expression represented as a string. Because we take advantage of MetaHaskell's support for C♭ as an object language, we don't have to include regular expression compilation as a primitive as .NET does with `RegexOptions.Compiled`. Instead, we can implement our own regular expression compiler as a library without sacrificing type safety.

Figure 10 shows the DFA compilation stage of our regular expression compiler written to use GHC's `QuasiQuotes` language extension (we have elided the other stages suchs as regular expression parsing and NFA to DFA conversion). The function `dfa2c` has type `DFA → C.Func`; it produces an abstract syntax tree representation of a C function that implements the DFA given to it as an argument. This version is not type safe—we have no guarantee about the type of the generated code, although we do at least know that the genrated code will be syntatically correct. However, we can use MetaHaskell to write the same function and gain type safety.

The type-safe version of the DFA compiler, written in MetaHaskell, is given in Figure 11. We have elided some code that is unchanged with respect to the previous version—all that has changed is the type signatures. Furthermore, these signatures are exactly the types inferred by the C♭ object language inference engine and could themselves be elided. Note that the use of the free variables `state` and `accept` in the quotation in the body of `trans2c` is propagated to the type of the quoted C♭ statement in the body of `state2c`.

Although we could have defined an erasure function, as we did with MiniML, to convert a quoted C♭ function definition to its corresponding abstract syntax, this would not have allowed us to maintain type safety when generating compiled code. The $[\![\mathbf{cfun}|\cdot]\!]$ quasiquoter, in addition to elaborating a quoted function

```
dfa2c :: DFA → C.Func
dfa2c (DFA start states) =
    [[cfun|int matches (char * s)
          {
            int state     = $int:(fromIntegral start);
            int accept    = $int:(if daccept startState
                                  then 1
                                  else 0);
            while (*s != 0) {
              switch (state) {
                $stms:(map state2c (map snd states))
              }
            }
            return accept;
          }]]
  where
    startState :: DFAState
    startState = fromJust (lookup start states)
    state2c :: DFAState → C.Stm
    state2c (DFAState i accept trans) =
        [[cstm| case $int:(fromIntegral i):{
                  switch (*s+) {
                    $stms:(map trans2c trans)
                    default:return 0;
                  }
                  break; }]]
    trans2c :: (Char, DLabel) → C.Stm
    trans2c (c, next) =
        [[cstm| case $char:c:{
                    state  = $int:(fromIntegral next);
                    accept = $int:(if daccept nextState
                                   then 1
                                   else 0);
                    break;
                  }]]
      where
        nextState :: DFAState
        nextState = fromJust (lookup next states)
```

Figure 10: DFA compiler in GHC

to a Haskell term representing a FunPtr, must also incorporate a type function that translates C♭ types to Haskell FFI types. The other delta with respect to MiniML and Linear MiniML is C♭'s support for quoting and antiquoting multiple syntactic categories: functions, statements, and expressions can be quoted, and statements, lists of statements, expressions, and various constants can be antiquoted.

7. Implementation

MetaHaskell is implemented in `mhc` which supports Haskell 98 plus MetaHaskell extensions. It includes a type checker that elaborates to FC_2 [39], a bytecode compiler, and a bytecode execution engine. Every MetaHaskell program can in fact be erased to a plain Haskell program, so it would take little effort to modify `mhc` to output pure Haskell.

In essence, an object language quasiquoter consists of a parser, a quasiquote type checker, and a unification procedure for object language types. The parser converts a quasiquote into a pair consisting of the quote itself and a list of antiquotations, each of which is then parsed by `mhc` as a MetaHaskell expression. The quasiquote type

```
dfa2c :: DFA → FunPtr (CString → IO CInt)
dfa2c (DFA start states) = ...
  where
    startState :: DFAState
    startState = ...
    state2c :: DFAState
            → [[cstm|{int accept; int state; char * s}`γ ▷ void]]
    state2c (DFAState i accept trans) = ...
    trans2c :: (Char, DLabel)
            → [[cstm|{int accept; int state}`γ ▷ void]]
    trans2c (c, next) = ...
```

Figure 11: DFA compiler in MetaHaskell

checker takes a quote and the Haskell expressions representing the elaborations of its antiquotes and returns the type of the quote, the types each of its antiquotations must have, an elaborated Haskell term representing the quote, and the type of the elaborated term. `mhc` then checks that the antiquotations have the types specified by the quasiquoter. The `mhc` type inference engine passes equality constraints between object language type to the object language's type inference engine.

Although the type inference engines for MiniML and Linear MiniML required a fair amount of work, incorporating them into `mhc` took comparatively little effort, as we detailed for Linear MiniML in Section 5. The C♭ object language reused a great deal of code from the `language-c-quote` package on Hackage [23]. We also reused much of the code from MiniML for inferring type environments, and as C♭ is not polymorphic, type inference was otherwise straightforward. The run-time code generation feature of C♭ is implemented via unsafePerformIO by calling out to `gcc` to compile a function into a shared library and then dynamically loading the shared library. This use of unsafePerformIO by the erasure of a $[\![\textbf{cfun}|\cdot|]\!]$ term is rendered safe by the extra type information carried by the un-erased term.

8. Related Work

There are a large number of type systems designed explicitly for metaprogramming. We attempt to describe the major players and their important features here. Most of these systems are multi-stage—they support arbitrary nesting of quotation and antiquotation. Stages are typically numbered by the quotation nesting level at which they occur, so programs without any quotations exist entirely at stage 0. All systems are homogeneous, so the object language and the metalanguage are identical. Some of these type systems support "open code" in the sense that quoted terms at stage n may contain free variables, but in all system but one, these free variables must be bound in a previous stage $m < n$. The use of variables bound in stage m by stages $n > m$ is called *cross-stage persistence*. Cross-stage persistence and support for multiple stages make sense for homogeneous metaprogramming languages, but they seem significantly less important in the heterogeneous setting; support for cross-stage persistence would require providing a meaningful translation of values from one language to the other, and multiple stages would require that the object language, which is different from the metalanguage, also be a metaprogramming language.

Mini-ML$^{\Box}$ [10, 11] and its corresponding core calculus, $\lambda^{\Box}$, support staged computation but not open code. In contrast, $\lambda^{\bigcirc}$ [9] supports cross-stage persistence, but it does not provide a way to express (in the type system) the fact that a particular term is closed. Nanevski [27] adds support for intensional analysis to $\lambda^{\Box}$.

λ^{α} [36] classifies open code terms by their environment. Calcagno et al. [5] show that inference for λ^{α} is not possible and give a subset, λ^{i}_{let}, for which inference can be performed. As with $\lambda^{\bigcirc}$, a free variable must be bound in a previous stage. MiniML$^{\text{meta}}_{\text{ref}}$ [4] adds support for computational effects.

Chen and Xi [8] develop λ^{+}_{code} and show that it can be embedded in a language with GADTs. Neither polymorphic object terms nor open object terms are supported. We believe that the combination of quasiquotation and GADTs in Haskell would serve to implement all the features of λ^{+}_{code}.

A prototype implementation of λ^{+}_{code} exists, and λ^{i}_{let} is essentially the language supported by MetaOCaml. None of these systems support explicitly heterogeneous code. However, Eckhardt et al. [13] describe *implicitly* heterogeneous metaprogramming in which the meta- and object languages are both MetaOCaml, but a subset of object language terms can be translated—or *offshored*—automatically to C. Though offshoring cannot handle all MetaOCaml terms, it is capable of handling non-trivial examples [34]. In contrast, our system supports explicit metaprogramming in multiple object languages. We also provide a path to adding heterogeneous metaprogramming support to an existing language through relatively minor language extensions, whereas offshoring must be built on an existing homogeneous metaprogramming language.

The work most similar to ours is Kim et al. [20]. They support open code, inference, and effects for the polymorphic lambda calculus. Like us, they use extensible records to provide principal typings for their multi-stage homogenous metaprogramming language. Unlike MiniML, their system does not support polymorphic uses of free variables, so they cannot type the term

$$(\lambda e \to \langle\!\langle \underline{\texttt{let}}\ \texttt{f} = \lambda \texttt{x} \to \texttt{x}\ \underline{\texttt{in}}\ \tilde{e} \rangle\!\rangle)\langle\!\langle (\texttt{f}\ 1, \texttt{f}\ \underline{\texttt{true}}) \rangle\!\rangle$$

because the free variable `f` is instantiated at two different types in the object term $\langle\!\langle (\texttt{f}\ 1, \texttt{f}\ \underline{\texttt{true}}) \rangle\!\rangle$. They can, however, type the β-reduced form, $\langle\!\langle \underline{\texttt{let}}\ \texttt{f} = \lambda \texttt{x} \to \texttt{x}\ \underline{\texttt{in}}\ (\texttt{f}\ 1, \texttt{f}\ \underline{\texttt{true}}) \rangle\!\rangle$. Rhiger [32] shows how to support open code and inference, but in a simply-typed setting. Nanevski et al. [28] give a foundational account of the kind of modal type system needed to support languages metaprogramming, but they ignore the practical issue of inference. Kim et al. [20] describe an inference algorithm, but they do not provide an implementation.

A significant body of work addresses practical metaprogramming issues in the context of MetaOCaml [6, 17, 18, 21, 34]. We are very interested in adapting ideas from the MetaOCaml community to our setting, particularly those related to efficient object language code generation described by Swadi et al. [34] and Kameyama et al. [18].

9. Conclusions and Future Work

MetaHaskell provides a modular framework for supporting type safe heterogeneous meatprogramming with multiple object languages. Although two of the object languages we describe, MiniML and Linear MiniML, required novel type system features to support antiquotation and open terms, integrating then into MetaHaskell as object languages required little work. Our framework accommodates even exotic languages with substructural type systems. We also provide a methodology for constructing an object language and its type system from a base language that does not support metaprogramming.

MetaHaskell meets many of the goals outlined in Section 3. It provides syntactic support, antiquotation, heterogeneity, type soundness, inference, open terms, and subterm typability. Although it seems we cannot in general provide subterm abstractability, we believe annotations are an acceptable solution. We have not addressed fresh name generation or hygiene, but we expect that hygiene can be provided as outlined in Section 4.3.

Our language only allows construction of type safe object language terms. Intensional analysis of the terms is only possible by first calling a function like `erase` to yield an abstract syntax tree, losing type information in the process. Maintaining types while allowing intensional analysis of object terms requires moving to a dependently typed language. An intermediate solution that we plan to implement in MetaHaskell is a partial function `check` that, given an abstract syntax tree and an object type, will check that the term represented by the abstract syntax tree has the specified type and cast the AST to an object term of that type.

Although we do not describe them here, we have both a syntax-directed and an algorithmic version of the declarative type system for MiniML given in Figure 2. We are in the process of proving progress and preservation, as well as soundness and completeness of the algorithmic system. We use Rémy's unification algorithm [31], which is decidable and unitary, to unify typing contexts, and we use a similar algorithm to unify our extendable intersection types. Our inference algorithm for Linear MiniML is not complete because the type system does not quantify over linearity; inference always makes the assumption that a free variable applied to an argument is not a linear function.

We also plan to translate our work to a dependently typed setting, like Coq. Although embedding an object language in Coq will not require modifications to Coq's type system since a strong embedding of the object language's type system should be possible, our techniques for constructing object languages and providing inference will still be useful. Furthermore, the modifications we made to Haskell's type system to support metaprogramming will provide guidance as we encode object language types in Coq.

Acknowledgments

We are grateful to Greg Morrisett for his early support and helpful discussions. Dimitrios Vytiniotis helped clarify our thinking about generalization in the presence of antiquotation and suggested a greater focus on developing a methodology for constructing object languages from base languages. Claudio Russo and Nick Benton provided useful feedback on drafts of this paper. Andrew Kennedy brought Boolean unification to our attention.

References

[1] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*, pages 37–48, Edinburgh, Scotland, 2009. ACM.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Gershom Bazerman. jmacro, jul 2011.

[4] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(03):545–571, 2003.

[5] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. ML-Like inference for classifiers. In *In European Symposium on Programming (ESOP '04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 79—93, 2004.

[6] Jacques Carette. Gaussian elimination: A case study in efficient genericity with MetaOCaml. *Science of Computer Programming*, 62 (1):3–24, sep 2006.

[7] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, page 246–257, New York, NY, USA, 1989. ACM. ISBN 0-89791-306-X.

[8] Chiyan Chen and Hongwei Xi. Meta-Programming through typeful code representation. *Journal of Functional Programming*, 15(06): 797–835, 2005.

[9] R. Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 184. IEEE Computer Society, 1996.

[10] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–270, St. Petersburg Beach, Florida, United States, 1996. ACM.

[11] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM (JACM)*, 48:555–604, may 2001. ACM ID: 382785.

[12] Edsko de Vries. *Making Uniqueness Typing Less Unique*. PhD thesis, Trinity College, Dublin, Ireland, 2008.

[13] Jason Eckhardt, Roumen Kaiabachev, Emir Pašalić, Kedar Swadi, and Walid Taha. Implicitly heterogeneous multi-stage programming. *New Gen. Comput.*, 25(3):305–336, 2007.

[14] Matteo Frigo and Steven G Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[15] Hideyuki Tanaka. peggy, feb 2012.

[16] Trevor Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–53, St. Petersburg Beach, Florida, United States, 1996. ACM.

[17] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Closing the stage: from staged code to typed closures. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 147–157, San Francisco, California, USA, 2008. ACM.

[18] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-Chieh Shan. Shifting the stage: staging with delimited control. *Journal of Functional Programming*, 21(06):617–662, 2011.

[19] Andrew J. Kennedy. Type inference and equational theories. Technical Report LIX/RR/96/09, LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France, sep 1996.

[20] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 257–268, Charleston, South Carolina, USA, 2006. ACM.

[21] Oleg Kiselyov and Walid Taha. Relating FFTW and Split-Radix. In *Embedded Software and Systems*, pages 488–493. 2005.

[22] Geoffrey Mainland. Why it's nice to be quoted: Quasiquoting for haskell. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*, page 73–82, New York, NY, USA, 2007. ACM.

[23] Geoffrey Mainland. language-c-quote, 2010.

[24] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 67–78, Baltimore, Maryland, USA, 2010. ACM.

[25] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, page 335–346, New York, NY, USA, 2008. ACM.

[26] Urusula Martin and Tobias Nipkow. Boolean unification—The story so far. *Journal of Symbolic Computation*, 7(3–4):275—293, apr 1989.

[27] Aleksandar Nanevski. Meta-programming with names and necessity. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 206–217, Pittsburgh, PA, USA, 2002. ACM.

[28] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9:23:1–23:49, jun 2008. ACM ID: 1352591.

[29] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for Platform-Adapted libraries of signal processing alogorithms. *International Journal of High Performance Computing Applications*, 18(1):21—45, feb 2004.

[30] Didier Rémy. Type inference for records in natural extension of ML. Research Report 1431, Institut National de Recherche en Informatique et Automatisme, 1991.

[31] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993.

[32] Morten Rhiger. First-class open and closed code fragments. *In Proceedings of the Sixth Symposium on Trends in Functional Programming*, 2005.

[33] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 439–450, Charleston, South Carolina, United States, 1993. ACM.

[34] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '06, page 160–169, New York, NY, USA, 2006. ACM. ACM ID: 1111570.

[35] Walid Taha. A gentle introduction to multi-stage programming. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, page 30–50. Springer, 2003.

[36] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, page 26–37, New York, NY, USA, 2003. ACM.

[37] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, pages 203–217, Amsterdam, The Netherlands, 1997. ACM.

[38] Philip Wadler. Is there a use for linear logic? In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '91, page 255–273, New York, NY, USA, 1991. ACM. ISBN 0-89791-433-3.

[39] Stephanie Weirich, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '11)*, Austin, TX, 2011.

[40] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw. Pract. Exper.*, 35(2):101–121, 2005.

[41] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '03*, pages 224–235, New Orleans, Louisiana, USA, 2003.