Programming Research Group

A REFLECTIVE FUNCTIONAL LANGUAGE FOR HARDWARE DESIGN AND THEOREM PROVING

Jim Grundy, Intel Corporation
Tom Melham, Oxford University
John O'Leary, Intel Corporation

PRG-RR-03-16

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

A Reflective Functional Language for Hardware Design and Theorem Proving

Jim Grundy[1]   Tom Melham[2]   John O'Leary[1]

Abstract

This paper introduces *reFLect*, a functional programming language with reflection features intended for applications in hardware design and verification. The *reFLect* language is strongly typed and similar to ML, but has quotation and antiquotation constructs. These may be used to construct and decompose expressions in the *reFLect* language itself. The paper motivates and presents the syntax and type system of this language, which brings together a new combination of pattern-matching and reflection features targeted specifically at our application domain. It also gives an operational semantics based on a new use of contexts as expression constructors, and it presents a scheme for compiling *reFLect* programs into the λ-calculus using the same context mechanism.

1 Introduction

In this paper we describe *reFLect*, a new programming language for applications in hardware design and verification. The *reFLect* language is strongly typed and similar to ML [11], but has quotation and antiquotation constructs. These may be used to construct and decompose expressions in the *reFLect* language itself and provide a form of reflection, similar to that in LISP but in a typed setting. The design of *reFLect* draws on the experience of applying an earlier reflective functional language called *FL* [1] to large-scale verification problems at Intel [13, 14, 15].

Hardware designs are modeled as *reFLect* programs. As with similar work based on Haskell [5, 19] or LISP [12, 16], a key capability is simulation of hardware models by executing functional programs. In *reFLect*, however, we also wish to do various operations on the abstract syntax of models written in the language—for example circuit design transformations [27]. Moreover, we want the *reFLect* language to form the core of a typed higher-order logic for specifying and verifying hardware properties [9, 20], and simultaneously the implementation language of a theorem prover for this logic.

[1]Strategic CAD Labs, Intel Corporation, JF4-211, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA. {jgrundy,joleary}@ichips.intel.com

[2]Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, England. Tom.Melham@comlab.ox.ac.uk

Formal reasoning about hardware is performed using the Forte tool [13], which was originally designed around *FL* but now uses *reFL*^{ect}. Forte includes a theorem prover of similar design to the HOL system [10]. In such systems the object language is embedded as a data-type in the meta-language. Representing object-language expressions as a data-type makes implementing the various term analysis and transformation functions required by a theorem prover straightforward. But separating the object-language and meta-language causes duplication and inefficiency. Theorem provers like HOL, for example, include special code for efficient execution of object-language expressions [4].

In *reFL*^{ect} we have made the data-structure used by the underlying language implementation to represent syntax trees available as a data-type within the language itself. Functions on that data-structure, like evaluation, are also made available. Our aim was to retain all the term inspection and manipulation abilities of the conventional theorem prover approach while borrowing an efficient execution mechanism from the meta-language implementation.

The logic of HOL-like systems is constructed following the model of Church's formulation of simple type theory [6], in which higher-order logic is defined on top of the λ-calculus. Our theorem prover follows this approach and constructs a variant of higher-order logic on top of the *reFL*^{ect} language. The reduction rules for the language in this paper are among the inference rules in our higher-order logic.

The applications just described give *intensional analysis* a primary role in *reFL*^{ect}. The design of our language is therefore different from staged functional languages like MetaML [30] and Template Haskell [25], which are aimed more at program generation and the control and optimization of evaluation. The *reFL*^{ect} language also provides a native pattern matching mechanism designed to make it easy to analyze the structure of code (and logical formulas).

In the sections that follow, we describe *reFL*^{ect} by presenting extensions to the λ-calculus that implement its key features. We first present the syntax and type system, and then give an operational semantics of evaluation. We conclude by presenting a scheme for compiling *reFL*^{ect} programs into the λ-calculus. This compilation scheme forms the basis of the *reFL*^{ect} implementation used at Intel.

2 Examples

The *reFL*^{ect} language augments λ-calculus with a form of quotation, written by enclosing an expression between '$\langle\!\langle$' and '$\rangle\!\rangle$'. The denotation of a quoted expression is its own abstract syntax. There is also an antiquotation mechanism, written by prefixing an expression with '^', that escapes the effect of a quotation.

Quotation and antiquotation may also be used for pattern matching. For example, in theorem proving systems like HOL there are ML functions for constructing and destructing function applications in the object language. In *reFL*^{ect} analogous functions can be implemented for constructing and destructing quoted *reFL*^{ect} applications as follows:

$$\begin{array}{l}
\mathsf{let}\ make\_apply = \lambda f.\,\lambda x.\,\langle\!\langle \hat{}f\cdot\hat{}x\rangle\!\rangle \\
\mathsf{let}\ dest\_apply = \lambda\langle\!\langle \hat{}f\cdot\hat{}x\rangle\!\rangle.\,(f, x)
\end{array}$$

A more complex example is the *reFL*^{ect} function below, which traverses a quoted *reFL*^{ect} expression and commutes the arguments of any addition.

$$\begin{array}{rl}
\mathsf{letrec}\ comm = & \lambda\langle\!\langle \hat{}x + \hat{}y\rangle\!\rangle.\,\langle\!\langle \hat{}(comm\cdot y) + \hat{}(comm\cdot x)\rangle\!\rangle \\
\mid & \lambda\langle\!\langle \hat{}f\cdot\hat{}x\rangle\!\rangle.\,\langle\!\langle \hat{}(comm\cdot f)\cdot\hat{}(comm\cdot x)\rangle\!\rangle \\
\mid & \lambda\langle\!\langle \lambda\hat{}p.\,\hat{}b\rangle\!\rangle.\,\langle\!\langle \lambda\hat{}p.\,\hat{}(comm\cdot b)\rangle\!\rangle \\
\mid & \lambda\langle\!\langle \lambda\hat{}p.\,\hat{}b \mid \hat{}a\rangle\!\rangle.\,\langle\!\langle \lambda\hat{}p.\,\hat{}(comm\cdot b) \mid \hat{}(comm\cdot a)\rangle\!\rangle \\
\mid & \lambda x.\,x
\end{array}$$

For example, the application $comm\cdot\langle\!\langle \lambda x.\,m * x + c\rangle\!\rangle$ evaluates to $\langle\!\langle \lambda x.\,c + m * x\rangle\!\rangle$.

3 Syntax

The syntax of *reFL*^{ect} is similar to that of the typed λ-calculus, but with function abstraction constructed over general patterns, rather than just variables, and with primitive syntax for quotations and anti-quotations.

3.1 Types

The *reFL*^{ect} language is simply typed in the Hindley-Milner style, like ML. A type may be a type variable, written with a lower-case letter from the start of the Greek alphabet: α, β, etc.; or a compound type, made up of a type operator applied to a list of argument types. We use lower-case letters from the end of the Greek alphabet, σ, τ, etc., for syntactic meta-variables ranging over types. Type operators are usually written post-fix, but certain binary type operators, such as $\rightarrow$ and $\times$, are written infix. Atomic types, like *int* and *bool*, are considered to be zero-ary type operators applied to empty lists of arguments. The *reFL*^{ect} type system contains one interesting atomic type: *term*, the type of a quoted *reFL*^{ect} expression. Figure 1 shows the syntax of the *reFL*^{ect} type system assuming a syntactic class of type operator symbols, written c.

We assume the existence of a meta-linguistic function *vars* from types to the sets of type variables that occur in them. We also apply *vars* to sets of types, implicitly taking the union of their sets of variables. Figure 2 defines the function *vars*.

$$\begin{array}{rcll} \sigma, \tau, \ldots & ::= & \alpha \mid \beta \mid \gamma \mid \ldots & \text{– A type variable} \\ & \mid & (\sigma_1, \ldots \sigma_n)c & \text{– A compound type} \end{array}$$

Figure 1: The Syntax of Types

$$\begin{array}{lcl} \mathsf{vars}\,\alpha & = & \{\alpha\} \\ \mathsf{vars}(\sigma_1, \ldots \sigma_n)c & = & \mathsf{vars}\,\sigma_1 \cup \ldots \mathsf{vars}\,\sigma_n \end{array}$$

Figure 2: The Type Variables of a Type

3.1.1 Type Instantiation

A *type instantiation* is a mapping from type variables to types that is the identity on all but finitely many arguments. We use the meta-variables ϕ and χ to stand for type instantiations. We will write $\mathsf{dom}\,\phi$ for the *domain* of ϕ, meaning the set of variables for which ϕ is not the identity. If $\mathsf{dom}\,\phi = \{\alpha_1, \ldots \alpha_n\}$ and $\phi\,\alpha_i = \sigma_i$ for $1 \leq i \leq n$, then we sometimes write ϕ as $[\sigma_1, \ldots \sigma_n/\alpha_1, \ldots \alpha_n]$.

Every type instantiation induces a map from types to types. For any type σ and instantiation ϕ we will write σ_ϕ for the result of applying the map induced by ϕ to σ. The induced map is described in Figure 3.

3.2 Expressions

The syntax of *reFL^ect* expressions, shown in Figure 4, is an extension of the syntax of the λ-calculus. Uppercase letters from the middle of the Greek alphabet, Λ, M, etc., range over expressions. We assume the existence of syntactic classes of constant names and variable names, ranged over by k and v respectively. For clarity of presentation, we will write constants such as $+$, $*$, $\vee$ and , (pairing) in infix position. The syntax requires explicit type annotations for constants, variables, and antiquotations. For example, $v \mathbin{⁏} \sigma$ is a variable with name v and type σ. We may omit type annotations when the type is easily inferred.

Several extensions over the simple λ-calculus are apparent from the grammar.

$$\begin{array}{lcl} \alpha_\phi & = & \phi\,\alpha \\ (\sigma_1, \ldots \sigma_n)c_\phi & = & (\sigma_{1\phi}, \ldots \sigma_{n\phi})c \end{array}$$

Figure 3: Type Instantiation

$$
\begin{array}{llll}
\Lambda, \mathrm{M}, \ldots & ::= & k : \sigma & \text{– Constant} \\
 & | & v : \sigma & \text{– Variable} \\
 & | & \lambda\Lambda.\, \mathrm{M} & \text{– Abstraction} \\
 & | & \lambda\Lambda.\, \mathrm{M} \mid \mathrm{N} & \text{– Alternation} \\
 & | & \Lambda \cdot \mathrm{M} & \text{– Application} \\
 & | & \langle\!\langle \Lambda \rangle\!\rangle & \text{– Quotation} \\
 & | & \hat{}\Lambda : \sigma & \text{– Antiquotation}
\end{array}
$$

Figure 4: The Syntax of Expressions

These are discussed below.

3.2.1 Constants

Constants are not theoretically necessary in a presentation of the λ-calculus and are therefore often omitted. They are, however, important in any practical logic and so we include them here. Constants also play a special role in *reFLect* by facilitating a restricted form of polymorphism.

A practical functional language would normally be based on an extension of the λ-calculus with a polymorphic local let construct. However, simple type-theories based on such extended λ-calculi exhibit Girard's Paradox [7]. Since we use the *reFLect* language as the foundation for such a logic we must eschew this extension. This does not mean that our language lacks all polymorphism; constants may be polymorphic. We assume a top level let command that defines a constant to stand for a closed, possibly polymorphic, expression. The type checking rules given later in section 4.2 will allow that constant to be used at any instance of its polymorphic type. The logical soundness of this restricted form of polymorphism is exhibited in Pitts's semantics for the HOL logic [10].

3.2.2 Quotations

A *reFLect* expression may contain a quoted *reFLect* expression. These are written using the form $\langle\!\langle \Lambda \rangle\!\rangle$. Note that $1 + 2$ is considered semantically equal to 3, but that $\langle\!\langle 1 + 2 \rangle\!\rangle$ is considered semantically different from $\langle\!\langle 3 \rangle\!\rangle$. The expressions $1 + 2$ and 3 both denote the same integer value, namely 3. The expression $\langle\!\langle 1 + 2 \rangle\!\rangle$ denotes the abstract syntax tree of the expression $1 + 2$, which is different from the abstract syntax tree of the expression 3.

The *reFLect* language also has an antiquotation operation, which is used to remove the quotes from around its argument. Antiquotation is written $\hat{}\Lambda : \sigma$ or (omitting the type) just $\hat{}\Lambda$ and may be used only inside quotation. Section 6.1

will explain how in certain circumstances subexpressions of the form $\hat{}\langle\!\langle\Lambda\rangle\!\rangle$ may be reduced to Λ. As an example, consider the expression $\langle\!\langle 1 + \hat{}\langle\!\langle 2+3\rangle\!\rangle\rangle\!\rangle$. This expression may be reduced to to $\langle\!\langle 1 + (2+3)\rangle\!\rangle$. The expressions are considered semantically equal, denoting the same abstract syntax tree.

3.2.3 Abstractions

In the λ-calculus each abstraction binds a single variable. In *reFL*^{ect} an expression may appear in the binding position of an abstraction, which then binds all the free variables of that expression. Not all such expressions will be executable, though all are meaningful. We leave a precise description of which expressions are executable until later. Abstractions with a quotation in the binding position are evaluated by pattern matching. By using these facilities we may write $(\lambda\langle\!\langle \hat{}x + \hat{}y\rangle\!\rangle.\ \langle\!\langle \hat{}y + \hat{}x\rangle\!\rangle)\cdot\langle\!\langle 1+2\rangle\!\rangle$, which is semantically equal to $\langle\!\langle 2+1\rangle\!\rangle$.

Not all attempts to execute an application by pattern matching will succeed, so *reFL*^{ect} includes an alternation construct that can be used to try alternative patterns. Using this construct we may write the following function that commutes the arguments of quoted additions and multiplications:

$$\lambda\langle\!\langle \hat{}x + \hat{}y\rangle\!\rangle.\ \langle\!\langle \hat{}y + \hat{}x\rangle\!\rangle \mid \lambda\langle\!\langle \hat{}x * \hat{}y\rangle\!\rangle.\ \langle\!\langle \hat{}y * \hat{}x\rangle\!\rangle$$

Most logical languages omit pattern matching from their abstract syntax so as to simplify their semantics. We considered doing this with *reFL*^{ect}, but decided against it for two reasons. The first is that pattern matching quoted expressions seemed the most natural interface for inspecting and destructing term values. The second is that we wish to support reasoning about all well-founded *reFL*^{ect} functions including those that make use of pattern matching, which the implemented language also supports on algebraic data-types. The earlier *FL* system excluded pattern matching from the logical language, and so reasoning was performed on expressions after pattern matching had been translated into conditional expressions. In *reFL*^{ect} we support reasoning about expressions in a form closer to the surface syntax in which the user wrote them.

Syntactically, any *reFL*^{ect} expression can appear as a pattern. A natural alternative would be to have a separate syntactic class of patterns, but this was rejected because in the implemented language we allow a rather broad class of patterns. These include literal constants for integers, booleans and string, as well as an open-ended class of patterns built up from data-type constructors for free algebras. A separate grammar for patterns would therefore have to duplicate much of the expression language anyway. In addition, the expression of algorithms that traverse expressions would be more complicated, with separate cases for patterns and other expressions. Users often write expression-traversal code in theorem proving and design transformation applications—unlike in a compiler, where the developers write it once.

We could treat patterns as a subtype of expressions, and use a runtime check when an expression is antiquoted into a pattern position to confirm that it is a valid pattern. We may add such a check in a future version of *reFLect* if our experience suggests it is warranted and we can devise an implementation that does not degrade the performance of theorem proving algorithms that make heavy use of antiquotation for expression construction.

3.3 Contexts

For later use in describing the semantics of *reFLect*, we introduce the notation of a *context* to represent an expression with a number of *holes* that occur at specific subexpression positions in the abstract syntax tree. The notion of context we use here is similar to that readers may be familiar with from other language descriptions, except that the holes in our contexts are typed.

Formally, contexts are described by the same grammar as expressions, with the addition of a new production to represent a hole.

$$\begin{array}{lcll} \Lambda, \mathrm{M}, \ldots & ::= & \ldots & \textit{(as in Figure 4)} \\ & | & \_ \mathbin{⁝} \sigma & \text{– A hole} \end{array}$$

A hole is represented by the symbol '_' annotated by a type. We may omit type annotations on holes in a context when they are irrelevant or easily inferred.

We use the calligraphic letters, $\mathcal{C}$, $\mathcal{D}$, etc., as syntactic meta-variables ranging over contexts. We will use the notation $\mathcal{C}[\_ ⁝ \sigma_1, \ldots \_ ⁝ \sigma_n]$ to indicate that the context $\mathcal{C}$ has the n holes shown. The order in which the holes are indicated is unimportant, except that it be must fixed for any given context. We write $\mathcal{C}[\Lambda_1, \ldots \Lambda_n]$ to stand for the expression resulting from a context $\mathcal{C}[\_ ⁝ \sigma_1, \ldots \_ ⁝ \sigma_n]$, where $\sigma_1, \ldots\ \sigma_n$ are the types of $\Lambda_1, \ldots\ \Lambda_n$ respectively, in which each hole $\_ ⁝ \sigma_i$ has been filled by expression Λ_i. Note that this is different from the usual notion of expression *substitution*, in that there is no renaming to avoid variable capture.

4 Static Semantics

In this section we introduce the two well-formedness criteria for expressions. The first is a notion of 'level' which constrains the nesting of quotations and antiquotations allowed in an expression. The second is a notion of strong typing.

$$\frac{}{0 \vdash \_ \overset{\circ}{\circ} \sigma} \qquad \frac{}{n \vdash k \overset{\circ}{\circ} \sigma} \qquad \frac{}{n \vdash v \overset{\circ}{\circ} \sigma}$$

$$\frac{n \vdash \mathcal{C} \quad n \vdash \mathcal{D}}{n \vdash \lambda \mathcal{C}.\, \mathcal{D}} \qquad \frac{n \vdash \mathcal{C} \quad n \vdash \mathcal{D} \quad n \vdash \mathcal{E}}{n \vdash \lambda \mathcal{C}.\, \mathcal{D} \,\|\, \mathcal{E}}$$

$$\frac{n \vdash \mathcal{C} \quad n \vdash \mathcal{D}}{n \vdash \mathcal{C} \cdot \mathcal{D}} \qquad \frac{n + 1 \vdash \mathcal{C}}{n \vdash \langle\!\langle \mathcal{C} \rangle\!\rangle} \qquad \frac{n \vdash \mathcal{C}}{n + 1 \vdash (\hat{}\mathcal{C} \overset{\circ}{\circ} \sigma)}$$

Figure 5: A Level Consistent Context, $n \geq 0$

4.1 Level

We use the term *level* to mean the number of quotations that surround a subexpression. The level of a quoted subexpression is one higher than the level of the surrounding expression. The level of an antiquoted subexpression is one lower than the level of the surrounding subexpression. The level of an entire expression is zero, and no expression may occur at negative level.

Level is an important notion in *reFL*^{ect} because it affects variable binding and reduction. Generally speaking, expressions that occur at level zero may be reduced while those that occur at a higher level may not. For example the normal form of the expression $(1 + 2, \langle\!\langle 1 + 2 \rangle\!\rangle)$ is $(3, \langle\!\langle 1 + 2 \rangle\!\rangle)$ because the first occurrence of $1+2$ occurs at level zero in the expression and therefore may be reduced, while the second occurrence is at level one and therefore may not.

We formalize our notion of level in relation to contexts. Since all expressions may be considered as contexts with no holes, the definitions and properties we describe for contexts also apply to expressions. We consider a context to be well formed only if all its holes occur at level zero and no portion of the context occurs at a negative level. We will say that such a context is *level consistent.* For example, $\hat{}\_ + 1$ not level consistent, but $\langle\!\langle \hat{}\_ + 1 \rangle\!\rangle$ is.

Figure 5 formalizes our notion of a level consistent context by defining judgments of the form $n \vdash \mathcal{C}$, which should be read as '$\mathcal{C}$ is level consistent at level n'. We may read judgments of the form $0 \vdash \mathcal{C}$ as simply '$\mathcal{C}$ is level consistent'. If the unique derivation of $0 \vdash \mathcal{C}$ contains a subderivation with the intermediate conclusion $n \vdash \mathcal{D}$, then we say that '$\mathcal{D}$ occurs at level n in $\mathcal{C}$'.

The following properties follow from the definition of level consistency.

Proposition 1 *If $\mathcal{C}$ contains no holes and $n \vdash \mathcal{C}$, then $m \vdash \mathcal{C}$ for any $m \geq n$.*

Proposition 2 *For any n and $\mathcal{C}$, there is at most one derivation concluding $n \vdash \mathcal{C}$.*

Proposition 3 *If $\mathcal{C}$ contains one or more holes, then there exists at most one n such that $n \vdash \mathcal{C}$.*

Proposition 4 *If Λ is an expression such that $1 \vdash \Lambda$ then there is a unique context $\mathcal{C}[\_ : \sigma_1, \ldots \_ : \sigma_n]$ and set of expressions $\mathrm{M}_1, \ldots\ \mathrm{M}_n$ such that $\mathcal{C}[\hat{}\mathrm{M}_1 : \sigma_1, \ldots \hat{}\mathrm{M}_n : \sigma_n]$ is syntactically identical to Λ and $0 \vdash \mathcal{C}[\_ : \sigma_1, \ldots \_ : \sigma_n]$.*

Proposition 4 allows us to treat contexts as a form of general constructor for quoted expressions. We will use an expression of the form $\langle\!\langle \mathcal{C}[\hat{}\Lambda_1 : \sigma_1, \ldots \hat{}\Lambda_n : \sigma_n] \rangle\!\rangle$ under the condition $0 \vdash \mathcal{C}[\_ : \sigma_1, \ldots \_ \sigma_n]$ to stand for any quoted expression with level zero subexpressions $\Lambda_1, \ldots\ \Lambda_n$. Many of the remaining figures contain recursive definitions over the structure of expressions that use this property to give the case for quoted expressions. Figures 6 and 9 are typical examples. This mechanism allows us to write our structural definitions such that they traverse only the level zero portions of an expression. This contrasts with the presentation technique used for other reflective languages [30, 25] in which the entire term in traversed, and the traversal function tracks the level of the current expression.

4.2 Typing

All quoted expressions in $\mathit{reFL}^{\mathit{ect}}$ have the same type, *term*. In *FL*, the type of a quoted expression depended on what was inside the quote [1]. For example, $\langle\!\langle x + y \rangle\!\rangle$ had type *int term*, while $\langle\!\langle p \vee q \rangle\!\rangle$ had type *bool term*. The idea was similar to the code type <σ> of MetaML [30]. But this scheme means that certain functions that destruct or traverse the structure of an expression cannot be typed. Such functions are common in our target application domain of theorem proving; the functions in section 2 are typical examples.

Pašalić et al. show how to address the problem of typing transformation routines with dependent types [23]. But there remain functions, like finding free variables, that are important for implementing theorem provers and which still cannot be typed in a dependent type system. Even if it were possible to type such routines with dependent types we would reject this option because we wish to present our end-users, practicing hardware design engineers, with the simplest type system that meets their needs. By giving all quoted expressions the same type, *term*, we can type such expressions in a Hindley-Milner type system. The same decision is made for similar reasons in Template Haskell [25].

This means, of course, that in $\mathit{reFL}^{\mathit{ect}}$ some type-checking must be done at run time.[1] For example the expression $\langle\!\langle 1 + \hat{}x \rangle\!\rangle$ is well-typed and requires x to be of

[1]Unlike Template Haskell, in which second-level type errors can still be caught at compile time.

$$\frac{(\textsf{\textit{mgtype}}\ k)_\phi = \sigma}{\vdash k \mathbin{\overset{\circ}{\circ}} \sigma \colon \sigma} \qquad \frac{}{\vdash v \mathbin{\overset{\circ}{\circ}} \sigma \colon \sigma} \qquad \frac{\vdash \Lambda\colon \sigma \quad \vdash \mathrm{M}\colon \tau}{\vdash \lambda\Lambda.\,\mathrm{M}\colon \sigma \to \tau}$$

$$\frac{\vdash \Lambda\colon \sigma \quad \vdash \mathrm{M}\colon \tau \quad \vdash \mathrm{N}\colon \sigma \to \tau}{\vdash \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N}\colon \sigma \to \tau} \qquad \frac{\vdash \Lambda\colon \sigma \to \tau \quad \vdash \mathrm{M}\colon \sigma}{\vdash \Lambda\cdot\mathrm{M}\colon \tau}$$

$$\frac{0 \vdash \mathcal{C}[\_ \mathbin{\overset{\circ}{\circ}} \sigma_1, \ldots \_ \mathbin{\overset{\circ}{\circ}} \sigma_n] \quad \vdash \Lambda_1\colon \mathit{term} \quad \ldots \quad \vdash \Lambda_n\colon \mathit{term} \quad \vdash \mathcal{C}[v_1 \mathbin{\overset{\circ}{\circ}} \sigma_1, \ldots v_n \mathbin{\overset{\circ}{\circ}} \sigma_n]\colon \tau}{\vdash \langle\!\langle \mathcal{C}[\hat{}\Lambda_1 \mathbin{\overset{\circ}{\circ}} \sigma_1, \ldots \hat{}\Lambda_n \mathbin{\overset{\circ}{\circ}} \sigma_n] \rangle\!\rangle\colon \mathit{term}}$$

Figure 6: A Well Typed Expression

type *term*. But the further requirement that x is bound only to integer-valued expressions cannot be checked statically; it must be enforced at run time.

This design decision goes against the common functional programming ideal of catching as many type errors as possible statically. Our approach, however, is similar to the way typing is handled in conventional theorem-proving systems that have a separate meta-language and object-language, such as HOL. Both languages are strongly typed, but evaluating a meta-language expression may attempt to construct an ill-typed object language expression, resulting in a run-time error. Our experience in the theorem proving domain is that this seemingly 'late' discovery of type errors is not a problem in practice.

4.2.1 A Well Typed Expression

We say Λ is well-typed with type σ if it is level consistent and we may derive the judgment $\vdash \Lambda\colon \sigma$ by the rules of Figure 6. Some of the rules merit explanation:

- We suppose that each constant symbol k has an associated *most general type*, *mgtype* k. The type of a constant named k may be any instance of this type.
- A variable may be explicitly annotated with any type, and it is well-typed with this type.
- If the body of a quotation is well-typed with some type σ then the quotation is well-typed with type *term*. The type of the body does not figure in the type of the quoted expression as a whole.
- An antiquotation expression will be well typed if the body of the antiquotation has type *term*, regardless of the type annotated on the antiquote.

$$\frac{(\mathsf{mgtype}\ k)_\phi = \sigma}{\Gamma \vdash k \mathbin{\overset{\circ}{\circ}} \sigma \colon \sigma} \qquad \frac{(v \mapsto \sigma) \in \Gamma}{\Gamma \vdash v \mathbin{\overset{\circ}{\circ}} \sigma \colon \sigma} \qquad \frac{\Delta \vdash \Lambda\colon \sigma \quad (\Delta \cup \Gamma) \vdash \mathrm{M}\colon \tau}{\Gamma \vdash \lambda\Lambda.\,\mathrm{M}\colon \sigma \to \tau}$$

$$\frac{\Delta \vdash \Lambda\colon \sigma \quad (\Delta \cup \Gamma) \vdash \mathrm{M}\colon \tau \quad \Gamma \vdash \mathrm{N}\colon \sigma \to \tau}{\Gamma \vdash \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N}\colon \sigma \to \tau} \qquad \frac{\Gamma \vdash \Lambda\colon \sigma \to \tau \quad \Gamma \vdash \mathrm{M}\colon \sigma}{\Gamma \vdash \Lambda \cdot \mathrm{M}\colon \tau}$$

$$\frac{\begin{array}{l} 0 \vdash \mathcal{C}[\_ \mathbin{\overset{\circ}{\circ}} \sigma_1, \ldots \_ \mathbin{\overset{\circ}{\circ}} \sigma_n] \\ \Gamma \vdash \Lambda_1\colon \mathit{term} \quad \ldots \quad \Gamma \vdash \Lambda_n\colon \mathit{term} \\ \Delta \vdash \mathcal{C}[v_1 \mathbin{\overset{\circ}{\circ}} \sigma_1, \ldots v_n \mathbin{\overset{\circ}{\circ}} \sigma_n]\colon \tau \end{array}}{\Gamma \vdash \langle\!\langle \mathcal{C}[\hat{\ }\Lambda_1 \mathbin{\overset{\circ}{\circ}} \sigma_1, \ldots \hat{\ }\Lambda_n \mathbin{\overset{\circ}{\circ}} \sigma_n] \rangle\!\rangle\colon \mathit{term}}$$

Figure 7: Type Inference

Proposition 5 *For any Λ there is at most one type σ such that $\vdash \Lambda\colon \sigma$.*

4.2.2 Type Inference

Type inference in *reFLect* takes user input and constructs well-typed expressions, attaching the type annotations required to variables, constants and antiquotations. Users need not include these annotations in their input, though they may if they wish a more restricted type than would otherwise be inferred. The type inference algorithm used is essentially the Hindley-Milner algorithm, which performs type-checking relative to an environment associating each variable with its type. The algorithm is different for *reFLect* in that it performs type checking relative to the typing environment on the top of a stack of such environments. A fresh environment is pushed for each quotation. The stack is popped while traversing an antiquotation.

The *reFLect* type inference system will produce well-typed expressions with the most general type consistent with the rules of figure 7. Each judgment of the form $\Sigma \vdash \Lambda\colon \sigma$ should be interpreted as meaning that the expression Λ may have the type σ under the environment Σ. The environment of a judgment is a map, Γ, from variable names to their types.

Proposition 6 *If $\Sigma \vdash \Lambda\colon \sigma$ can be deduced from the type inference rules in Figure 7, then $\vdash \Lambda\colon \sigma$ may be deduced from the type checking rules in Figure 6.*

4.2.3 Variables and Types

In *reFL*^{ect} the identity of a variable is determined by the combination of its name and type. A well-typed expression may have two or more (different) variables with the same name but different types. The type inference algorithm will never produce such an expression, but they may arise as a result of evaluation. For example, the expression $\langle\!\langle \hat{}\langle\!\langle x \mathrel{⦂} \alpha \to \beta \rangle\!\rangle \cdot \hat{}\langle\!\langle x \mathrel{⦂} \alpha \rangle\!\rangle \rangle\!\rangle$ may be reduced using the rules in section 6.1 to $\langle\!\langle x \mathrel{⦂} \alpha \to \beta \cdot x \mathrel{⦂} \alpha \rangle\!\rangle$. Both these expressions are well typed according to the definition in figure 6, but only the first could be constructed by the type-inference system of figure 7. Accordingly, *subject reduction* holds of *reFL*^{ect} only with respect to the notion of being well-typed, not the stronger property of being type inferable.

Note that while the rules in Figure 7 require variables (in the same scope) in a common quotation to share a common type, they do not require variables with the same name in different quotations to share a type. For example, the type inference system may construct the well typed expression $f \cdot \langle\!\langle 1 + x \rangle\!\rangle \cdot \langle\!\langle \mathsf{T} \wedge x \rangle\!\rangle$.

We could avoid the construction of expressions with multiple variables of the same name and different type if for quoted expressions we retained not only the information about the type of the expression, but also the type-checking environment describing the types of the variables it contains. For antiquotations we would record not only the expected type, but also the prevailing type-checking environment, which describes expectations about the types of incoming variables. The operation to splice one expression into another could then complete a conventional type inference operation on the entire expression.

This approach is not, however, appropriate for our applications in theorem proving. Consider the standard logical rule for conjunction introduction:

$$\frac{\vdash P \qquad \vdash Q}{\vdash P \wedge Q}$$

An implementation of this rule is straightforward in *reFL*^{ect} using quoted expressions. In the rule, P and Q stand for two separate and arbitrary boolean expressions, perhaps with free variables. Logically, the rule is valid even if P and Q contain variables with the same name but different types.

It would complicate the presentation and use of the logic if rules like this were restricted with side-conditions to ensure the consistent typing of variables in the result. The decision to allow well typed expressions containing variables with the same name and different types is one that *reFL*^{ect} shares with the object languages of more conventional theorem proving systems for typed logics, such as HOL.

4.2.4 Extending Static Typing

It is possible to design a more elaborate static typing system than the one just described, with the aim of catching more type errors before runtime. For example,

$$
\begin{array}{lcl}
\_ \mathrel{{}^{\circ}_{\circ}} \sigma_{n\phi} & = & \begin{cases} \_ \mathrel{{}^{\circ}_{\circ}} \sigma & \text{, if } n > 0 \\ \_ \mathrel{{}^{\circ}_{\circ}} \sigma_{\phi} & \text{, if } n = 0 \end{cases} \\
k \mathrel{{}^{\circ}_{\circ}} \sigma_{n\phi} & = & \begin{cases} k \mathrel{{}^{\circ}_{\circ}} \sigma & \text{, if } n > 0 \\ k \mathrel{{}^{\circ}_{\circ}} \sigma_{\phi} & \text{, if } n = 0 \end{cases} \\
v \mathrel{{}^{\circ}_{\circ}} \sigma_{n\phi} & = & \begin{cases} v \mathrel{{}^{\circ}_{\circ}} \sigma & \text{, if } n > 0 \\ v \mathrel{{}^{\circ}_{\circ}} \sigma_{\phi} & \text{, if } n = 0 \end{cases} \\
(\lambda \mathcal{C}.\, \mathcal{D})_{n\phi} & = & \lambda \mathcal{C}_{n\phi}.\, \mathcal{D}_{n\phi} \\
(\lambda \mathcal{C}.\, \mathcal{D} \mid \mathcal{E})_{n\phi} & = & \lambda \mathcal{C}_{n\phi}.\, \mathcal{D}_{n\phi} \mid \mathcal{E}_{n\phi} \\
(\mathcal{C} \cdot \mathcal{D})_{n\phi} & = & \mathcal{C}_{n\phi} \cdot \mathcal{D}_{n\phi} \\
\langle\!\langle \mathcal{C} \rangle\!\rangle_{n\phi} & = & \langle\!\langle \mathcal{C}_{n+1\phi} \rangle\!\rangle \\
(\hat{}\mathcal{C} \mathrel{{}^{\circ}_{\circ}} \sigma)_{n\phi} & = & \hat{}\mathcal{C}_{n-1\phi} \mathrel{{}^{\circ}_{\circ}} \sigma \quad \text{, if } n > 0
\end{array}
$$

Figure 8: Type Instantiation of a Context, $n \geq 0$

it shouldn't really be necessary to wait until runtime to find out that $\langle\!\langle 1 + \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle \rangle\!\rangle$ is going to run into trouble. The only concern is that this extension might complicate the static semantics of the language. The next paragraph shows some less obvious examples that an extended static type system might detect.

Consider the expression $(\langle\!\langle 1 + \hat{}x \rangle\!\rangle, \langle\!\langle \mathsf{T} \wedge \hat{}x \rangle\!\rangle)$. This expression is statically well-typed, but at runtime we can already see that it will fail, because no runtime value of x could contain simultaneously both an integer and a boolean. Now consider the expression $(\langle\!\langle 1 {::} \hat{}x \rangle\!\rangle, \langle\!\langle \mathsf{T} {::} \hat{}x \rangle\!\rangle)$. This expression looks like it might be dynamically type correct as x may be bound to $\langle\!\langle [\,] \mathrel{{}^{\circ}_{\circ}} \alpha\ \mathit{list} \rangle\!\rangle$. However, the reduction rules presented later in Section 6 will not allow the α type variables in the two copies of this expression to be instantiated, and so this too will result in a runtime type failure.

4.2.5 Type Instantiation of Contexts

We may apply a type instantiation to a context by instantiating every type that appears at level zero in the context. We write $\mathcal{C}_{n\phi}$ to indicate the result of applying the type instantiation ϕ to the context (or expression) $\mathcal{C}$ at level n. In the case where n is zero we will simply write $\mathcal{C}_{\phi}$. Type instantiation of a context is defined in Figure 8.

Proposition 7 *If* $0 \vdash \Lambda$ *and* $\vdash \Lambda{:}\, \sigma$*, then* $0 \vdash \Lambda_{\phi}$ *and* $\vdash \Lambda_{\phi}{:}\, \sigma_{\phi}$ *for any type instantiation* ϕ*.*

5 Abstractions

Abstractions in *reFLect* are more complex than in the λ-calculus because any expression may appear in the binding position. This complicates our notion of variable binding and therefore our notion of substitution. Binding and substitution are further complicated by the notion of level. This section describes binding and substitution, and gives an informal introduction to the meaning of abstraction in *reFLect*.

5.1 Binding

An abstraction in the λ-calculus is an expression of the form $\lambda v.\, \Lambda$. The free variables of this expression are the free variables of Λ except for v, which the expression is said to *bind*. Let us ignore the presence of quotation and antiquotation in *reFLect* for a moment and imagine a language that allowed abstractions of the form $\lambda\Lambda.\, \mathrm{M}$. We will say that the free variables of this expression are the free variables of M except for the free variables of Λ.

We now consider the effect of level on binding. Consider $\lambda v.\, v + 1$ and $\lambda w.\, 1 + w$. These expressions have different syntax, but they denote the same semantic object, namely the function that increments its argument. Now consider $\langle\!\langle \lambda v.\, v + 1 \rangle\!\rangle$ and $\langle\!\langle \lambda w.\, 1 + w \rangle\!\rangle$. These expressions denote different semantic objects, namely the syntax of two programs that compute the increment function in different ways. In fact, we even consider the expressions $\langle\!\langle \lambda v.\, v \rangle\!\rangle$ and $\langle\!\langle \lambda w.\, w \rangle\!\rangle$ to be different. They denote semantic objects that represent the syntax of different programs, albeit different programs that both compute the identity function.

In *reFLect*, therefore, the expressions $\lambda v.\, v$ and $\lambda w.\, w$ are equal while $\langle\!\langle \lambda v.\, v \rangle\!\rangle$ and $\langle\!\langle \lambda w.\, w \rangle\!\rangle$ are not. The unquoted λs in the first pair of expressions act as binders, but the quoted λs in the second pair of expressions do not; they act like syntax constructors. This allows us to write functions that construct lambda expressions. Consider β-reducing the expression $(\lambda v.\, \langle\!\langle \lambda\hat{}v.\, \hat{}v + 1 \rangle\!\rangle)\cdot\langle\!\langle w \rangle\!\rangle$ to $\langle\!\langle \lambda\hat{}\langle\!\langle w \rangle\!\rangle.\, \hat{}\langle\!\langle w \rangle\!\rangle + 1 \rangle\!\rangle$. Section 6.1 will explain how this expression may be reduced to $\langle\!\langle \lambda w.\, w + 1 \rangle\!\rangle$. We can think of the *reFLect* expression $\langle\!\langle \lambda\hat{}t.\, \hat{}u \rangle\!\rangle$ as a meta-language program that constructs an object-level abstraction. Viewed from this perspective, t is free in this expression, at least at the meta-level, but any variables in the value t takes on will be bound at the object level in the result.

The approach we take is to consider only those variables that appear at level zero in the binding position of a level zero abstraction to be bound. For example, consider the expression $\lambda\langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle.\, \langle\!\langle \hat{}y + \hat{}x \rangle\!\rangle$, which binds x and y. This denotes a function that pattern matches quoted additions and commutes them. In contrast, consider the expression $\lambda\langle\!\langle x + y \rangle\!\rangle.\, \langle\!\langle y + x \rangle\!\rangle$, which binds no variables. This denotes a function that pattern matches quoted additions where the first argument liter-

$$\begin{array}{lcl}
\mathit{free}\; k : \sigma & = & \{\} \\
\mathit{free}\; v : \sigma & = & \{v : \sigma\} \\
\mathit{free}\; \lambda\Lambda.\, \mathrm{M} & = & \mathit{free}\, \mathrm{M} - \mathit{free}\, \Lambda \\
\mathit{free}\; \lambda\Lambda.\, \mathrm{M} \mid \mathrm{N} & = & (\mathit{free}\, \mathrm{M} - \mathit{free}\, \Lambda) \cup \mathit{free}\, \mathrm{N} \\
\mathit{free}\; \Lambda{\cdot}\mathrm{M} & = & \mathit{free}\, \Lambda \cup \mathit{free}\, \mathrm{M} \\
\mathit{free}\langle\!\langle \mathcal{C}[\hat{}\Lambda_1 : \sigma_1, \ldots \hat{}\Lambda_n : \sigma_n] \rangle\!\rangle & = & \mathit{free}\, \Lambda_1 \cup \ldots \mathit{free}\, \Lambda_n \\
\multicolumn{3}{c}{(\text{where } 0 \vdash \mathcal{C}[\_ : \sigma_1, \ldots \_ : \sigma_n])}
\end{array}$$

Figure 9: Free Variables

ally is 'x' and the second argument literally is 'y', and always returns the quoted addition $\langle\!\langle y + x \rangle\!\rangle$. Patterns with fixed variable names—like this last one—don't appear useful, but they have application in searching for specific variables in a large expression. Figure 9 shows the definition of a function *free*, which describes the free variables of an expression.

5.1.1 Binding and Level

An alternative binding scheme would allow abstractions to bind variables at equal or higher level. In such a system the expression $(\lambda x.\, \langle\!\langle x \rangle\!\rangle){\cdot}1$ would evaluate to $\langle\!\langle 1 \rangle\!\rangle$. This binding scheme is used in MetaML, where it is called *cross-stage persistence.*

Cross-stage persistence is not appropriate for the object language of a theorem prover for standard logics. Consider the formula $\neg(\langle\!\langle x \rangle\!\rangle = \langle\!\langle 1 \rangle\!\rangle)$. This statement seems transparently true, and indeed *reFLect* evaluates this expression to true. We desire this behavior because we want to write programs that distinguish between the syntax of an object-language variable x and the syntax of an object-language constant 1. But if quantifiers were to bind variables at higher levels then we could make the following sequence of deductions using standard logical quantifier rules, leading to an inconsistent logic.

$$\dfrac{\dfrac{\vdash \neg(\langle\!\langle x \rangle\!\rangle = \langle\!\langle 1 \rangle\!\rangle)}{\vdash \forall x.\, \neg(\langle\!\langle x \rangle\!\rangle = \langle\!\langle 1 \rangle\!\rangle)}}{\vdash \neg(\langle\!\langle 1 \rangle\!\rangle = \langle\!\langle 1 \rangle\!\rangle)}$$

Suppes [28] also observes this problem and concludes that 'Rule (II) [the prohibition on binding at higher levels] ... is to be abandoned only for profound reasons.' Taha [29] observes the same problem from the perspective of including intensional analysis in MetaML. He notes, as we do, that intensional analysis requires reductions to be allowed only at level zero, but that this restriction cannot be enforced in a language with cross-stage persistence without loss of confluence.

5.2 The Meaning of Abstractions

The expression that occurs in the binding position of an abstraction in *reFLect* is treated as a pattern. As discussed above, a pattern may bind several variables simultaneously. A pattern may also be partial, in the sense that it does not match all possible values of the relevant type. For example, the pattern in $\lambda\langle\!\langle \hat{}f \cdot \hat{}x \rangle\!\rangle .\, f$ ranges over only that subset of the type of expressions containing syntactic applications. When applied to an expression outside this subset, the result of this function is unspecified.

Moreover, it is syntactically possible for a pattern to contain several instances of a variable, as in $\lambda\langle\!\langle \hat{}x + \hat{}x \rangle\!\rangle .\, \langle\!\langle 2 * \hat{}x \rangle\!\rangle$. We do not require an implementation to evaluate such expressions; any attempt to do so may cause a run-time error. But because such expressions may occur in a logic based on *reFLect*, we need to take at least an informal position on their semantics, so that basic operations like substitution and type instantiation respect this semantics.

One possible approach to the semantics of duplicate pattern variables is to consider only the rightmost occurrence of a variable in a pattern to bind the variable in the body. Then we would expect $(\lambda\langle\!\langle \hat{}x + \hat{}x \rangle\!\rangle .\, \langle\!\langle 2 * \hat{}x \rangle\!\rangle) \cdot \langle\!\langle 1 + 2 \rangle\!\rangle$ to be semantically equal to $\langle\!\langle 2 * 2 \rangle\!\rangle$. This works for patterns that are essentially terms in a free algebra. However, in *reFLect* any expression can occur in pattern position, so we instead take the position that in the pattern of a function such as $\lambda\langle\!\langle \hat{}x + \hat{}x \rangle\!\rangle .\, \langle\!\langle 2 * \hat{}x \rangle\!\rangle$ *both* occurrences of x bind the variable x in the body. The pattern then places a constraint on which applications of the function can be reduced. In this example, the constraint is that the expression to which the function must be an additions of two syntactically identical expressions. Hence we expect $(\lambda\langle\!\langle \hat{}x + \hat{}x \rangle\!\rangle .\, \langle\!\langle 2 * \hat{}x \rangle\!\rangle) \cdot \langle\!\langle 1 + 1 \rangle\!\rangle$ to be semantically equal to $\langle\!\langle 2 * 1 \rangle\!\rangle$. If the constraint is not satisfied, then application of the function is not defined.

In the HOL logic, we would usually express this kind of partially-defined object as an 'under-specified' total function [21]. Formally, one uses a *selection* operator [18] to construct an expression '$\varepsilon\, x.\, P[x]$' with the meaning 'an x such that $P[x]$, or a fixed but unknown value if no such x exists'. With this approach, we can view the abstraction $\lambda\Lambda.\, \mathrm{M}$ as an abbreviation for

$$\varepsilon f.\, \forall\, \textsf{\textit{free}}\, \Lambda.\, f \Lambda = \mathrm{M}$$

For example, $\lambda(x, y).\, y$ is the function $\varepsilon f.\, \forall x\, y.\, f(x, y) = y$. We may then view $\lambda\Lambda.\, \mathrm{M} \mid \mathrm{N}$ as an abbreviation for

$$\lambda v.\, \textsf{if}\ (\forall\, \textsf{\textit{free}}\, \Lambda.\, v \neq \Lambda)\ \textsf{then}\ \mathrm{N}\ v\ \textsf{else}\ (\lambda\Lambda.\, \mathrm{M})\ v$$

where the variable v is chosen to be distinct from all variables in $\textsf{\textit{free}}\{\Lambda, \mathrm{M}, \mathrm{N}\}$.

$$
\begin{array}{lcl}
k{:}\,\sigma\theta & = & k{:}\,\sigma \\
v{:}\,\sigma\theta & = & \theta(v{:}\,\sigma) \\
(\lambda\Lambda.\,\mathrm{M})\theta & = & \lambda\Lambda\iota.\,\mathrm{M}\iota\theta \\
(\lambda\Lambda.\,\mathrm{M} \mid \mathrm{N})\theta & = & \lambda\Lambda\iota.\,\mathrm{M}\iota\theta \mid \mathrm{N}\theta \\
(\Lambda{\cdot}\mathrm{M})\theta & = & \Lambda\theta{\cdot}\mathrm{M}\theta \\
\langle\!\langle \mathcal{C}[\hat{}\Lambda_1{:}\,\sigma_1, \ldots \hat{}\Lambda_n{:}\,\sigma_n] \rangle\!\rangle\theta & = & \langle\!\langle \mathcal{C}[\hat{}\Lambda_1\theta{:}\,\sigma_1, \ldots \hat{}\Lambda_n\theta{:}\,\sigma_n] \rangle\!\rangle
\end{array}
$$

$$
\begin{array}{l}
\quad \text{(where } 0 \vdash \mathcal{C}[\_{:}\,\sigma_1, \ldots \_{:}\,\sigma_n] \text{ and } \iota \text{ is a renaming such that:} \\
\qquad \mathit{dom}\,\iota \subseteq \mathit{free}\,\Lambda, \text{ and } \mathit{dom}\,\theta \cap \iota(\mathit{free}\,\Lambda) = \{\}, \text{ and} \\
\qquad (\mathit{free}\,\mathrm{M} - \mathit{free}\,\Lambda) \cap \iota(\mathit{dom}\,\iota) = \{\}, \text{ and} \\
\qquad \mathit{free}(\theta(\mathit{free}\,\mathrm{M} - \mathit{free}\,\Lambda)) \cap \iota(\mathit{free}\,\Lambda) = \{\})
\end{array}
$$

Figure 10: Substitution

5.3 Substitution and Type Instantiation

Substitution and type instantiation in *reFLect* are a little more complex than in the λ-calculus, owing to the presence of pattern matching. The two operations are defined as follows.

5.3.1 Substituting Expressions

A *substitution* is a mapping from variables to expressions of the same type that is the identity on all but finitely many variables. We typically use the meta-variables θ and ι to stand for substitutions. We write $\mathit{dom}\,\theta$ for the *domain* of θ, meaning the set of variables for which θ is not the identity. If $\mathit{dom}\,\theta = \{v_1{:}\,\sigma_1, \ldots v_n{:}\,\sigma_n\}$ and $\theta(v_i{:}\,\sigma_i) = \Lambda_i$ for all $1 \leq i \leq n$, then we sometimes write θ using the notation $[\Lambda_1, \ldots \Lambda_n / v_1{:}\,\sigma_1, \ldots v_n{:}\,\sigma_n]$. A *renaming* is an injective substitution that maps variables to variables.

For any expression Λ and substitution θ we may write $\Lambda\theta$ to stand for the action of applying the substitution to all the free variables of Λ, with appropriate renaming of the bound variables in Λ to avoid capture. Figure 10 defines this operation.[2]

Note that substitution must be consistent with the interpretation we place on repeated pattern variables. We require the result of $(\lambda(x, x).\, y)\,[x/y]$ to be $\lambda(x', x').\, x$. That is, both occurrences of x in the pattern are renamed.

Proposition 8 *If* $0 \vdash \Lambda$ *and* $\vdash \Lambda{:}\,\sigma$*, then* $0 \vdash \Lambda\theta$ *and* $\vdash \Lambda\theta{:}\,\sigma$ *for any substitution* θ*.*

[2]In the condition of figure 10, ι, θ, and *free* are implicitly extended to image functions over sets where required.

Most HOL-style theorem provers have a more general substitution primitive, which allows one to substitute for arbitrary *subexpressions* occurring free in an expression, not just for free variables. This is also the case in the *reFLect* theorem prover, but variable-substitution suffices for presenting the operational semantics.

5.3.2 Type Instantiation

We may also apply a type instantiation to an expression. For any expression Λ and type instantiation ϕ, we write $\Lambda\phi$ to mean the result of applying the instantiation to the expression. This applies the instantiation to every level zero type in the expression, using the notion of instantiation defined in Section 3.1.1. Since the identity of a variable in *reFLect* consists of its name and type, we need to rename bound variables to avoid capture during a type instantiation. For example, $(\lambda(x\,{\circ\atop\circ}\,\alpha, x\,{\circ\atop\circ}\,\beta).\, x\,{\circ\atop\circ}\,\alpha)[\beta/\alpha]$ should produce $\lambda(x'\,{\circ\atop\circ}\,\beta, x\,{\circ\atop\circ}\,\beta).\, x'\,{\circ\atop\circ}\,\beta$ or $\lambda(x\,{\circ\atop\circ}\,\beta, x'\,{\circ\atop\circ}\,\beta).\, x\,{\circ\atop\circ}\,\beta$.

The formal definition of type instantiation for expressions is similar to the definition of substitution in Figure 10. Note that $\Lambda\phi$ is not the same as the context type instantiation operation $\mathcal{C}_\phi$ in Figure 8, which does not rename variables to avoid capture. We will not use type instantiation on expressions as described here until section 8.1.

6 Operational Semantics

Figures 11 and 12 present the reduction rules for evaluating a *reFLect* expression. The rules in Figure 11 describe individual reductions, while those in Figure 12 describe how reductions may be applied to subexpressions. The judgments are of the form $\vdash \Lambda \to \Lambda'$, which means that Λ reduces to Λ' in one step. These rules ensure that reductions apply only to level zero subexpressions, and then only to those that do not fall in the binding position of a level zero abstraction. We use the standard notation $\vdash \Lambda \xrightarrow{*} \Lambda'$ to indicate that Λ can be reduced to Λ' in zero or more steps. This is formalized in figure 13.

The rules of Figure 11 use some auxiliary meta-functions, which we briefly introduce here and describe in more detail later. The function *definition* returns the definition of a constant. The predicate *pattern* characterizes the expressions we consider valid for pattern matching against, variables or quotations whose level zero subexpressions are variables. The relation (Λ, θ) *matches* Ξ means that applying the substitution θ to the pattern Λ causes it to match the expression Ξ (in a sense we define precisely later). The relation Λ *ready* M means that the expression M has been sufficiently evaluated to determine whether or not it matches the pattern Λ.

Proposition 9 *If* $0 \vdash \Lambda$ *and* $\vdash \Lambda\colon \sigma$*, then for any* M *such that* $\vdash \Lambda \xrightarrow{*} \mathrm{M}$ *we have* $0 \vdash \mathrm{M}$ *and* $\vdash \mathrm{M}\colon \sigma$*.*

$$\frac{\vdash \textsf{\textit{definition}}\,k\colon\tau \quad \tau_\phi = \sigma}{\vdash k \mathbin{\overset{\circ}{\circ}} \sigma \rightarrow (\textsf{\textit{definition}}\,k)\phi}\,[\delta]$$

$$\frac{\textsf{\textit{pattern}}\,\Lambda \quad \Lambda\ \textsf{\textit{ready}}\ \Xi \quad (\Lambda,\theta)\ \textsf{\textit{matches}}\ \Xi}{\vdash (\lambda\Lambda.\,\mathrm{M})\cdot\Xi \rightarrow \mathrm{M}\theta}\,[\beta]$$

$$\frac{\textsf{\textit{pattern}}\,\Lambda \quad \Lambda\ \textsf{\textit{ready}}\ \Xi \quad (\Lambda,\theta)\ \textsf{\textit{matches}}\ \Xi}{\vdash ((\lambda\Lambda.\,\mathrm{M}) \mid \mathrm{N})\cdot\Xi \rightarrow \mathrm{M}\theta}\,[\gamma]$$

$$\frac{\textsf{\textit{pattern}}\,\Lambda \quad \Lambda\ \textsf{\textit{ready}}\ \Xi \quad \nexists\theta.\,(\Lambda,\theta)\ \textsf{\textit{matches}}\ \Xi}{\vdash ((\lambda\Lambda.\,\mathrm{M}) \mid \mathrm{N})\cdot\Xi \rightarrow \mathrm{N}\cdot\Xi}\,[\zeta]$$

$$\frac{0 \vdash \mathcal{C}[\_\mathbin{\overset{\circ}{\circ}}\sigma_1,\ldots\_\mathbin{\overset{\circ}{\circ}}\sigma_n] \quad \vdash \Lambda_1\colon{\sigma_1}_\phi \quad \ldots \quad \vdash \Lambda_n\colon{\sigma_n}_\phi \quad \textsf{\textit{dom}}\,\phi \subseteq \textsf{\textit{vars}}\{\sigma_1,\ldots\sigma_n\}}{\vdash \langle\!\langle \mathcal{C}[\hat{}\langle\!\langle\Lambda_1\rangle\!\rangle\mathbin{\overset{\circ}{\circ}}\sigma_1,\ldots\hat{}\langle\!\langle\Lambda_n\rangle\!\rangle\mathbin{\overset{\circ}{\circ}}\sigma_n]\rangle\!\rangle \rightarrow \langle\!\langle \mathcal{C}_\phi[\Lambda_1,\ldots\Lambda_n]\rangle\!\rangle}\,[\psi]$$

Figure 11: Reduction

$$\frac{\vdash \mathrm{M} \to \mathrm{M}'}{\vdash \lambda\Lambda.\,\mathrm{M} \to \lambda\Lambda.\,\mathrm{M}'} \qquad \frac{\vdash \mathrm{M} \to \mathrm{M}'}{\vdash \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N} \to \lambda\Lambda.\,\mathrm{M}' \mid \mathrm{N}} \qquad \frac{\vdash \mathrm{N} \to \mathrm{N}'}{\vdash \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N} \to \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N}'}$$

$$\frac{\vdash \Lambda \to \Lambda'}{\vdash \Lambda\cdot\mathrm{M} \to \Lambda'\cdot\mathrm{M}} \qquad \frac{\vdash \mathrm{M} \to \mathrm{M}'}{\vdash \Lambda\cdot\mathrm{M} \to \Lambda\cdot\mathrm{M}'}$$

$$\frac{0 \vdash \mathcal{C}[\_ ⦂ \sigma_1, \ldots \_ ⦂ \sigma_m, \ldots \_ ⦂ \sigma_n] \quad \vdash \Lambda_m \to \Lambda'_m}{\vdash \langle\!\langle \mathcal{C}[\hat{}\Lambda_1 ⦂ \sigma_1, \ldots \hat{}\Lambda_m ⦂ \sigma_m, \ldots \hat{}\Lambda_n ⦂ \sigma_n] \rangle\!\rangle \to \langle\!\langle \mathcal{C}[\hat{}\Lambda_1 ⦂ \sigma_1, \ldots \hat{}\Lambda'_m ⦂ \sigma_m, \ldots \hat{}\Lambda_n ⦂ \sigma_n] \rangle\!\rangle}$$

Figure 12: Reducing Subexpressions

$$\frac{}{\vdash \Lambda \xrightarrow{*} \Lambda} \qquad \frac{\vdash \Lambda \to \mathrm{M} \quad \vdash \mathrm{M} \xrightarrow{*} \mathrm{N}}{\vdash \Lambda \xrightarrow{*} \mathrm{N}}$$

Figure 13: Reduction Closure

Proposition 9 is the subject reduction property for *reFLect*. The property states that a level consistent and well typed expression remains so as it is reduced, and that the expression retains the same type as it is reduced. Krstić and Matthews have a proof of this property [17].

6.1 Reducing Quotations

The rule for ψ-reduction in Figure 11 allows the elimination of antiquoted quotations at level one. The rule caters for the possibility that the type variables of a quoted region may need to be instantiated in order to be type consistent with the antiquoted regions being spliced into it. Suppose, for example, that *inc* is a constant of type *int* $\to$ *int*. The ψ rule lets us reduce $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle ⦂ \alpha \to \beta \cdot \hat{}\langle\!\langle 1 \rangle\!\rangle ⦂ \alpha \rangle\!\rangle$ to $\langle\!\langle inc \cdot 1 \rangle\!\rangle$ by allowing α and β to be instantiated to *int*.

This type-instantiation behavior of ψ is the basis for run-time type checking in *reFLect*. At compile time, we type-check quotation contexts at their most general types. Then at run-time—when the expressions being spliced into the holes become available—we check type consistency by instantiating the context's type variables to match the types inside the incoming expressions. For example, consider the function *comm* in Section 2. Static type checking will assign polymorphic types to the quotations in the definition of *comm* so that, for example, $comm \cdot \langle\!\langle inc \cdot (1 + 2) \rangle\!\rangle$

reduces at run time to $\langle\!\langle \hat{}\langle\!\langle \mathit{inc} \rangle\!\rangle \mathrel{⦂} \alpha \to \beta \cdot \hat{}\langle\!\langle 2+1 \rangle\!\rangle \mathrel{⦂} \alpha \rangle\!\rangle$. Then, using ψ-reduction, we get the expected expression $\langle\!\langle \mathit{inc} \cdot (2+1) \rangle\!\rangle$.

The rule does not allow reductions to create badly-typed expressions. For example, we cannot use this rule to reduce the expression $\langle\!\langle \hat{}\langle\!\langle \mathit{inc} \rangle\!\rangle \mathrel{⦂} \alpha \to \beta \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle \mathrel{⦂} \alpha \rangle\!\rangle$. Note also that the rule does not allow type instantiations of the expressions inside the antiquotes. For example, we cannot use this rule to reduce the expression $\langle\!\langle \hat{}\langle\!\langle f \mathrel{⦂} \alpha \to \beta \rangle\!\rangle \mathrel{⦂} \mathit{int} \to \mathit{int} \cdot 1 \rangle\!\rangle$.

6.1.1 Instantiation Must Affect the Entire Term

One might first imagine a simpler rule for ψ-reduction like the one shown below:

$$\frac{\vdash \Lambda : \tau_\phi}{\vdash \hat{}\langle\!\langle \Lambda \rangle\!\rangle \mathrel{⦂} \tau \to \Lambda}$$

Unfortunately the effect of this rule does not cover enough of the expression to ensure type consistency. Consider again the expression $\langle\!\langle \hat{}\langle\!\langle \mathit{inc} \rangle\!\rangle \mathrel{⦂} \alpha \to \beta \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle \mathrel{⦂} \alpha \rangle\!\rangle$. We could use this incorrect rule to reduce it to $\langle\!\langle \mathit{inc} \mathrel{⦂} \mathit{int} \to \mathit{int} \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle \mathrel{⦂} \alpha \rangle\!\rangle$ and then again to $\langle\!\langle \mathit{inc} \mathrel{⦂} \mathit{int} \to \mathit{int} \cdot \mathsf{T} \mathrel{⦂} \mathit{bool} \rangle\!\rangle$.

6.1.2 All Antiquotes Eliminated Simultaneously

The assumption $0 \vdash \mathcal{C}[\_ \mathrel{⦂} \sigma_1, \ldots \_ \mathrel{⦂} \sigma_n]$ of the ψ-reduction rule ensures that it eliminates every level one antiquote enclosed by a given quotation. We could imagine a version of this rule that need not eliminate every antiquote simultaneously. We could then reduce $\langle\!\langle \hat{}\langle\!\langle \mathit{inc} \rangle\!\rangle \mathrel{⦂} \alpha \to \beta \cdot \hat{}\langle\!\langle 1 \rangle\!\rangle \mathrel{⦂} \alpha \rangle\!\rangle$ to $\langle\!\langle \mathit{inc} \cdot \hat{}\langle\!\langle 1 \rangle\!\rangle \mathrel{⦂} \mathit{int} \rangle\!\rangle$ and later to $\langle\!\langle \mathit{inc} \cdot 1 \rangle\!\rangle$. But this rule would also allow us to reduce $\langle\!\langle \hat{}\langle\!\langle \mathit{inc} \rangle\!\rangle \mathrel{⦂} \alpha \to \beta \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle \mathrel{⦂} \alpha \rangle\!\rangle$ to both $\langle\!\langle \mathit{inc} \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle \mathrel{⦂} \mathit{int} \rangle\!\rangle$ and $\langle\!\langle \hat{}\langle\!\langle \mathit{inc} \rangle\!\rangle \mathrel{⦂} \mathit{int} \to \beta \cdot \mathsf{T} \rangle\!\rangle$. Since these expressions may not be further reduced this would leave *reFLect* with a non-confluent reduction system.[3]

We could, however, allow a rule that requires only that all the antiquotes occurring *at the same level* within a given quoted region need be eliminated simultaneously. For example, consider

$$\langle\!\langle (\hat{}\langle\!\langle 1 \rangle\!\rangle, \hat{}\langle\!\langle 2 \rangle\!\rangle, \langle\!\langle (\hat{}\hat{}\langle\!\langle \langle\!\langle 3 \rangle\!\rangle \rangle\!\rangle, \hat{}\hat{}\langle\!\langle \langle\!\langle 4 \rangle\!\rangle \rangle\!\rangle) \rangle\!\rangle) \rangle\!\rangle$$

The first two antiquotes must be eliminated simultaneously, and so must the second two, but it would be possible to develop a valid semantics that did not require all four to be eliminated together. Expressions like this, however, do not arise in our applications—so we do not complicate the semantics to facilitate this relaxation.

[3]It may still have some property similar to confluence, in which expressions like these are considered equivalent.

6.1.3 Type Instantiation Impacts Only the Context

The ψ-reduction operation ensures that it constructs a well typed expression by type instantiating the context into which the antiquoted expressions are spliced. One might also consider *unifying* the types of the context and the incoming expressions to achieve a match. This is the approach taken in the system of Shields et al. [26].

This option was rejected for reasons that derive from the target application of *reFL*^{ect} to theorem proving and circuit transformation. In these applications most operations that manipulate expressions are expected to preserve the types of the manipulated expressions. In this case, unification is not appropriate. This is in contrast to systems designed for code-generation [25] or staged evaluation [30], which focus more on flexible ways of constructing or specializing programs.

For example, a ubiquitous theorem proving application is term rewriting [22], in which an expression is transformed by application of general rewrite rules to its subexpressions. The matching that makes a general rewrite rule applicable at a subexpression is always one-way and type unification is not appropriate. The semantics of our hole-filling ψ rule therefore exactly achieves the *reFL*^{ect} design requirement for a native mechanism to support rewriting.

In theorem proving and transformation applications, contexts are typically small and the incoming expressions very large. The same expression may also be spliced into more than one context. If we unified types when splicing an expression into a context, we could not do it by destructively instantiating type variables, a constant time operation. Rather, we would have to copy incoming expressions using time and space proportionate to their size. Since the speed of rewriting is key to the effectiveness of a theorem prover, we would not be able to use this splicing operation to implement our rewriter.

6.2 Patterns May Not Be Reduced

An examination of the rules in Figure 12 reveals that it is possible to reduce any level zero subexpression, except those in the binding position of an abstraction. Patterns may not be reduced. We might imagine a system that allowed reductions on patterns as well. For example, it seems reasonable to reduce the expression $(\lambda\langle\!\langle \hat{}\langle\!\langle 1\rangle\!\rangle + \hat{}x\rangle\!\rangle.\, x)\cdot\langle\!\langle 1+2\rangle\!\rangle$ to $(\lambda\langle\!\langle 1 + \hat{}x\rangle\!\rangle.\, x)\cdot\langle\!\langle 1+2\rangle\!\rangle$ and then to $\langle\!\langle 2\rangle\!\rangle$.

But unrestricted reduction of patterns is unsafe. As an example, consider the expression $\lambda(\lambda y.\, z)\cdot x.\, x$, in which the pattern $(\lambda y.\, z)\cdot x$ occurs in binding position. If we were to allow reduction of this pattern, we could reduce the whole expression to $\lambda z.\, x$. But then the variable x, which was bound in the original expression, has become free—perhaps to be captured by some enclosing scope. It might be possible to avoid this problem by not allowing pattern reductions that change the free variable set of the pattern. But in the absence of a compelling application, it

$$\frac{}{\textit{pattern}\ v : \sigma} \qquad \frac{0 \vdash \mathcal{C}[\_ : \sigma_1, \ldots \_ : \sigma_n]}{\textit{pattern} \langle\!\langle \mathcal{C}[\hat{}(v_1 : \mathit{term}) : \sigma_1, \ldots \hat{}(v_n : \mathit{term}) : \sigma_n] \rangle\!\rangle}$$

Figure 14: Valid Pattern

seems simpler just to forbid all pattern reductions.

6.3 Pattern Matching

The rules for β-reduction, γ-reduction, and ζ-reduction apply only to abstractions over *valid* patterns. Not all expressions make valid patterns. For example, the expressions in the binding positions of $\lambda x.\, x$ and $\lambda \langle\!\langle \hat{}x + 1 \rangle\!\rangle.\, \langle\!\langle 1 + \hat{}x \rangle\!\rangle$ are both valid patterns, but the binding expression in $\lambda x + 1.\, x$ is not. This is not to say that such bindings are without meaning, only that we do not support the evaluation of such patterns, and so they are considered invalid for the purposes of this operational semantics.

Figure 14 defines the predicate *pattern* that characterizes which patterns are considered valid. It can be summarized by saying that a valid pattern is either a variable or a quotation where every level zero subexpression is a variable. The definition does not rule out patterns containing more than one instance of the same variable. An implementation, however, may have a stricter notion of valid pattern that disallows this. Any attempt to match a invalid pattern should lead to a run-time failure.

We also make some restrictions on when we are prepared to consider matching a pattern. If a pattern is a simple variable, then we may match it straightaway, but if a pattern is a quotation then we must wait until the expression we are trying to match has been reduced to a quotation with level one antiquotes eliminated. We will say that the expression M is *ready* to be matched to the pattern Λ, Λ *ready* M, if this condition holds. Figure 15 formalizes this notion, which is used in the rules for β and γ-reduction in figure 11.

Consider what can happen without this restriction by contemplating the effect of *dest_apply* from section 2. If we apply this to the expression $\langle\!\langle g : \alpha \to \alpha \cdot \hat{}\langle\!\langle 1 \rangle\!\rangle : \alpha \rangle\!\rangle$ and

$$\frac{}{v\ \textit{ready}\ \Lambda} \qquad \frac{0 \vdash \Lambda}{\mathrm{M}\ \textit{ready}\ \langle\!\langle \Lambda \rangle\!\rangle}$$

Figure 15: Match Readiness

$$\frac{v{:}\sigma\,\theta = \Xi}{(v{:}\sigma, \theta) \textsf{ matches } \Xi}$$

$$\frac{\begin{array}{l} 0 \vdash \mathcal{C}[\_{:}\sigma_1, \ldots \_{:}\sigma_n] \quad \phi \vdash \mathcal{C}[w_1{:}\sigma_1, \ldots w_n{:}\sigma_n] \rightsquigarrow \mathcal{D}[w_1{:}\sigma_{1\phi}, \ldots w_n{:}\sigma_{n\phi}] \\ v_1{:}\,\mathit{term}\,\theta = \langle\!\langle \Xi_1 \rangle\!\rangle \quad \ldots \quad v_n{:}\,\mathit{term}\,\theta = \langle\!\langle \Xi_n \rangle\!\rangle \end{array}}{\begin{array}{l}(\langle\!\langle \mathcal{C}[\hat{}(v_1{:}\,\mathit{term}){:}\sigma_1, \ldots \hat{}(v_n{:}\,\mathit{term}){:}\sigma_n] \rangle\!\rangle, \theta) \textsf{ matches } \langle\!\langle \mathcal{D}[\Xi_1, \ldots \Xi_n] \rangle\!\rangle \\ \quad \text{(where } w_1, \ldots\ w_n \text{ are fresh)}\end{array}}$$

Figure 16: Pattern Matching an Expression

we were to evaluate the application before ψ-reducing the argument we would get the result $(\langle\!\langle g{:}\,\alpha \to \alpha \rangle\!\rangle, \langle\!\langle \hat{}\langle\!\langle 1 \rangle\!\rangle{:}\,\alpha \rangle\!\rangle)$, which would then reduce to $(\langle\!\langle g{:}\,\alpha \to \alpha \rangle\!\rangle, \langle\!\langle 1 \rangle\!\rangle)$. If we were to ψ-reduce the argument before reducing the application we would get the result $(\langle\!\langle g{:}\,\mathit{int} \to \mathit{int} \rangle\!\rangle, \langle\!\langle 1 \rangle\!\rangle)$.

As with the possible generalization to ψ-reduction discussed in Section 6.1.2, we believe there is an equally valid semantics that doesn't force the elimination of all level one antiquotes from an expression before it may be matched, but only those from level contiguous regions that are in some way accessed by the match. But this would complicate the semantics without benefit to practical applications.

6.3.1 Matching An Alternative

Once we have determined that a pattern is valid and an expression is ready to be matched by it then we are ready to determine whether (and how) the expression matches the pattern. The predicate *matches*, defined in Figure 16, makes this determination.

When the pattern is a variable, we say that the pattern matches an expression under a substitution precisely when the substitution maps that variable to the expression. When the pattern is a quotation, we first find a level-consistent context $\mathcal{C}[\_{:}\sigma_1, \ldots \_{:}\sigma_n]$ and term variables $v_1, \ldots\ v_n$ such that the pattern we are trying to match against is $\langle\!\langle \mathcal{C}[\hat{}(v_1{:}\,\mathit{term}){:}\sigma_1, \ldots \hat{}(v_n{:}\,\mathit{term}){:}\sigma_n] \rangle\!\rangle$. Next we must find a level consistent context $\mathcal{D}[\_{:}\tau_1, \ldots \_{:}\tau_n]$ and list of subexpressions $\Xi_1, \ldots\ \Xi_n$ such that $\langle\!\langle \mathcal{D}[\Xi_1, \ldots \Xi_n] \rangle\!\rangle$ is the expression we are trying to match. The expression matches the pattern if $\mathcal{D}$ is a type instance, in the sense explained below, of $\mathcal{C}$ and we can match each expression $\Xi_1, \ldots\ \Xi_n$ to the corresponding variable $v_1, \ldots\ v_n$ under the same substitution.

The notation $\phi \vdash \Lambda \rightsquigarrow \mathrm{M}$ indicates that M is a type instance of Λ under some type instantiation ϕ and is defined in Figure 17. The role of the $\rightsquigarrow$ relation is to

$$\frac{\sigma_\phi = \tau}{\phi \vdash k \mathbin{⦂} \sigma \rightsquigarrow k \mathbin{⦂} \tau} \qquad \frac{\sigma_\phi = \tau}{\phi \vdash v \mathbin{⦂} \sigma \rightsquigarrow v \mathbin{⦂} \tau}$$

$$\frac{\phi \vdash \Lambda \rightsquigarrow \Lambda' \quad \phi \vdash \mathrm{M} \rightsquigarrow \mathrm{M}'}{\phi \vdash \lambda\Lambda.\, \mathrm{M} \rightsquigarrow \lambda\Lambda'.\, \mathrm{M}'}$$

$$\frac{\phi \vdash \Lambda \rightsquigarrow \Lambda' \quad \phi \vdash \mathrm{M} \rightsquigarrow \mathrm{M}' \quad \phi \vdash \mathrm{N} \rightsquigarrow \mathrm{N}'}{\phi \vdash \lambda\Lambda.\, \mathrm{M} \mathbin{\boldsymbol{|}} \mathrm{N} \rightsquigarrow \lambda\Lambda'.\, \mathrm{M}' \mathbin{\boldsymbol{|}} \mathrm{N}'}$$

$$\frac{\phi \vdash \Lambda \rightsquigarrow \Lambda' \quad \phi \vdash \mathrm{M} \rightsquigarrow \mathrm{M}'}{\phi \vdash \Lambda{\cdot}\mathrm{M} \rightsquigarrow \Lambda'{\cdot}\mathrm{M}'}$$

$$\frac{\begin{array}{l} {\sigma_1}_\chi = \tau_1 \quad \ldots \quad {\sigma_n}_\chi = \tau_n \\ \chi \vdash \mathcal{C}[v_1 \mathbin{⦂} \sigma_1, \ldots v_n \mathbin{⦂} \sigma_n] \rightsquigarrow \mathcal{C}'[v_1 \mathbin{⦂} \tau_1, \ldots v_n \mathbin{⦂} \tau_n] \\ 0 \vdash \mathcal{C}[\_{}_1 \mathbin{⦂} \sigma_1, \ldots \_{}_n \mathbin{⦂} \sigma_n] \quad 0 \vdash \mathcal{C}'[\_{}_1 \mathbin{⦂} \tau_1, \ldots \_{}_n \mathbin{⦂} \tau_n] \\ \phi \vdash \Lambda_1 \rightsquigarrow \Lambda_1' \quad \ldots \quad \phi \vdash \Lambda_n \rightsquigarrow \Lambda_n' \end{array}}{\begin{array}{l} \phi \vdash \langle\!\langle \mathcal{C}[\hat{}\Lambda_1 \mathbin{⦂} \sigma_1, \ldots \hat{}\Lambda_n \mathbin{⦂} \sigma_n] \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{C}'[\hat{}\Lambda_1' \mathbin{⦂} \tau_1, \ldots \hat{}\Lambda_n' \mathbin{⦂} \tau_n] \rangle\!\rangle \\ \quad \text{(where } v_1 \ \ldots \ v_n \text{ are fresh)} \end{array}}$$

Figure 17: Type-Match Relation

allow the types within quotations in the pattern to be more general than those of the argument. This allows functions on expressions to be defined by pattern matching, as in the following example:

$$\begin{aligned}\mathsf{let}\ len = {} & \lambda\langle\!\langle \mathsf{Len}\cdot([]\,\text{⸬}\,\alpha\ \mathit{list})\rangle\!\rangle.\ \langle\!\langle 0\rangle\!\rangle \\ \mid\ & \lambda\langle\!\langle \mathsf{Len}\cdot(\hat{}h{::}\hat{}t)\rangle\!\rangle.\ \langle\!\langle(\mathsf{Len}\cdot\hat{}t)+1\rangle\!\rangle\end{aligned}$$

We would expect to be able to apply the first λ-abstraction in this function to expressions such as $\langle\!\langle \mathsf{Len}\cdot([]\,\text{⸬}\,\mathit{int\ list})\rangle\!\rangle$, and so the pattern $\langle\!\langle \mathsf{Len}\cdot([]\,\text{⸬}\,\alpha\ \mathit{list})\rangle\!\rangle$ must match up to some instantiation of type variables.

6.3.2 Discarding an Alternative

The rules for β and γ-reduction require the argument expression to be ready to match the pattern before a match is made. Similarly, the rule for ζ-reduction requires the argument expression to be ready to match the pattern before the match is rejected.

In general, we may have $(\lambda\Lambda.\,\mathrm{M} \mid \mathrm{N})\cdot\Xi$, where Ξ has type *term* but has not yet been evaluated to yield a quotation. It is not possible to tell if and how Ξ might match the pattern Λ until Ξ has been evaluated. The assumption Λ ***ready*** M on the ζ-reduction rule prevents a match from being discarded too early. If an expression M is ready to match a pattern Λ, but there is no substitution θ such that (Λ,θ) ***matches*** M, then we may safely conclude that the expression doesn't match the pattern and discard this alternative.

In some circumstances a pattern will never match an expression and yet may also not be discarded. Consider the following application:

$$(\lambda\langle\!\langle \hat{}f\cdot\hat{}x\rangle\!\rangle.\,\Lambda \mid \mathrm{M})\cdot\langle\!\langle \hat{}\langle\!\langle inc\,\text{⸬}\,int\to int\rangle\!\rangle\,\text{⸬}\,\alpha\to\alpha\ \cdot\hat{}\langle\!\langle \mathsf{T}\rangle\!\rangle\,\text{⸬}\,\alpha\rangle\!\rangle$$

In this example the argument is not ready to match the pattern, however it may not be further reduced. The *reFLect* language does not let us conclude anything about the internal structure of expressions that are not sufficiently evaluated to tell if they are well typed. In an implementation, the inability to apply either the γ-reduction or ζ-reduction rules would result in the argument being forced to point where ψ-reduction was attempted and a run-time type error raised.

7 Compiling to λ-Calculus

Given data-structures for lists, expressions and contexts—and two functions, fill and match for manipulating them—the special features of *reFLect* (quotation, antiquotation, pattern matching) can be compiled away to produce ordinary λ-calculus.

$$
\begin{aligned}
&\mathsf{E}\ \text{“}v ⦂ \tau\text{”} = v\\
&\mathsf{E}\ \text{“}k ⦂ \tau\text{”} = k\\
&\mathsf{E}\ \text{“}\Lambda{\cdot}\mathrm{M}\text{”} = (\mathsf{E}\ \text{“}\Lambda\text{”})\ (\mathsf{E}\ \text{“}\mathrm{M}\text{”})\\
&\mathsf{E}\ \text{“}\lambda\Lambda.\,\mathrm{M}\text{”} = \mathsf{P}\ \text{“}\Lambda\text{”}\ (\mathsf{E}\ \text{“}\mathrm{M}\text{”})\ \mathsf{error}\\
&\mathsf{E}\ \text{“}\lambda\Lambda.\,\mathrm{M}\ \mathbf{|}\ \Xi\text{”} = \mathsf{P}\ \text{“}\Lambda\text{”}\ (\mathsf{E}\ \text{“}\mathrm{M}\text{”})\ (\mathsf{E}\ \text{“}\Xi\text{”})\\
&\mathsf{E}\ \text{“}\langle\!\langle \mathcal{C}[\hat{}\Lambda_1 ⦂ \tau_1, \ldots \hat{}\Lambda_n ⦂ \tau_n]\rangle\!\rangle\text{”} = \mathsf{fill}\ \text{‘}\langle\!\langle \mathcal{C}[\_ ⦂ \tau_1, \ldots \_ ⦂ \tau_n]\rangle\!\rangle\text{’}\ [\mathsf{E}\ \text{“}\Lambda_1\text{”}, \ldots \mathsf{E}\ \text{“}\Lambda_n\text{”}]\\
&\qquad\text{(where } 0 \vdash \mathcal{C}[\_ ⦂ \tau_1, \ldots \_ ⦂ \tau_n])
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{P}\ \text{“}v ⦂ \tau\text{”}\ \mathrm{M}\ \mathrm{N} = \lambda v.\,\mathrm{M}\\
&\mathsf{P}\ \text{“}\langle\!\langle \mathcal{C}[\hat{}(v_1 ⦂ \mathit{term}) ⦂ \tau_1, \ldots \hat{}(v_n ⦂ \mathit{term}) ⦂ \tau_n]\rangle\!\rangle\text{”}\ \mathrm{M}\ \mathrm{N} =\\
&\quad \mathsf{match}\ \text{‘}\langle\!\langle \mathcal{C}[\_ ⦂ \tau_1, \ldots \_ ⦂ \tau_n]\rangle\!\rangle\text{’}\ (\lambda w.\,\mathsf{L}\ [\text{“}v_1 ⦂ \mathit{term}\text{”}, \ldots \text{“}v_n ⦂ \mathit{term}\text{”}]\ \mathrm{M}\ w)\ \mathrm{N}\\
&\qquad\text{(where } 0 \vdash \mathcal{C}[\_ ⦂ \tau_1, \ldots \_ ⦂ \tau_n],\ w \text{ is fresh, and } v_1, \ldots\ \ v_n \text{ are distinct)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{L}\ []\ \mathrm{M}\ x = \mathrm{M}\\
&\mathsf{L}\ [\text{“}v ⦂ \tau\text{”}]\ \mathrm{M}\ x = (\lambda v.\,\mathrm{M})\ (\mathsf{hd}\ x)\\
&\mathsf{L}\ (\text{“}v ⦂ \tau\text{”} :: \mathit{vs})\ \mathrm{M}\ x = (\lambda v.\,\mathsf{L}\ \mathit{vs}\ \mathrm{M}\ (\mathsf{tl}\ x))\ (\mathsf{hd}\ x)
\end{aligned}
$$

Figure 18: Compilation to λ-calculus

Expressions and contexts can be represented as ordinary algebraic data-types. The function fill and match can then be defined on those data-types. The data-types for expressions and contexts must be as described in section 3. This section describes the required behavior of fill and match, and how those functions can then be used to implement the special features of *reFLect*.

The implementation of *reFLect* used at Intel follows the technique described here, except that the types of expressions and contexts are not implemented as ordinary algebraic data-types. Rather, the representation of *reFLect* syntax trees used by the underlying compiler is reused for these types. As a result, once an expression has been constructed it may be evaluated directly. The functions fill and match are implemented as primitives.

We write values of the type *term*—representing *reFLect* expressions—inside double quotes, “like this”. Similarly, we will write values of type *context*—representing *reFLect* contexts—inside single quotes, ‘like this’. We do this to distinguish values of these types from each other and from the surrounding λ-calculus expressions, with which they share much common syntax.

We describe the compilation process with three recursive functions: E (for compiling expressions), P (for compiling pattern abstractions), L (for compiling abstractions over lists lists of variables). The definitions of these functions are in Figure 18. Together these functions can compile expressions in *reFLect* to λ-calculus that uses the functions fill and match for constructing and destructing expressions. These functions are explained in Sections 7.1 and 7.2. The generated code also uses a constant value error to signal the outcome of a pattern matching failure.[4]

It must also be noted that because *reFLect* distinguishes variables by name and type, while ordinary λ-calculus distinguishes variables solely by name, we must avoid inadvertent variable capture by first α-converting any *reFLect* program to an equivalent one in which distinct variables at level zero have distinct names before compiling with the method described here.

7.1 Constructing Terms

Values of the *term* and *context* types are ground expressions and therefore not reducible. This means that the *reFLect* program $\lambda x.\, \lambda y.\, \langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle$ is not represented directly in the λ-calculus by the term "$\lambda x.\, \lambda y.\, \langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle$". To achieve the effect of this program we assume a function fill that takes a context with n holes and a list of n expressions, and forms a new expression by applying the minimal type instantiation to the context that makes the type of each hole agree with the type of the corresponding expression in the list, and then replaces each hole with the corresponding expression from the list (with surrounding quotations removed). The function fails if the list does not have the same number of the expressions as there are holes in the context or if a type instantiation cannot be found that brings the type of each hole in the context into agreement with the type of the corresponding expression in the list. The fill function is described below.[5]

$$\mathsf{fill}\colon \mathit{context} \rightarrow \mathit{term\ list} \rightarrow \mathit{term}$$

$$\frac{\vdash \Lambda_1 \colon \sigma_{1\phi} \quad \ldots \quad \vdash \Lambda_n \colon \sigma_{n\phi} \quad \mathsf{dom}\,\phi \subseteq \mathsf{vars}\{\sigma_1, \ldots \sigma_n\}}{\vdash \mathsf{fill}\ `\langle\!\langle \mathcal{C}[\_ \mathbin{\raisebox{0pt}{$\circ\!\!\!\circ$}} \sigma_1, \ldots \_ \mathbin{\circ} \sigma_n] \rangle\!\rangle\text{'}\ [\text{``}\langle\!\langle \Lambda_1 \rangle\!\rangle\text{''}, \ldots \text{``}\langle\!\langle \Lambda_n \rangle\!\rangle\text{''}] \xrightarrow{\lambda} \text{``}\langle\!\langle \mathcal{C}_\phi[\Lambda_1, \ldots \Lambda_n] \rangle\!\rangle\text{''}}$$

The *reFLect* program $\langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle$ can now be translated into the λ-calculus expression fill '$\langle\!\langle \_ + \_ \rangle\!\rangle$' $[x, y]$.

7.2 Destructing Terms

For the compilation of quotation patterns we will require another built-in function called match. The first argument to match is a context with n holes that will serve

[4]The compiler requires a slightly stricter definition of valid patterns, in that it is applicable only to linear patters, i.e., those in which no variable is repeated.

[5]We use $\xrightarrow{\lambda}$ to indicate a reduction in the λ-calculus as opposed to *reFLect*.

as a pattern. The second argument is a function that takes a list of n expressions as its argument. The third argument is a function from an expression to the same return type as the second argument. The fourth argument is an expression. It then attempts to match the expression to the context, producing a list of expressions for the regions that were matched to the holes. If the match was successful then the result is the application of the second argument to this list. If the match is not successful then the third argument is applied to the expression instead.

match:
$\quad \mathit{context} \to (\mathit{term\ list} \to \alpha) \to (\mathit{term} \to \alpha) \to \mathit{term} \to \alpha$

$$\frac{(\langle\!\langle \mathcal{C}[{}^{\wedge}(v_1 {:} \mathit{term}) {:} \sigma_1, \ldots {}^{\wedge}(v_n {:} \mathit{term}) {:} \sigma_n] \rangle\!\rangle, [\langle\!\langle \Xi_1 \rangle\!\rangle, \ldots \langle\!\langle \Xi_n \rangle\!\rangle / v_1, \ldots v_n]) \ \textit{matches}\ \langle\!\langle \mathrm{N} \rangle\!\rangle}{\vdash \mathsf{match}\ `\langle\!\langle \mathcal{C}[\_ {:} \sigma_1, \ldots \_ {:} \sigma_n] \rangle\!\rangle\text{'}\ \Lambda\ \mathrm{M}\ ``\langle\!\langle \mathrm{N} \rangle\!\rangle\text{''} \xrightarrow{\lambda} \Lambda\ [``\langle\!\langle \Xi_1 \rangle\!\rangle\text{''}, \ldots ``\langle\!\langle \Xi_n \rangle\!\rangle\text{''}]}$$
(where $v_1, \ldots\ v_n$ are fresh)

$$\frac{\nexists \theta.\, (\langle\!\langle \mathcal{C}[{}^{\wedge}v_1 {:} \sigma_1, \ldots {}^{\wedge}v_n {:} \sigma_n] \rangle\!\rangle, \theta)\ \textit{matches}\ \langle\!\langle \mathrm{N} \rangle\!\rangle}{\vdash \mathsf{match}\ `\langle\!\langle \mathcal{C}[\_ {:} \sigma_1, \ldots \_ {:} \sigma_n] \rangle\!\rangle\text{'}\ \Lambda\ \mathrm{M}\ ``\langle\!\langle \mathrm{N} \rangle\!\rangle\text{''} \xrightarrow{\lambda} \mathrm{M}\ ``\langle\!\langle \mathrm{N} \rangle\!\rangle\text{''}}$$
(where $v_1, \ldots\ v_n$ are fresh)

Using match the *reFL^ect* program $\lambda\langle\!\langle {}^{\wedge}x + {}^{\wedge}y \rangle\!\rangle.\, (x, y)$ can be translated into λ-calculus as follows:

$$\mathsf{match}\ `\langle\!\langle \_ + \_ \rangle\!\rangle\text{'}\ (\lambda[x, y].\, (x, y))\ \mathsf{error}$$

7.3 Compilation Example

We illustrate the action of the compiler on the definition of the *comm* function from section 2.

$$\begin{aligned}
\mathsf{E}\ ``&\lambda\langle\!\langle {}^{\wedge}x + {}^{\wedge}y \rangle\!\rangle.\, \langle\!\langle {}^{\wedge}(\mathit{comm}{\cdot}y) + {}^{\wedge}(\mathit{comm}{\cdot}x) \rangle\!\rangle \\
&|\, \lambda\langle\!\langle {}^{\wedge}f {\cdot} {}^{\wedge}x \rangle\!\rangle.\, \langle\!\langle {}^{\wedge}(\mathit{comm}{\cdot}f) {\cdot} {}^{\wedge}(\mathit{comm}{\cdot}x) \rangle\!\rangle \\
&|\, \lambda\langle\!\langle \lambda {}^{\wedge}p.\, {}^{\wedge}b \rangle\!\rangle.\, \langle\!\langle \lambda {}^{\wedge}p.\, {}^{\wedge}(\mathit{comm}{\cdot}b) \rangle\!\rangle \\
&|\, \lambda\langle\!\langle {}^{\wedge}p.\, {}^{\wedge}b \mid {}^{\wedge}a \rangle\!\rangle.\, \langle\!\langle \lambda {}^{\wedge}p.\, {}^{\wedge}(\mathit{comm}{\cdot}b) \mid {}^{\wedge}(\mathit{comm}{\cdot}a) \rangle\!\rangle \\
&|\, \lambda x.\, x\text{''}
\end{aligned}$$

We begin by repeatedly invoking the expression compiler E. In this first step we have invoked E on any instances of the fourth and fifth clauses of its definition. These clauses translate *reFL^ect* pattern matching λs into calls to the pattern compiler P.

$$\begin{aligned}
&\mathsf{P}\ ``\langle\!\langle {}^{\wedge}x + {}^{\wedge}y \rangle\!\rangle\text{''}\ (\mathsf{E}\ ``\langle\!\langle {}^{\wedge}(\mathit{comm}{\cdot}y) + {}^{\wedge}(\mathit{comm}{\cdot}x) \rangle\!\rangle\text{''}) \\
&(\mathsf{P}\ ``\langle\!\langle {}^{\wedge}f {\cdot} {}^{\wedge}x \rangle\!\rangle\text{''}\ (\mathsf{E}\ ``\langle\!\langle {}^{\wedge}(\mathit{comm}{\cdot}f) {\cdot} {}^{\wedge}(\mathit{comm}{\cdot}x) \rangle\!\rangle\text{''}) \\
&(\mathsf{P}\ ``\langle\!\langle \lambda {}^{\wedge}p.\, {}^{\wedge}b \rangle\!\rangle\text{''}(\mathsf{E}\ ``\langle\!\langle \lambda {}^{\wedge}p.\, {}^{\wedge}(\mathit{comm}{\cdot}b) \rangle\!\rangle\text{''}) \\
&(\mathsf{P}\ ``\langle\!\langle \lambda {}^{\wedge}p.\, {}^{\wedge}b \mid {}^{\wedge}a \rangle\!\rangle\text{''}(\mathsf{E}\ ``\langle\!\langle \lambda {}^{\wedge}p.\, {}^{\wedge}(\mathit{comm}{\cdot}b) \mid {}^{\wedge}(\mathit{comm}{\cdot}a) \rangle\!\rangle\text{''}) \\
&(\mathsf{P}\ ``x\text{''}\ (\mathsf{E}\ ``x\text{''})\ \mathsf{error}))))
\end{aligned}$$

Next we use E again, this time applying it to any instances of the sixth clause of its definition. This translates the use of antiquotation to perform expression construction into an application of the fill function.

$$\begin{array}{l}
\ \mathsf{P}\ \text{“}\langle\langle {}^{\wedge}x + {}^{\wedge}y\rangle\rangle\text{”}\ (\mathsf{fill}\ \text{‘}\langle\langle \_ + \_\rangle\rangle\text{’}\ [\mathsf{E}\ \text{“}\mathit{comm}{\cdot}y\text{”}, \mathsf{E}\ \text{“}\mathit{comm}{\cdot}x\text{”}])\\
(\mathsf{P}\ \text{“}\langle\langle {}^{\wedge}f{\cdot}{}^{\wedge}x\rangle\rangle\text{”}\ (\mathsf{fill}\ \text{‘}\langle\langle \_{\cdot}\_\rangle\rangle\text{’}\ [\mathsf{E}\ \text{“}\mathit{comm}{\cdot}f\text{”}, \mathsf{E}\ \text{“}\mathit{comm}{\cdot}x\text{”}])\\
(\mathsf{P}\ \text{“}\langle\langle \lambda{}^{\wedge}p.\,{}^{\wedge}b\rangle\rangle\text{”}(\mathsf{fill}\ \text{‘}\langle\langle \lambda\_.\,\_\rangle\rangle\text{’}\ [\mathsf{E}\ \text{“}p\text{”}, \mathsf{E}\ \text{“}\mathit{comm}{\cdot}b\text{”}])\\
(\mathsf{P}\ \text{“}\langle\langle \lambda{}^{\wedge}p.\,{}^{\wedge}b \mid {}^{\wedge}a\rangle\rangle\text{”}(\mathsf{fill}\ \text{‘}\langle\langle \lambda\_.\,\_ \mid \_\rangle\rangle\text{’}\ [\mathsf{E}\ \text{“}p\text{”}, \mathsf{E}\ \text{“}\mathit{comm}{\cdot}b\text{”}, \mathsf{E}\ \text{“}\mathit{comm}{\cdot}a\text{”}])\\
(\mathsf{P}\ \text{“}x\text{”}\ x\ \mathsf{error}))))
\end{array}$$

A few more applications of E, this time focusing on instances of the first three clauses of its definition, remove the remaining uses of this function. In doing so we complete the translation of the bodies of the original *reFLect* abstractions.

$$\begin{array}{l}
\ \mathsf{P}\ \text{“}\langle\langle {}^{\wedge}x + {}^{\wedge}y\rangle\rangle\text{”}\ (\mathsf{fill}\ \text{‘}\langle\langle \_ + \_\rangle\rangle\text{’}\ [\mathit{comm}\ y, \mathit{comm}\ x])\\
(\mathsf{P}\ \text{“}\langle\langle {}^{\wedge}f{\cdot}{}^{\wedge}x\rangle\rangle\text{”}\ (\mathsf{fill}\ \text{‘}\langle\langle \_{\cdot}\_\rangle\rangle\text{’}\ [\mathit{comm}\ f, \mathit{comm}\ x])\\
(\mathsf{P}\ \text{“}\langle\langle \lambda{}^{\wedge}p.\,{}^{\wedge}b\rangle\rangle\text{”}(\mathsf{fill}\ \text{‘}\langle\langle \lambda\_.\,\_\rangle\rangle\text{’}\ [p, \mathit{comm}\ b])\\
(\mathsf{P}\ \text{“}\langle\langle \lambda{}^{\wedge}p.\,{}^{\wedge}b \mid {}^{\wedge}a\rangle\rangle\text{”}(\mathsf{fill}\ \text{‘}\langle\langle \lambda\_.\,\_ \mid \_\rangle\rangle\text{’}\ [p, \mathit{comm}\ b, \mathit{comm}\ a])\\
(\mathsf{P}\ \text{“}x\text{”}\ x\ \mathsf{error}))))
\end{array}$$

We now use P to compile the pattern matching code into an application of the match function.

$$\begin{array}{l}
\ \mathsf{match}\ \text{‘}\langle\langle \_ + \_\rangle\rangle\text{’}\ (\lambda k.\,\mathsf{L}\ [\text{“}x\text{”}, \text{“}y\text{”}]\ (\mathsf{fill}\ \text{‘}\langle\langle \_ + \_\rangle\rangle\text{’}\ [\mathit{comm}\ y, \mathit{comm}\ x])\ k)\\
(\mathsf{match}\ \text{‘}\langle\langle \_{\cdot}\_\rangle\rangle\text{’}\ (\lambda l.\,\mathsf{L}\ [\text{“}f\text{”}, \text{“}x\text{”}]\ (\mathsf{fill}\ \text{‘}\langle\langle \_{\cdot}\_\rangle\rangle\text{’}\ [\mathit{comm}\ f, \mathit{comm}\ x])\ l)\\
(\mathsf{match}\ \text{‘}\langle\langle \lambda\_.\,\_\rangle\rangle\text{’}(\lambda m.\,\mathsf{L}\ [\text{“}p\text{”}, \text{“}b\text{”}]\ (\mathsf{fill}\ \text{‘}\langle\langle \lambda\_.\,\_\rangle\rangle\text{’}\ [p, \mathit{comm}\ b])\ m)\\
(\mathsf{match}\ \text{“}\langle\langle \lambda\_.\,\_ \mid \_\rangle\rangle\text{”}(\lambda n.\,\mathsf{L}\ [\text{“}p\text{”}, \text{“}b\text{”}, \text{“}a\text{”}]\ (\mathsf{fill}\ \text{‘}\langle\langle \lambda\_.\,\_ \mid \_\rangle\rangle\text{’}\ [p, \mathit{comm}\ b, \mathit{comm}\ a])\ n)\\
(\lambda x.\,x))))
\end{array}$$

We complete the compilation with repeated application of the variable list abstraction compiler L.

$$\begin{array}{l}
\ \mathsf{match}\ \text{‘}\langle\langle \_ + \_\rangle\rangle\text{’}\ (\lambda k.\,(\lambda x.\,(\lambda y.\,\mathsf{fill}\ \text{‘}\langle\langle \_ + \_\rangle\rangle\text{’}\ [\mathit{comm}\ y, \mathit{comm}\ x])\ (\mathsf{hd}\ (\mathsf{tl}\ k)))\ (\mathsf{hd}\ k))\\
(\mathsf{match}\ \text{‘}\langle\langle \_{\cdot}\_\rangle\rangle\text{’}\ (\lambda l.\,(\lambda f.\,(\lambda x.\,\mathsf{fill}\ \text{‘}\langle\langle \_{\cdot}\_\rangle\rangle\text{’}\ [\mathit{comm}\ f, \mathit{comm}\ x])\ (\mathsf{hd}\ (\mathsf{tl}\ l)))\ (\mathsf{hd}\ l))\\
(\mathsf{match}\ \text{‘}\langle\langle \lambda\_.\,\_\rangle\rangle\text{’}(\lambda m.\,(\lambda p.\,(\lambda b.\,\mathsf{fill}\ \text{‘}\langle\langle \lambda\_.\,\_\rangle\rangle\text{’}\ [p, \mathit{comm}\ b])\ (\mathsf{hd}\ (\mathsf{tl}\ m)))\ (\mathsf{hd}\ m))\\
(\mathsf{match}\ \text{“}\langle\langle \lambda\_.\,\_ \mid \_\rangle\rangle\text{”}(\lambda n.\,(\lambda p.\,(\lambda b.\,(\lambda a.\,\mathsf{fill}\ \text{‘}\langle\langle \lambda\_.\,\_ \mid \_\rangle\rangle\text{’}\ [p, \mathit{comm}\ b, \mathit{comm}\ a])\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{hd}\ (\mathsf{tl}\ (\mathsf{tl}\ n))))\ (\mathsf{hd}\ (\mathsf{tl}\ n)))\ (\mathsf{hd}\ n))\\
(\lambda x.\,x))))
\end{array}$$

8 Reflection

Thus far we have described the core of the *reFLect* language. This language features facilities for constructing and destructing expressions using quotation, antiquotation and pattern matching. These allow *reFLect* to be used for applications, like theorem prover development, that might usually be approached with a system based on a separate meta-language and object-language. In this section we add some facilities for reflection.

8.1 Evaluation

The *reFLect* language has two built-in functions for evaluation of expressions: eval and value.[6] Suppose we use the notation $\Lambda \Rightarrow \Lambda'$ to mean that Λ is evaluated to produce Λ'. Certainly $\Lambda \Rightarrow \Lambda'$ implies $\Lambda \xrightarrow{*} \Lambda'$, but we consider the order of evaluation and the normal form at which evaluation stops to be implementation specific, and so we leave these unspecified. The eval function is then described as follows:

$$\vdash \mathsf{eval}\colon \mathit{term} \to \mathit{term}$$

$$\frac{0 \vdash \Lambda \qquad \vdash \Lambda \Rightarrow \Lambda'}{\vdash \mathsf{eval}\cdot\langle\!\langle \Lambda \rangle\!\rangle \to \langle\!\langle \Lambda' \rangle\!\rangle}$$

Next we consider value. It is a slight misstatement to say that value is a function in *reFLect*—rather there is an infinite family of functions value_σ indexed by type. The behavior of value is similar to that of antiquotation; it removes the quotes from around an expression and interprets the result as a value of the appropriate type.

$$\vdash \mathsf{value}_\sigma\colon \mathit{term} \to \sigma$$

$$\frac{0 \vdash \Lambda \qquad \mathit{free}\,\Lambda = \emptyset \qquad \vdash \Lambda\colon \tau \qquad \tau_\phi = \sigma}{\vdash \mathsf{value}_\sigma\cdot\langle\!\langle \Lambda \rangle\!\rangle \to \Lambda\phi}$$

There are several important differences between value and antiquotation:

- The value function may appear at level zero, while antiquotation may not.
- Like other functions, value has no effect when quoted, while a (once) quoted antiquote may be reduced.
- If the type required of an expression is different from the actual type, then value may instantiate the type of the expression. Antiquotation may instead instantiate the type of its context.
- Antiquotation does not alter the level of the quoted expression, but value moves the body of the quoted expression from level one to level zero.

[6]Note that of the two, it is value rather than eval that most closely corresponds to the `eval` operation in LISP.

This last difference has important consequences for the treatment of variable binding. In moving an expression to level zero, value could expose its free variables to capture by enclosing lambda bindings. We restrict value to operate on closed expressions to prevent this. The restriction is similar in motivation to the run-time variable check of run in MetaML [30] or the static check for closed code in the system λ^{BN} [3].

8.2 Value Reification

The *reFLect* language also supports a partial inverse of evaluation through the lift function; its purpose is to make quoted representations of values. For example, $\mathsf{lift}{\cdot}1$ is $\langle\!\langle 1\rangle\!\rangle$ and $\mathsf{lift}{\cdot}\mathsf{T}$ is $\langle\!\langle \mathsf{T}\rangle\!\rangle$. The function lift is strict, so $\mathsf{lift}{\cdot}(1+2)$ is equal to $\langle\!\langle 3\rangle\!\rangle$. Note also that lift may only be applied to closed expressions. Lifting quotations is easy: just wrap another quote around them. For example, $\mathsf{lift}{\cdot}\langle\!\langle x+y\rangle\!\rangle$ gives $\langle\!\langle\langle\!\langle x+y\rangle\!\rangle\rangle\!\rangle$. Lifting recursive data-structures follows a recursive pattern that can be seen from the following example of how lift works on lists.

$$\begin{array}{l}
\mathsf{lift}{\cdot}[]\mathbin{\circ\!\!\!\circ}\,\sigma\ \mathit{list} = \langle\!\langle []\mathbin{\circ\!\!\!\circ}\,\sigma\ \mathit{list}\rangle\!\rangle \\
\mathsf{lift}{\cdot}(::\mathbin{\circ\!\!\!\circ}\,\sigma \to \sigma\ \mathit{list} \to \sigma \mathit{list}){\cdot}\Lambda{\cdot}\mathrm{M}) = \\
\quad \langle\!\langle (::\mathbin{\circ\!\!\!\circ}\,\sigma \to \sigma\ \mathit{list} \to \sigma \mathit{list}){\cdot}\hat{}(\mathsf{lift}{\cdot}\Lambda)\mathbin{\circ\!\!\!\circ}\,\sigma{\cdot}\hat{}(\mathsf{lift}{\cdot}\mathrm{M})\mathbin{\circ\!\!\!\circ}\,\sigma\ \mathit{list}\rangle\!\rangle
\end{array}$$

Lifting numbers, booleans and recursive data-structures is easy because they have a canonical form, but the same is not true of other data-types. For example, how do we lift $\lambda x.\,x+1$? Naively wrapping quotations around the expressions would result in inconsistencies. For example, $\lambda x.\,x+1$ and $\lambda x.\,1+x$ are equal and extensionality therefore requires that $\mathsf{lift}{\cdot}(\lambda x.\,x+1)$ and $\mathsf{lift}{\cdot}(\lambda x.\,1+x)$ be equal, but $\langle\!\langle \lambda x.\,x+1\rangle\!\rangle$ and $\langle\!\langle \lambda x.\,1+x\rangle\!\rangle$ are not equal. If Λ is an expression of some type σ without a canonical form then we will use the following definition for lift.

$$\mathsf{lift}{\cdot}\Lambda = \langle\!\langle [\![\Lambda]\!]\mathbin{\circ\!\!\!\circ}\,\sigma\rangle\!\rangle$$

You should think of $[\![\Lambda]\!]$ as being a new and unusual constant name. These names have the property that if Λ and M are semantically equal, then $[\![\Lambda]\!]$ and $[\![\mathrm{M}]\!]$ are considered the same name. For example evaluating $\mathsf{lift}{\cdot}(\lambda x.\,x+1)$ produces $\langle\!\langle [\![\lambda x.\,x+1]\!]\rangle\!\rangle$ and evaluating $\mathsf{lift}{\cdot}(\lambda x.\,1+x)$ produces $\langle\!\langle [\![\lambda x.\,1+x]\!]\rangle\!\rangle$, and the two resulting expressions are equal since they are both quoted constants with 'equal' names.[7] When we do this, we say that we have put the expression in a *black box.* Since black boxes are just a kind of constant they require no further special treatment.

Note that eval is not simply the composition of value and lift. Consider the expressions $\langle\!\langle(\lambda x.\,\lambda y.\,x+y){\cdot}1)\rangle\!\rangle$ and $\langle\!\langle(\lambda x.\,\lambda y.\,y+x){\cdot}1)\rangle\!\rangle$. Applying eval to these

[7]Of course, the equality of such names is not decidable.

expressions produces $\langle\!\langle \lambda y.\, 1 + y \rangle\!\rangle$ and $\langle\!\langle \lambda y.\, y + 1 \rangle\!\rangle$. Applying value then lift however must yield $\langle\!\langle [\![\lambda y.\, 1 + y]\!] \rangle\!\rangle$ and $\langle\!\langle [\![\lambda y.\, y + 1]\!] \rangle\!\rangle$. Because the composition of value and lift takes an expression to an expression via the unquoted form that represents its meaning, two different expressions that represent semantically equal programs must produce the same result. By taking expressions to expressions directly the eval operation is not so constrained.

As an example, by using lift you can write the function sum defined by

$$\mathsf{letrec}\ \mathit{sum} = \lambda n.\, \mathsf{if}\ n = 0\ \mathsf{then}\ \langle\!\langle 0 \rangle\!\rangle\ \mathsf{else}\ \langle\!\langle \hat{}(\mathsf{lift}\cdot n) + \hat{}(\mathit{sum}\cdot(n-1)) \rangle\!\rangle$$

This maps n to an expression that sums all the numbers up to n. For example, $\mathsf{sum}\cdot 4$ produces $\langle\!\langle 4 + 3 + 2 + 1 + 0 \rangle\!\rangle$.

This feature addresses a shortcoming of the previous version of Forte based on *FL* . Users of this system sometimes want to verify a result by case analysis that can involve decomposing a goal into hundreds of similar cases, each of which is within reach of an automatic solver. It is difficult in *FL* to write a function that will produce (a conjunction of) all those cases. Facilities like lift make this easier, and the code that does it more transparent.

8.3 Reflection and Compilation

In an implementation based on compilation the reflection features just presented are complicated by the fact that type information is lost during the compilation process. To compensate for this the lift operation must reconstruct types as it processes an expression. In some cases the quoted expression returned will have a more general type than the original value because non-inferable type information will have been lost.

In such a system the eval and value functions would be implemented as follows.

$$\frac{\mathit{free}\,\Lambda = \emptyset \quad \vdash \Lambda\!: \sigma \quad \vdash \mathsf{lift}\cdot(\mathsf{E}\ \text{“}\Lambda\text{”}) \xrightarrow{\lambda} \text{“}\langle\!\langle \mathrm{M} \rangle\!\rangle\text{”} \quad \vdash \mathrm{M}\!: \tau \quad \tau_\theta = \sigma}{\vdash \mathsf{eval}\ \text{“}\langle\!\langle \Lambda \rangle\!\rangle\text{”} \xrightarrow{\lambda} \text{“}\langle\!\langle \mathrm{M}\theta \rangle\!\rangle\text{”}}$$

$$\frac{\mathit{free}\,\Lambda = \emptyset \quad \vdash \Lambda\!: \tau \quad \tau_\phi = \sigma}{\vdash \mathsf{value}_\sigma\ \text{“}\langle\!\langle \Lambda \rangle\!\rangle\text{”} \xrightarrow{\lambda} (\mathsf{E}\ \text{“}\Lambda\text{”})}$$

Note that the eval function here is less widely applicable than one based directly on the operational semantics. It may also return results with subexpressions at more general types than one based on the operational semantics. Using lift in the implementation of eval may also result in subexpressions that are black boxes, as described above. However, we may relax the behavior of lift in this case so that it does not produce black boxes as it poses no logical problem for eval to take syntactically different input terms to different output terms.

9 Related Work

The *reFL*^{ect} language can been seen as an application-specific contribution to the field of *meta-programming.* In Tim Sheard's taxonomy of meta-programming [24], *reFL*^{ect} is a framework for both generating and analyzing programs; it includes features for run-time program generation; and it is typed, 'manually staged', and 'homogeneous'. Our design decisions, however, were driven by the needs of our target applications: symbolic reasoning in higher-order logic, hardware modeling, and hardware transformation. So the 'analysis' aspect is much more important than for the design of functional meta-programming languages aimed at optimized program execution.

Nonetheless, *reFL*^{ect} has a family resemblance to languages for run-time code generation such as MetaML [30] and Template Haskell [25]. A distinguishing feature of MetaML is *cross-stage persistence*, in which a variable binding applies across the quotation boundary. The motivation is to allow programmers to take advantage of bindings made in one stage at all future stages. In *reFL*^{ect}, however, we wish to define a *logic* on top of the language and so we take the conventional logical view of quotation and binding. Variable bindings do not persist across levels. Constant definitions, however, are available in all levels. They therefore provide a limited and safe form of 'cross-level' persistence, just as they do with polymorphism.

For reasons already described, *reFL*^{ect} also differs from MetaML in typing all quotations with a universal type *term*. Template Haskell is similar to *reFL*^{ect} in this respect. One of the 'advertised goals' of Template Haskell is also to support user-defined code manipulation or optimization, though probably not logic.

Perhaps the closest framework to *reFL*^{ect} is the system described by Shields et al. in [26]. This has a universal term type, a splicing rule for quotation and antiquotation similar to our ψ rule, and run-time type checking of quoted regions. Our applications in theorem proving and design transformation have, however, led to some key differences. We adopt a simpler notion of type-consistency when splicing expressions into a context, ensuring only that the resulting expression is well typed, while the Shields system ensures consistent typing of variables. This relaxation keeps the logic we construct from *reFL*^{ect} simple, and the implementation of time critical theorem proving algorithms, like rewriting, efficient.

The *reFL*^{ect} language extends the notion of quotation and antiquotation, which have been used for term construction since the LCF system [8], by also allowing these constructs to be used for term decomposition via pattern matching. In this respect we follow the work of Aasa, Petersson and Synek [2] who proposed this mechanism for constructing and destructing object-language expressions within a meta-language. The other reflective languages discussed here [25, 26, 30] do not support this form of pattern matching, which is valuable for our applications in code inspection and transformation, but would find less application in the applications targeted by these systems.

10 Conclusion

In this paper we presented the language *reFL*^{ect}; a functional language with strong typing, quotation and antiquotation features for meta-programming, and reflection. The quotation and antiquotation features can be used not only to construct expressions, but also to transparently implement functions that inspect or traverse expressions via pattern matching. We made novel use of contexts with a level consistency property to give concise descriptions of the type system and operational semantics of *reFL*^{ect}, as well as using them to describe a method of compiling away the new syntactic features of *reFL*^{ect}.

We have completed an implementation of *reFL*^{ect} using the compilation technique described to translate *reFL*^{ect} into λ-calculus, which is then evaluated using essentially the same combinator compiler and run-time system as the previous *FL* system [1]. The performance of *FL* programs that do not use the new features of *reFL*^{ect} has not been impacted.

We have used *reFL*^{ect} to implement a mechanized reasoning system based on inspirations from HOL [10] and the Forte [1, 13] system, a tool used extensively within Intel for hardware verification. The ability to pattern match on expressions has made the logical kernel of this system more transparent and compact than those of similar systems. The system includes evaluation as a deduction rule, and combines evaluation with rewriting to simplify closed subexpressions efficiently.

This presentation of the type system and operational semantics for *reFL*^{ect} gives a good starting point for investigation of more theoretical properties of the language, like confluence, subject-reduction, and normalization. Sava Krstić and John Matthews of the Oregon Graduate Institute have proved these properties for the *reFL*^{ect} language features for expression construction and analysis, though not those that relate to evaluation of expressions [17]. Their proofs cover the language presented here up to, but not including, section 7.

11 Acknowledgments

The authors would like to thank the following people for their helpful discussions on *reFL*^{ect}: Rob Gerth and Carl Seger of the Intel Corporation, Mike Gordon of the University of Cambridge, Sava Krstić and John Matthews of the Oregon Graduate Institute, and Mark Shields of Galois Connections.

References

[1] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. *Lifted-FL*: A pragmatic implementation of combined model checking and theorem proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *LNCS*, pages 323–340. Springer-Verlag, 1999.

[2] A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects. In *LISP and Functional Programming: ACM Conference, LFP 88*, pages 96–105. ACM Press, 1988.

[3] Z. E.-A. Benaissa, E. Moggi, W. Taha, and T. Sheard. Logical modalities and multi-stage programming. In *Intuitionsitic Modal Logics and Applications: Federated Logic Conference Satellite Workshop, IMLA*, 1999.

[4] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs: International Workshop, TYPES 2000*, volume 2277 of *LNCS*, pages 24–40. Springer-Verlag, 2000.

[5] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Functional Programming: International Conference, ICFP'98*, pages 174–184. ACM Press, 1998.

[6] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[7] T. Coquand. An analysis of Girard's paradox. In *Logic in Computer Science: 1st IEEE Symposium, LICS'86*, pages 227–236. IEEE Computer Society Press, 1986.

[8] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[9] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Workshop*, pages 153–177. North-Holland, 1985.

[10] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[11] R. Harper, D. MacQueen, and R. Milner. Standard ML. Report 86-2, University of Edinburgh, Laboratory for Foundations of Computer Science, 1986.

[12] S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, 1984.

[13] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham. Practical formal verification in microprocessor design. *IEEE Design & Test of Computers*, 18(4):16–25, 2001.

[14] R. Kaivola and K. R. Kohatsu. Proof engineering in the large: Formal verification of the Pentium® 4 floating-point divider. In T. Margaria and T. F. Melham, editors, *Correct Hardware Design and Verification Methods: 11th Advanced Research Working Conference, CHARME 2001*, volume 2144 of *LNCS*, pages 196–211. Springer-Verlag, 2001.

[15] R. Kaivola and N. Narasimhan. Formal verification of the Pentium® 4 multiplier. In *High-Level Design Validation and Test: 6th International Workshop: HLDVT 2001*, pages 115–122, 2001.

[16] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer, 2000.

[17] S. Krstić and J. Matthews. Subject reduction and confluence for the *reFL^{ect}* language. Technical Report CSE-03-014, Department of Computer Science and Engineering, OGI School of Science and Engineering at Oregon Health and Sciences University, 2003.

[18] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol.* Macdonald, 1969.

[19] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Computer Languages: International Conference*, pages 90–101. IEEE Computer Society Press, 1998.

[20] T. F. Melham. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.

[21] O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–651, 1997.

[22] L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149, 1983.

[23] E. Pašalić, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. *SIGPLAN Notices*, 37(9):218–229, 2002.

[24] T. Sheard. Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation: 2nd International Workshop, SAIG 2001*, volume 2196 of *LNCS*, pages 2–44. Springer-Verlag, 2001.

[25] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Haskell: Workshop*, pages 1–16. ACM Press, 2002.

[26] M. Shields, T. Sheard, and S. P. Jones. Dynamic typing as staged type inference. In *Principles of Programming Language: 25th Annual Symposium, POPL 1998*, pages 289–302, 1998.

[27] G. Spirakis. Leading-edge and future design challenges: Is the classical EDA ready? In *Design Automation: 40th ACM/IEEE Conference, DAC 2003*, page 416. ACM Press, 2003.

[28] P. C. Suppes. *Introduction to Logic*, chapter 6. Dover, 1999.

[29] W. Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[30] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Notices*, 32(12):203–217, 2002.