JFP **16** (2): 157–196, 2006. © 2005 Cambridge University Press
doi:10.1017/S0956796805005757 First published online 28 October 2005 Printed in the United Kingdom

A reflective functional language for hardware design and theorem proving

JIM GRUNDY
Intel Corporation, Strategic CAD Labs
Mail Stop JF4-211, 2111 NE 25th Ave, Hillsboro, OR 97124-5961, USA
(*e-mail:* `Jim.D.Grundy@intel.com`)

TOM MELHAM
Oxford University, Computing Laboratory
Wolfson Building, Parks Rd, Oxford, OX1 3QD, UK
(*e-mail:* `Tom.Melham@comlab.ox.ac.uk`)

JOHN O'LEARY
Intel Corporation, Strategic CAD Labs
Mail Stop JF4-211, 2111 NE 25th Ave, Hillsboro, OR 97124-5961, USA
(*e-mail:* `John.W.O'Leary@intel.com`)

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 novel use of contexts as expression constructors, and it presents a scheme for compiling *reFLect* programs 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 (Harper *et al.*, 1986) but has quotation and antiquotation constructs. These are 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* (Aagaard *et al.*, 1999) to large-scale verification problems at Intel (Jones *et al.*, 2001; Kaivola & Kohatsu, 2001; Kaivola & Narasimhan, 2001).

Hardware designs are modeled as *reFLect* programs. As with similar work based on Haskell (Bjesse *et al.*, 1998; Matthews *et al.*, 1998) or LISP (Johnson, 1984;

This paper is dedicated to the memory of our friend and colleague Rob Gerth.

Kaufmann *et al.*, 2000a), 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 (Spirakis, 2003). Moreover, we want the *reFLect* language to form the core of a typed higher-order logic for specifying and verifying hardware properties (Gordon, 1985; Melham, 1993), and simultaneously the implementation language of a theorem prover for this logic.

Formal reasoning about hardware is performed using the Forte tool (Jones *et al.*, 2001), which was originally designed around *FL* but now uses *reFLect*. Forte includes a theorem prover of similar design to the HOL system (Gordon & Melham, 1993). In higher-order logic theorem provers like HOL the logical 'object language' in which reasoning is done is embedded as a data-type in the (functional) meta-language used to control the reasoning. In HOL, the meta-language is ML. Representing object-language expressions as a data-type makes it straightforward to implement the various term analysis and transformation functions required by a theorem prover. But separating the object-language and meta-language causes duplication and inefficiency. Many theorem provers, for example, include special code for efficient execution of object-language expressions (Barras, 2000; Berghofer & Nipkow, 2000).

In *reFLect* 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.

In systems like HOL, higher-order logic is constructed along the lines of Church's formulation of simple type theory (Church, 1940), in which the logic is defined on top of the λ-calculus. Our theorem prover also follows this approach but constructs a variant of higher-order logic on top of the *reFLect* language, rather than the λ-calculus. The reduction rules for the language, which are given in this paper, are among the inference rules in our higher-order logic.

The applications just described give *intensional analysis* a primary role in *reFLect*. The design of our language is therefore different from staged functional languages like MetaML (Taha & Sheard, 2002) and Template Haskell (Sheard & Peyton Jones, 2002), which are aimed more at program generation and the control and optimization of evaluation. The *reFLect* 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 *reFLect* by presenting extensions to the λ-calculus that implement its key features. To motivate our design decisions, we first discuss the target applications for *reFLect*. We then present the syntax and type system, and give an operational semantics of evaluation. We conclude by presenting a scheme for compiling *reFLect* programs into the λ-calculus. This compilation scheme forms the basis of the *reFLect* implementation used at Intel. We include a number of propositions about the language to clarify our presentation. A more formal presentation of *reFLect*, including proofs of propositions of the kind cited here, is provided by Krstić and Matthews (2003).

The version of *reFLect* presented here is a simplification of the language as it is used within Intel. We have omitted from the presentation descriptions of standard functional programming language features like (recursive) constant definitions, definitions of algebraic data-types and mechanisms for pattern matching over them, exceptions and exception handling, etc. This allows us to focus on presenting the novel features of *reFLect* as extensions to the λ-calculus. The complete *reFLect* language augments the language presented here with standard functional programming concepts in the usual ways. An implementation of *reFLect* may be downloaded from Intel as part of the public release of the forte verification system.[1]

2 Motivation and overview

The *reFLect* 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 an abstract syntax tree. Hence, for example, while $1 + 2$ is semantically equal to 3, the quoted expression $\langle\!\langle 1 + 2 \rangle\!\rangle$ is semantically different from $\langle\!\langle 3 \rangle\!\rangle$. The expressions $1 + 2$ and 3 both denote the same integer value, namely 3. But the expression $\langle\!\langle 1 + 2 \rangle\!\rangle$ denotes the abstract syntax tree of '$1 + 2$', which is different from the abstract syntax tree of '3'.

There is also an antiquotation mechanism, written by prefixing an expression with '^', that splices one abstract syntax tree into another. For example in the quotation $\langle\!\langle 1 + \hat{}\langle\!\langle 2 \rangle\!\rangle \rangle\!\rangle$ the abstract syntax for '2' is spliced into the position at which the right-hand operand of the infix $+$ operator occurs. Using the reduction system presented later, this quotation evaluates to $\langle\!\langle 1 + 2 \rangle\!\rangle$.

Quotation and antiquotation may also be used for pattern matching abstract syntax trees. For example, the quoted expression $\langle\!\langle 1 + 2 \rangle\!\rangle$ matches the pattern $\langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle$, with $\langle\!\langle 1 \rangle\!\rangle$ for x and $\langle\!\langle 2 \rangle\!\rangle$ for y. More examples are given below.

These *reFLect* features meet three related demands of our intended applications in hardware modeling and theorem proving. First, antiquotation and pattern matching make it easy to write term manipulation functions – specifically, the kinds of term manipulation needed to implement a theorem prover, but also term manipulations that transform embedded circuit descriptions. Second, the reflection features of *reFLect* allow us to mix evaluation and theorem proving. Finally, *reFLect* quotation provides a flexible framework in which to embed both logics and domain-specific languages.

2.1 Term manipulation

Theorem proving systems like HOL have many ML functions for constructing and destructing terms in the object language. Function applications are a typical example: HOL provides an ML function that maps terms 'f' and 'x' to the function application term '$f \cdot x$', and an ML function that takes an application term '$f \cdot x$' apart and returns the subterms 'f' and 'x'.

[1] http://www.intel.com/software/products/opensource/tools1/verification

Analogous functions can be implemented in $\mathit{reFL}^{\mathit{ect}}$ for constructing and destructing quoted $\mathit{reFL}^{\mathit{ect}}$ applications. The definitions are as follows:

$$\begin{array}{l}
\mathsf{let}\ \mathit{make\_apply}\ f\ x = \langle\!\langle {}^\wedge f \cdot {}^\wedge x \rangle\!\rangle \\
\mathsf{let}\ \mathit{dest\_apply}\ \langle\!\langle {}^\wedge f \cdot {}^\wedge x \rangle\!\rangle = (f, x)
\end{array}$$

The *make_apply* function shows the use of the antiquotation splicing operation to construct a term from the supplied arguments. For example $\mathit{make\_apply}\cdot\langle\!\langle \mathsf{inc} \rangle\!\rangle\cdot\langle\!\langle 7 \rangle\!\rangle$ results in $\langle\!\langle {}^\wedge\langle\!\langle \mathsf{inc} \rangle\!\rangle \cdot {}^\wedge\langle\!\langle 7 \rangle\!\rangle \rangle\!\rangle$, which under the reduction rules of $\mathit{reFL}^{\mathit{ect}}$ reduces to $\langle\!\langle \mathsf{inc}\cdot 7 \rangle\!\rangle$. The *dest_apply* function first illustrates the definition of a $\mathit{reFL}^{\mathit{ect}}$ function by pattern matching over quoted expressions. When *dest_apply* is applied to a specific quotation, $\langle\!\langle \mathsf{inc}\cdot 7 \rangle\!\rangle$ say, the pattern $\langle\!\langle {}^\wedge f \cdot {}^\wedge x \rangle\!\rangle$ gets matched to this quotation – in this case binding f to $\langle\!\langle \mathsf{inc} \rangle\!\rangle$ and x to $\langle\!\langle 7 \rangle\!\rangle$. The result returned is the pair of terms $(\langle\!\langle \mathsf{inc} \rangle\!\rangle, \langle\!\langle 7 \rangle\!\rangle)$.

A more complex example is the $\mathit{reFL}^{\mathit{ect}}$ function below, which traverses a quoted expression and swaps the operands of any occurrence of the infix function '+'.

$$\begin{array}{lll}
\mathsf{letrec}\ \mathit{comm}\ \langle\!\langle {}^\wedge x + {}^\wedge y \rangle\!\rangle & = & \langle\!\langle {}^\wedge(\mathit{comm}\cdot y) + {}^\wedge(\mathit{comm}\cdot x) \rangle\!\rangle \\
\mid\quad \mathit{comm}\ \langle\!\langle {}^\wedge f \cdot {}^\wedge x \rangle\!\rangle & = & \langle\!\langle {}^\wedge(\mathit{comm}\cdot f)\cdot{}^\wedge(\mathit{comm}\cdot x) \rangle\!\rangle \\
\mid\quad \mathit{comm}\ \langle\!\langle \lambda {}^\wedge p. {}^\wedge b \rangle\!\rangle & = & \langle\!\langle \lambda {}^\wedge p. {}^\wedge(\mathit{comm}\cdot b) \rangle\!\rangle \\
\mid\quad \mathit{comm}\ \langle\!\langle \lambda {}^\wedge p. {}^\wedge b \mid {}^\wedge a \rangle\!\rangle & = & \langle\!\langle \lambda {}^\wedge p. {}^\wedge(\mathit{comm}\cdot b) \mid {}^\wedge(\mathit{comm}\cdot a) \rangle\!\rangle \\
\mid\quad \mathit{comm}\ x & = & x
\end{array}$$

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

We draw another example from our application domain of theorem proving. In an LCF-architecture theorem prover (Gordon *et al.*, 1979), such as the one we have implemented in $\mathit{reFL}^{\mathit{ect}}$, an abstract data-type of *theorems* is provided by a module that exports only functions that implement valid rules of inference for the logic being provided. This module forms the 'trusted core' of the theorem prover; the soundness of the whole system depends on this core being correctly implemented. In this context, a merit of the $\mathit{reFL}^{\mathit{ect}}$ pattern-matching style of programming is that it gives compact code that is also easy to read and understand.

In the $\mathit{reFL}^{\mathit{ect}}$ theorem prover, we use a sequent-based representation of theorems. These are represented by values of an abstract data-type similar to the one defined by the $\mathit{reFL}^{\mathit{ect}}$ declarations below:

$$\begin{array}{l}
\mathsf{lettype}\ \mathit{thm} = \vdash\ (\mathit{term\,list})\ \mathit{term} \\
\mathsf{infix}\ 4\ \vdash
\end{array}$$

This makes $\vdash$ an infix data-type constructor for the type *thm*. The constructor maps a list of terms Γ (representing assumptions) and a term P (representing a conclusion) to the theorem $\Gamma \vdash P$. In natural deduction, the rule for conjunction introduction is stated as follows:

$$\frac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma, \Delta \vdash P \wedge Q}$$

In the trusted core of *reFLect*, a function for this inference rule can be implemented with pleasing syntactic directness:

$$\mathsf{let}\ \mathit{conj\_intro}\ (G \vdash P)\ (D \vdash Q) = (G\ \mathsf{union}\ D) \vdash \langle\!\langle \hat{}P \wedge \hat{}Q \rangle\!\rangle$$

One aim of the *reFLect* pattern matching mechanism is to achieve the transparency and simplicity of this style of programming for syntactic manipulation. Aasa, Petersson and Synek (Aasa *et al.*, 1988) advocate a similar mechanism, called *quotation patterns*, for manipulating object-language expressions within a meta-language.

2.2 Reflection

The object logic of systems like HOL is typically a version of higher-order logic defined on top of the λ-calculus. The construction follows the lines of Church's formulation of simple type theory (Church, 1940), in which primitive symbols are added for certain constants such as equality and the quantifiers are defined using λ abstraction. The logic inherits a semantics for term equality from the λ-calculus; in particular, it inherits the various reduction rules of the λ-calculus, which appear in the logic as inference rules.

Defining a logic on top of *reFLect* gives a higher-order logic that includes the *reFLect* reduction rules in the same way. In a theorem prover implemented in *reFLect*, the data representations of the object and meta-languages are the same. Hence reduction by execution of *reFLect* programs also directly implements reduction by formal inference in the logic. Theorem provers with separate object and meta languages, on the other hand, need to include special code for efficient execution of object-language expressions.

This link between program execution and logical inference provides a form of *reflection* (Harrison, 1995) in the *reFLect* theorem prover. We can do equality proofs by term reduction in the theorem prover efficiently, just by evaluating the *reFLect* expressions. In particular, we can prove certain logical theorems just by using the *reFLect* evaluation mechanism to evaluate the statement of each theorem to true. Conversely, we may obtain any *reFLect* program that evaluates to true as a theorem of our logic.

In the Forte system, the invocation of a model checker is just a *reFLect* function call, so reflection also provides a logically principled connection between theorems in higher-order logic and model checking results. The source text of any call to the model-checker that returns true becomes a theorem of the logic – in effect, making the model checker part of the trusted core of the logic, but through the general mechanism of reflection rather than in an ad-hoc way.

A similar mechanism called *lifted-FL* (Aagaard *et al.*, 1999) was available in earlier versions of Forte, but *reFLect* provides richer possibilities. For example, one can use quantifiers to create a bookkeeping framework that cleanly separates logical content from model-checking control parameters.

The unification of object language and meta-language data representations also allows efficient evaluation to be incorporated into term rewriting in the theorem prover. Consider, for example, the following theorem about bit-vectors that represent integers:

$$\begin{array}{l} \vdash \forall as\ bs. \\ \quad (\mathsf{length}\ as = \mathsf{length}\ bs) \supset \\ \quad\quad \mathsf{bvless}\ as\ bs = (\mathsf{bv2int}\ as < \mathsf{bv2int}\ bs) \end{array}$$

This says that if the lists *as* and *bs* are of equal length, then the result of comparing the bit-vectors they represent is equal to the result of converting them to integers and comparing these integers. In our theorem prover, this fact can be used as a conditional left-to-right rewrite rule; provided the condition about equal lengths can be discharged, we can eliminate uses of bvless in favor of bv2int and <.

Discharging the condition requires a proof, and in general this proof may be arbitrarily hard. In practice, however, the instantiation of *as* and *bs* for an application of this rule in a specific context will typically be to concrete list values, whose lengths we can just compute. Reflection allows us to do precisely this; our rewriter can discharge easy side conditions by doing a proof by evaluation using the *reFLect* interpreter.

2.3 Embeddings

The quotation construct in *reFLect* makes the whole of the *reFLect* language available as an embedded language within the *reFLect* programming language itself. Quotations are essentially a 'deep' embedding, in which the embedded language is represented as a data type. This is in contrast to a 'shallow' embedding, in which the embedded language is just a sublanguage of the language in which it is embedded. In a shallow embedding, one typically defines a collection of functions to represent various components of the language. The embedded language itself is then just the collection of programs that can be written using these functions.

The merit of a shallow embedding is that it directly inherits an efficient execution mechanism from the programming language in which it is defined. On the other hand, one cannot define functions that inspect the syntactic form of phrases in the embedded language. For example one cannot define functions that do transformations of expressions in the embedded language. For this, the language must be deeply embedded as a data type.

In *reFLect* we can have much of the power of a deep embedding in a shallow embedding; since we have a built-in deep embedding of all of *reFLect*, we also have a deep embedding of any sublanguage of it. Suppose, for example, we define a shallow embedding of an HDL into a standard functional programming language. We could then express circuits in the HDL as functional programs and simulate them by execution. On the other hand, suppose we perform a deep embedding by defining the HDL as a data type within a functional language, then we can write code that inspects and transforms HDL programs, but we cannot evaluate them without writing our own evaluator. In *reFLect*, however, we can do a shallow embedding and get simulation-by-execution – but also use quotations and pattern matching to inspect and transform the phrases of the embedded HDL.

Theorem provers are typically implemented by a deep embedding of the expressions of some logical language. The LCF system is an example of an early theorem prover implemented in this style (Gordon *et al*., 1979). Expressions of the logic of computable functions were represented by a data type in ML. Modern theorem provers, like HOL (Gordon & Melham, 1993), Isabelle (Nipkow *et al*., 2002) and Coq (Dowek *et al*., 1993), continue to use this approach. The ACL2 system for untyped first-order logic is an exception (Kaufmann *et al*., 2000b). Like *reFLect*, ACL2 uses reflection so that its implementation language, Applicative Common LISP, is available as a deep embedding within itself and also serves as the language of logical expressions. This facility allows shallowly embedded hardware models in ACL2 to be efficiently simulated as well as reasoned about (Greve *et al*., 2000; Moore, 1998). Indeed, *reFLect* has been developed in part to support this key benefit of ACL2 in a theorem prover based on a typed higher-order logic.

In verification and other applications, languages are also sometimes embedded into the *logical* languages of theorem provers. For example, if you wanted to reason about designs expressed in a hardware description language, say VHDL, you would first embed (probably a subset of) VHDL in the logic of your theorem prover. In HOL, for instance, you would embed the language of VHDL statements in the language of higher-order logic expressions, which is in turn embedded in the ML programming language.

Embeddings into logics may also be deep or shallow, with similar trade-offs to those to be considered when embedding into a programming language. A shallow embedding in a programming language immediately confers an execution semantics for evaluation on the embedded language. When embedding into a logic, a shallow embedding immediately confers a logical semantics for reasoning about expressions in the embedded language. While a shallow embedding in a programming language immediately allows execution of the embedded language, a deep embedding is required if you wish to support transformation of expressions in the embedded language. Similarly, a shallow embedding in the logical expression language of a theorem prover immediately supports reasoning about the embedded language, but a deep embedding is required if you wish to support reasoning about functions that transform expressions in the embedded language.

Since quotations in *reFLect* may be nested, the reflection features of the language allow us to do a shallow embedding in a logic but also get many of the benefits of a deep embedding. For example, a shallow embedding of a hardware description language in *reFLect* allows efficient execution of expressions in that embedded language, and hence efficient hardware simulation. We can also reason about the meaning of expressions in the hardware description language using quotations and the *reFLect* theorem prover. Further, with another level of quotation, we can reason about *reFLect* functions that transform hardware designs expressed in the embedded language.

3 Syntax

The syntax of *reFLect* 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.

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

Fig. 1. The syntax of types.

$$\begin{array}{lll@{\qquad}lll} \mathsf{vars}\,\alpha & = & \{\alpha\} & \alpha_\phi & = & \phi\,\alpha \\ \mathsf{vars}(\sigma_1, \ldots \sigma_n)c & = & \mathsf{vars}\,\sigma_1 \cup \ldots \mathsf{vars}\,\sigma_n & (\sigma_1, \ldots \sigma_n)c_\phi & = & (\sigma_{1\phi}, \ldots \sigma_{n\phi})c \end{array}$$

Fig. 2. Type operations: variables and instantiation.

3.1 Types

The *reFLect* 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 *reFLect* type system contains one interesting atomic type: *term*, the type of a quoted *reFLect* expression. Figure 1 shows the syntax of the *reFLect* 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*.

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 $1 \leqslant i \leqslant n$, 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 2.

3.2 Expressions

The syntax of *reFLect* expressions, shown in figure 3, 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

$\Lambda, \mathrm{M}, \ldots$	$::=$	$k \mathbin{⦂} \sigma$	– Constant
	$\mid$	$v \mathbin{⦂} \sigma$	– Variable
	$\mid$	$\lambda\Lambda.\,\mathrm{M}$	– Abstraction
	$\mid$	$\lambda\Lambda.\,\mathrm{M} \mid \mathrm{N}$	– Alternation
	$\mid$	$\Lambda\cdot\mathrm{M}$	– Application
	$\mid$	$\langle\!\langle\Lambda\rangle\!\rangle$	– Quotation
	$\mid$	$\hat{}\,\Lambda \mathbin{⦂} \sigma$	– Antiquotation

Fig. 3. The syntax of expressions.

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. These will be explained and motivated below. Our intent has been to keep the abstract syntax of *reFLect* as simple as possible, rather than – say – designing an abstract syntax that follows the concrete syntax more closely. Our motivation for this is to keep the algorithms that traverse and transform abstract syntax trees as simple as they can be.

3.2.1 Constants

Constants are not theoretically necessary in a presentation of a λ-calculus and are therefore often omitted. Constants are, however, needed for the construction of a logic on top of a λ-calculus. Since *reFLect* is intended equally as a programming language and as the foundation for a logic, we include constants in our presentation from the beginning.

3.2.2 Quotations

A *reFLect* expression may contain a quoted *reFLect* expression. These are written using the form $\langle\!\langle\Lambda\rangle\!\rangle$ and denote abstract syntax trees. There is also an antiquotation operation, which is used to splice one abstract syntax tree into another. Antiquotation of the term Λ into a context requiring an abstract syntax tree of type σ is written $\hat{}\,\Lambda \mathbin{⦂} \sigma$. Antiquotation may only appear inside a quotation; it is most properly thought of as a two place operation between the antiquoted term and the quotation into which it will be spliced. We will not always show the type annotation on an antiquotation when the type required of the term is readily apparent. Section 6.1 will explain how and when antiquotations may be reduced, but as an example consider the expression $\langle\!\langle 1 + \hat{}\langle\!\langle 2+3\rangle\!\rangle\rangle\!\rangle$. This expression may be reduced 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 *reFLect* 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 an expression like $(\lambda\langle\!\langle \hat{}x + \hat{}y\rangle\!\rangle.\,\langle\!\langle \hat{}y + \hat{}x\rangle\!\rangle)\cdot\langle\!\langle 1+2\rangle\!\rangle$, which is semantically equal to $\langle\!\langle 2+1\rangle\!\rangle$.

Not all attempts to execute an application by pattern matching will succeed, so *reFLect* 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 languages omit pattern matching from their abstract syntax so as to simplify their semantics. We considered doing this with *reFLect*, but decided against it because we wished to be able to reason about *reFLect* programs – represented by their abstract syntax trees – that perform term manipulations by pattern matching. The earlier *FL* system took the traditional approach of excluding pattern matching from abstract syntax, and so reasoning about *FL* programs that used pattern matching had to be conducted on the abstract syntax trees of those programs after pattern matching had been compiled into conditional expressions. This was inconvenient and unnatural. By retaining pattern matching in the abstract syntax trees of *reFLect* we support reasoning about *reFLect* programs in a form closer to the concrete syntax in which the user wrote them. For example, the definition of the *comm* function described in section 2.1 is captured directly by the following expression.

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

Syntactically, any *reFLect* 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, algorithms that traverse expressions would be more complicated to write, 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 we can devise an implementation that does not degrade the performance of algorithms that make heavy use of antiquotation for expression construction.

3.2.4 Polymorphism

Functional programming languages are usually based on extensions of the λ-calculus with a form of abstraction over types, giving a *polymorphic* λ-calculus. This can be done by introducing quantified types and a special language construct to abstract over types, as for example in System-F (Girard *et al.*, 1989). Such languages are computationally sound, in the sense that they are strongly normalizing.

The *reFLect* language does not support this kind of polymorphism; there is no type quantification clause in figure 1, and this omission is deliberate. This is because we construct a higher-order logic on top of *reFLect*, in the same way that Church constructs a higher-order logic on top of the simply typed λ-calculus. If we were to use a polymorphic λ-calculus as our basis, we would arrive at a polymorphic higher-order logic – and such logics have been shown to exhibit paradoxes (Coquand, 1986; Coquand, 1991; Stump, 1999).

To avoid these paradoxes, polymorphism in the underlying λ-calculus must be restricted. In *reFLect* we do this by having a constant definition mechanism that can introduce polymorphic constants – but also taking the identity of an occurrence of such a constant to be determined by its name *and* type at that occurrence, which avoids the paradoxes. The result is a calculus with a limited but useful form of polymorphism. This is exactly the same approach taken in the HOL system. The higher-order logic constructed on the basis of a λ-calculus with this limited form of polymorphism has been shown to have a set theoretic semantics by Pitts (Gordon & Melham, 1993). We could have chosen a more elaborate type theory, like the Calculus of Constructions, as the basis for our language and its logic, but the HOL logic has the advantage of simplicity, and previous experience with this logic at Intel (Aagaard *et al.*, 2000; Jones *et al.*, 2001; Kaivola & Aagaard, 2000; Kaivola & Kohatsu, 2001; Kaivola & Narasimhan, 2001) has shown it well suited for our applications in hardware verification.

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.

$$\begin{array}{rcll} \Lambda, \mathrm{M}, \ldots & ::= & \ldots & \textit{(as in figure 3)} \\ & \mid & \_ \mathbin{\circ\!\!\circ} \sigma & \text{– A hole} \end{array}$$

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, $\mathscr{C}$, $\mathscr{D}$, etc., as syntactic meta-variables ranging over contexts. We will use the notation $\mathscr{C}[\_ \mathbin{\circ\!\!\circ} \sigma_1, \ldots \_ \mathbin{\circ\!\!\circ} \sigma_n]$ to indicate that the context $\mathscr{C}$ has the n holes shown. The order in which the holes are indicated is unimportant, except that it must be fixed for any given context. We write $\mathscr{C}[\Lambda_1, \ldots \Lambda_n]$ to stand for the expression resulting from a context $\mathscr{C}[\_ \mathbin{⦂} \sigma_1, \ldots \_ \mathbin{⦂} \sigma_n]$, where $\sigma_1, \ldots\ \sigma_n$ are the types of $\Lambda_1, \ldots\ \Lambda_n$ respectively, in which each hole $\_ \mathbin{⦂} \sigma_i$ 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.

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 subexpression may occur at negative level.

Level is an important notion in *reFLect* because it affects variable binding and reduction. 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 any expression may be considered as a context 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, $\hat{\ }\_ + 1$ is not level consistent, but $\langle\!\langle \hat{\ }\_ + 1 \rangle\!\rangle$ is.

Figure 4 formalizes our notion of a level consistent context by defining judgments of the form $n \vdash \mathscr{C}$, which should be read as '$\mathscr{C}$ is level consistent at level n'. We may read judgments of the form $0 \vdash \mathscr{C}$ as simply '$\mathscr{C}$ is level consistent'. If the unique derivation of $0 \vdash \mathscr{C}$ contains a subderivation with the intermediate conclusion $n \vdash \mathscr{D}$, then we say that '$\mathscr{D}$ occurs at level n in $\mathscr{C}$'.

The following properties follow from the definition of level consistency.

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

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

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

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

$$\frac{n \vdash \mathscr{C} \quad n \vdash \mathscr{D}}{n \vdash \lambda\mathscr{C}.\mathscr{D}} \qquad \frac{n \vdash \mathscr{C} \quad n \vdash \mathscr{D} \quad n \vdash \mathscr{E}}{n \vdash \lambda\mathscr{C}.\mathscr{D} \mid \mathscr{E}}$$

$$\frac{n \vdash \mathscr{C} \quad n \vdash \mathscr{D}}{n \vdash \mathscr{C}\cdot\mathscr{D}} \qquad \frac{n+1 \vdash \mathscr{C}}{n \vdash \langle\!\langle \mathscr{C} \rangle\!\rangle} \qquad \frac{n \vdash \mathscr{C}}{n+1 \vdash ({}^{\wedge}\mathscr{C} : \sigma)}$$

Fig. 4. A level consistent context, $n \geqslant 0$.

Proposition 4

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

Proposition 4 allows us to treat contexts as a form of general constructor for quoted expressions. We will use an expression of the form $\langle\!\langle \mathscr{C}[{}^{\wedge}\Lambda_1 : \sigma_1, \ldots {}^{\wedge}\Lambda_n : \sigma_n] \rangle\!\rangle$ under the condition $0 \vdash \mathscr{C}[\_ : \sigma_1, \ldots \_\sigma_n]$ 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 5 and 7 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 (Taha & Sheard, 2002; Sheard & Peyton Jones, 2002) in which the entire term is traversed, and the traversal function tracks the level of the expression.

4.2 Typing

All quoted expressions in *reFL*^{ect} have the same type, *term*. In *FL*, the type of a quoted expression depended on what was inside the quote (Aagaard *et al.*, 1999). 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 (Taha & Sheard, 2002). 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 use dependent types to address the problem of typing transformation routines (Pašalić *et al.*, 2002). And for some functions, such as the *dest_apply* of section 2.1, existential types (Cardelli & Wegner, 1985) might suffice. But there are still functions, such as finding a list of free variables, that are important for implementing theorem provers and which cannot be typed with existential or even dependent types. 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

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

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

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

Fig. 5. A well typed expression.

in a Hindley-Milner type system. The same decision is made for similar reasons in Template Haskell (Sheard & Peyton Jones, 2002).

This means, of course, that in $\mathit{reFL}^{\mathit{ect}}$ some type-checking must be done at run time.[2] For example the expression $\langle\!\langle 1 + \hat{}x \rangle\!\rangle$ is well-typed and requires x to be of 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 $\vdash \Lambda : \sigma$ by the rules of figure 5. Some of the rules merit explanation:

- We suppose that each constant symbol k has an associated *most-general type*, $\mathsf{mgtype}\, k$. 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 that type.
- A quoted expression – which can be factored into a context and a list of antiquoted expressions – is well-typed with type *term* if each of the antiquoted expressions is well-typed with type *term*, and filling the context with level-consistent expressions of appropriate type yields a well-typed expression. The simplest collection of expressions to fill the context with is a collection of fresh variables. The type of the expression formed by filling the holes in the context does not figure in the type of the quoted expression.

Type inference in $\mathit{reFL}^{\mathit{ect}}$ takes user input and constructs well-typed expressions, attaching the type annotations required to variables, constants and antiquotations.

[2] This is unlike Template Haskell, where second-level type errors are caught at compile time.

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.

4.2.2 Variables and types

In *reFLect* 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 $\langle\!\langle \hat{}\langle\!\langle x{:}\alpha \to \beta\rangle\!\rangle \cdot \hat{}\langle\!\langle x{:}\alpha\rangle\!\rangle\rangle\!\rangle$ may be reduced, using rules we will describe in section 6.1, to $\langle\!\langle x{:}\alpha \to \beta \cdot x{:}\alpha\rangle\!\rangle$.

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-inference environment describing the types of the variables it contains. For antiquotations we would record not only the expected type, but also the prevailing type-inference 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 again the standard logical rule for conjunction introduction, and its implementation from section 2.1:

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

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 *reFLect* shares with the object languages of more conventional theorem proving systems for typed logics, such as HOL.

4.2.3 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 $\mathscr{C}_{n\phi}$ to indicate the result of applying the type instantiation ϕ to the context (or expression) $\mathscr{C}$ at level n. In the case where n is zero we will simply write $\mathscr{C}_{\phi}$. Type instantiation of a context is defined in figure 6.

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

Fig. 6. Type instantiation of a context, $n \geqslant 0$.

Proposition 5
If $0 \vdash \Lambda$ and $\vdash \Lambda : \sigma$, then for any type instantiation ϕ, $0 \vdash \Lambda_{\phi}$ and $\vdash \Lambda_{\phi} : \sigma_{\phi}$

5 Abstractions

Abstractions in *reFL^ect* 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 explanation of the meaning of abstraction in *reFL^ect*.

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*. In *reFL^ect* we allow 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, which the expression is said to bind.

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 are α equivalent.

In *reFL^ect*, 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

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

Fig. 7. Free variables.

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 literally 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 7 shows the definition of the 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.

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

Suppes (1957) 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 (1999) 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.

The derivation of the contradiction that would follow from having cross-stage persistence in our logic relies on quantification over a cross-stage persistent value. Since constants are not bound by quantifiers, this objection to cross-stage persistence does not apply to them. In *reFLect*, therefore, constants are taken to refer to the same value at all levels of quotation. Just as the inclusion of a constant definition mechanism in *reFLect* provides a logically sound form of restricted polymorphism, so too it provides a sound form of restricted cross-stage persistence.

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 terms made up of 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 a 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 is applied must be an addition 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 (Müller & Slind, 1997). Formally, one uses a *selection* operator (Leisenring, 1969) 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

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

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

(where $0 \vdash \mathscr{C}[\_{:}\sigma_1, \ldots \_{:}\sigma_n]$ and ι is a renaming such that:
$\mathsf{dom}\,\iota \subseteq \mathsf{free}\,\Lambda$, and $\mathsf{dom}\,\theta \cap \iota(\mathsf{free}\,\Lambda) = \{\}$, and
$(\mathsf{free}\,\mathrm{M} - \mathsf{free}\,\Lambda) \cap \iota(\mathsf{dom}\,\iota) = \{\}$, and
$\mathsf{free}(\theta(\mathsf{free}\,\mathrm{M} - \mathsf{free}\,\Lambda)) \cap \iota(\mathsf{free}\,\Lambda) = \{\}$)

Fig. 8. Substitution.

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

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

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

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 $\mathsf{dom}\,\theta$ for the *domain* of θ, meaning the set of variables for which θ is not the identity. If $\mathsf{dom}\,\theta = \{v_1{:}\sigma_1, \ldots v_n{:}\sigma_n\}$ and $\theta(v_i{:}\sigma_i) = \Lambda_i$ for all $1 \leqslant i \leqslant n$, 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 8 defines this operation.[3]

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 consistently renamed. Note also that only bound variables, variables at level 0 in the patterns of level 0 abstractions, need be renamed to avoid capture; quoted variables are not impacted.

[3] In the condition of figure 8, ι, θ, and free are implicitly extended to image functions over sets where required.

$$\frac{\vdash \mathit{definition}\,k : \tau \qquad \tau_\phi = \sigma}{\vdash k ⦂ \sigma \rightarrow (\mathit{definition}\,k)\phi}\,[\delta]$$

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

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

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

$$\frac{0 \vdash \mathscr{C}[\_ ⦂ \sigma_1, \ldots \_ ⦂ \sigma_n] \qquad \vdash \Lambda_1 : \sigma_{1\phi} \quad \ldots \quad \vdash \Lambda_n : \sigma_{n\phi} \qquad \mathit{dom}\,\phi \subseteq \mathit{vars}\{\sigma_1, \ldots \sigma_n\}}{\vdash \langle\!\langle \mathscr{C}[\hat{}\langle\!\langle \Lambda_1 \rangle\!\rangle ⦂ \sigma_1, \ldots \hat{}\langle\!\langle \Lambda_n \rangle\!\rangle ⦂ \sigma_n] \rangle\!\rangle \rightarrow \langle\!\langle \mathscr{C}_\phi[\Lambda_1, \ldots \Lambda_n] \rangle\!\rangle}\,[\psi]$$

Fig. 9. Reduction.

Proposition 6
If $0 \vdash \Lambda$ and $\vdash \Lambda : \sigma$, then for any substitution θ, $0 \vdash \Lambda\theta$ and $\vdash \Lambda\theta : \sigma$.

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, $(\lambda(x ⦂ \alpha, x ⦂ \beta).\, x ⦂ \alpha)[\beta/\alpha]$ should produce $\lambda(x' ⦂ \beta, x ⦂ \beta).\, x' ⦂ \beta$ or $\lambda(x ⦂ \beta, x' ⦂ \beta).\, x ⦂ \beta$.

The formal definition of type instantiation for expressions is similar to the definition of substitution in figure 8. Note that $\Lambda\phi$ is not the same as the context type instantiation operation $\mathscr{C}_\phi$ in figure 6, 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 9 and 10 present the reduction rules for evaluating a *reFLect* expression. The rules in figure 9 describe individual reductions, while those in figure 10 describe how reductions may be applied to subexpressions. The judgments are of the form $\vdash \Lambda \rightarrow \Lambda'$, 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

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

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

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

Fig. 10. Reducing subexpressions.

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

Fig. 11. Reduction closure.

not fall in the binding position of a level zero abstraction. We use the standard notation $\vdash \Lambda \overset{*}{\to} \Lambda'$ to indicate that Λ can be reduced to Λ' in zero or more steps. This is formalized in figure 11.

The rules of figure 9 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 to be executable patterns: 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. Note that we do not define any reduction rules for abstractions with patterns that we do not consider to be executable.

Proposition 7
If $0 \vdash \Lambda$ and $\vdash \Lambda : \sigma$, then for any M such that $\vdash \Lambda \overset{*}{\to} \mathrm{M}$ we have $0 \vdash \mathrm{M}$ and $\vdash \mathrm{M} : \sigma$.

Proposition 7 is the subject reduction property for *reFL*ᵉᶜᵗ. 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 (Krstić & Matthews, 2003).

6.1 Reducing quotations

The rule for ψ-reduction in figure 9 allows the elimination of antiquoted quotations at level one. The rule admits 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* → *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. Type inference will annotate the antiquotations in the definition of *comm* with polymorphic types so that, for example, $comm \cdot \langle\!\langle inc \cdot (1+2) \rangle\!\rangle$ reduces at run time to $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle \mathbin{\raisebox{0.2ex}{:}} \alpha \to \beta \cdot \hat{}\langle\!\langle 2+1 \rangle\!\rangle \mathbin{:} \alpha \rangle\!\rangle$. Then, using ψ-reduction, we get the expected expression $\langle\!\langle inc \cdot (2+1) \rangle\!\rangle$.

The rule does not allow reductions to create badly-typed expressions. For example, we cannot use this rule to reduce the expression $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle : \alpha \to \beta \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle : \alpha \rangle\!\rangle$. 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 $\langle\!\langle \hat{}\langle\!\langle f : \alpha \to \beta \rangle\!\rangle : int \to \gamma \cdot 1 \rangle\!\rangle$.

6.1.1 *Instantiation must affect the entire term*

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

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

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

6.1.2 *All antiquotes eliminated simultaneously*

The assumption $0 \vdash \mathscr{C}[\_ : \sigma_1, \ldots \_ : \sigma_n]$ 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 $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle : \alpha \to \beta \cdot \hat{}\langle\!\langle 1 \rangle\!\rangle : \alpha \rangle\!\rangle$ to $\langle\!\langle inc \cdot \hat{}\langle\!\langle 1 \rangle\!\rangle : int \rangle\!\rangle$ and later to $\langle\!\langle inc \cdot 1 \rangle\!\rangle$. But this rule would also allow us to reduce $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle : \alpha \to \beta \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle : \alpha \rangle\!\rangle$ to both $\langle\!\langle inc \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle : int \rangle\!\rangle$ and $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle : int \to \beta \cdot \mathsf{T} \rangle\!\rangle$. Since these expressions may not be further reduced this would leave *reFLect* with a non-confluent reduction system.[4]

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

[4] It does have a property similar to confluence, if expressions like these are considered equivalent.

in our applications – so we do not complicate the semantics to facilitate this relaxation.

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.* (1998).

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 (Sheard & Peyton Jones, 2002) or staged evaluation (Taha & Sheard, 2002), which focus more on flexible ways of constructing or specializing programs.

For example, a ubiquitous theorem proving application is term rewriting (Paulson, 1983), 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 subexpressions large. The same subexpression may be spliced into more than one context to form several expressions. In rewriting, for example, the original term and the rewritten term typically have many common subexpression. If we unified types when splicing a subexpression into a context then it would not be possible to share subexpressions, we would have to copy it. Since the efficiency 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 10 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 seems simpler just to forbid pattern reductions.

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

Fig. 12. Valid pattern.

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

Fig. 13. Match readiness.

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 expression in the binding position of $\lambda x + 1.\, x$ 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 12 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 an invalid pattern should lead to a 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 13 formalizes this notion, which is used in the rules for β and γ-reduction in figure 9.

Consider what can happen without this restriction by contemplating the effect of *dest_apply* from section 2. If we apply this to the expression $\langle\!\langle g{:}\alpha \to \beta \hat{}\langle\!\langle 1\rangle\!\rangle{:}\alpha\rangle\!\rangle$ and we were to evaluate the application before ψ-reducing the argument we would get the result $(\langle\!\langle g{:}\alpha \to \beta\rangle\!\rangle, \langle\!\langle \hat{}\langle\!\langle 1\rangle\!\rangle{:}\alpha\rangle\!\rangle)$, which reduces to $(\langle\!\langle g{:}\alpha \to \beta\rangle\!\rangle, \langle\!\langle 1\rangle\!\rangle)$. If we were to ψ-reduce the argument before reducing the application we would get the result $(\langle\!\langle g{:}\, int \to \beta\rangle\!\rangle, \langle\!\langle 1\rangle\!\rangle)$.

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

$$\frac{v \mathbin{⦂} \sigma\,\theta = \Xi}{(v \mathbin{⦂} \sigma, \theta)\ \textsf{\textit{matches}}\ \Xi}$$

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

Fig. 14. Pattern matching an expression.

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

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

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

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

$$\frac{\begin{array}{l} \sigma_{1\chi} = \tau_1 \quad \ldots \quad \sigma_{n\chi} = \tau_n \\ \chi \vdash \mathscr{C}[v_1 \mathbin{⦂} \sigma_1, \ldots v_n \mathbin{⦂} \sigma_n] \rightsquigarrow \mathscr{C}'[v_1 \mathbin{⦂} \tau_1, \ldots v_n \mathbin{⦂} \tau_n] \\ 0 \vdash \mathscr{C}[\__1 \mathbin{⦂} \sigma_1, \ldots \__n \mathbin{⦂} \sigma_n] \quad 0 \vdash \mathscr{C}'[\__1 \mathbin{⦂} \tau_1, \ldots \__n \mathbin{⦂} \tau_n] \\ \phi \vdash \Lambda_1 \rightsquigarrow \Lambda'_1 \quad \ldots \quad \phi \vdash \Lambda_n \rightsquigarrow \Lambda'_n \end{array}}{\begin{array}{l} \phi \vdash \langle\!\langle \mathscr{C}[\hat{}\Lambda_1 \mathbin{⦂} \sigma_1, \ldots \hat{}\Lambda_n \mathbin{⦂} \sigma_n] \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathscr{C}'[\hat{}\Lambda'_1 \mathbin{⦂} \tau_1, \ldots \hat{}\Lambda'_n \mathbin{⦂} \tau_n] \rangle\!\rangle \\ \quad \text{(where } v_1 \ldots v_n \text{ are fresh)} \end{array}}$$

Fig. 15. Type-match relation.

expression matches the pattern. The predicate *matches*, defined in figure 14, 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 $\mathscr{C}[\_ \mathbin{⦂} \sigma_1, \ldots \_ \mathbin{⦂} \sigma_n]$ and term variables $v_1, \ldots\ v_n$ such that the pattern we are trying to match against is

$$\langle\!\langle \mathscr{C}[\hat{}(v_1 \mathbin{⦂} \mathit{term}) \mathbin{⦂} \sigma_1, \ldots \hat{}(v_n \mathbin{⦂} \mathit{term}) \mathbin{⦂} \sigma_n] \rangle\!\rangle$$

Next we must find a level consistent context $\mathscr{D}[\_ \mathbin{⦂} \tau_1, \ldots \_ \mathbin{⦂} \tau_n]$ and list of subexpressions $\Xi_1, \ldots\ \Xi_n$ such that the expression we are trying to match is $\langle\!\langle \mathscr{D}[\Xi_1, \ldots \Xi_n] \rangle\!\rangle$. The expression matches the pattern if $\mathscr{D}$ is a type instance, in the sense explained below, of $\mathscr{C}$ 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 15. The role of the $\rightsquigarrow$ relation is to 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:

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

We would expect to be able to apply the first λ-abstraction in this function to expressions such as $\langle\!\langle \mathsf{Len}\cdot([]\mathbin{\vdots}\mathit{int\ list})\rangle\!\rangle$, and so the pattern $\langle\!\langle \mathsf{Len}\cdot([]\mathbin{\vdots}\alpha\ \mathit{list})\rangle\!\rangle$ 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:

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

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 reduced to the 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 λ-calculus. Expressions and contexts can be represented as ordinary algebraic data-types. The functions 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

$$
\begin{aligned}
&\mathsf{E}\ \ulcorner v : \tau \urcorner = v\\
&\mathsf{E}\ \ulcorner k : \tau \urcorner = k\\
&\mathsf{E}\ \ulcorner \Lambda\cdot\mathrm{M} \urcorner = (\mathsf{E}\ \Lambda)\ (\mathsf{E}\ \mathrm{M})\\
&\mathsf{E}\ \ulcorner \lambda\Lambda.\,\mathrm{M} \urcorner = \mathsf{P}\ \Lambda\ (\mathsf{E}\ \mathrm{M})\ \mathsf{error}\\
&\mathsf{E}\ \ulcorner \lambda\Lambda.\,\mathrm{M} \mid \Xi \urcorner = \mathsf{P}\ \Lambda\ (\mathsf{E}\ \mathrm{M})\ (\mathsf{E}\ \Xi)\\
&\mathsf{E}\ \ulcorner \langle\!\langle \mathscr{C}[\hat{}\Lambda_1 : \tau_1, \ldots \hat{}\Lambda_n : \tau_n] \rangle\!\rangle \urcorner = \mathsf{fill}\ \ulcorner \langle\!\langle \mathscr{C}[\_ : \tau_1, \ldots \_ : \tau_n] \rangle\!\rangle \urcorner\ \mathbf{[}\mathsf{E}\ \Lambda_1, \ldots \mathsf{E}\ \Lambda_n\mathbf{]}\\
&\qquad\qquad (\text{where } 0 \vdash \mathscr{C}[\_ : \tau_1, \ldots \_ : \tau_n])
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathsf{L}\ \mathbf{[]}\ \mathrm{M}\ x = \mathrm{M}\\
&\mathsf{L}\ \mathbf{[}\ulcorner v : \tau \urcorner\mathbf{]}\ \mathrm{M}\ x = (\lambda v.\,\mathrm{M})\ (\mathsf{hd}\ x)\\
&\mathsf{L}\ (\ulcorner v : \tau \urcorner :: vs)\ \mathrm{M}\ x = (\lambda v.\,\mathsf{L}\ vs\ \mathrm{M}\ (\mathsf{tl}\ x))\ (\mathsf{hd}\ x)
\end{aligned}
$$

Fig. 16. Compilation to λ-calculus.

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 describe the compilation process with three functions: E (for compiling expressions), P (for compiling pattern abstractions), L (for compiling abstractions over lists of variables). The definitions of these functions are in figure 16. 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.[5] We will use quasi-quotation to separate values of the term and context data-types from the surrounding λ-calculus expressions, with which they share much common syntax.

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 with syntax $\lambda x.\,\lambda y.\,\langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle$. To achieve the effect of this program we assume a function fill that takes a context with n holes and a list of n terms, and forms a new term by applying the minimal type instantiation to

[5] The compiler described here 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.

the context that makes the type of each hole agree with the type of the corresponding term in the list, and then replaces each hole with the corresponding term from the list (with surrounding quotations removed). The function fails if the list does not have the same number of the terms 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 term in the list. The fill function is described below.[6]

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

$$\frac{\vdash \Lambda_1 : \sigma_{1\phi} \quad \ldots \quad \vdash \Lambda_n : \sigma_{n\phi} \quad \mathit{dom}\,\phi \subseteq \mathit{vars}\{\sigma_1, \ldots \sigma_n\}}{\vdash \mathsf{fill}\ \ulcorner\langle\!\langle \mathscr{C}[\_ {:} \sigma_1, \ldots \_ {:} \sigma_n]\rangle\!\rangle\urcorner\ [\ulcorner\langle\!\langle\Lambda_1\rangle\!\rangle\urcorner, \ldots \ulcorner\langle\!\langle\Lambda_n\rangle\!\rangle\urcorner] \overset{\lambda}{\rightarrow} \ulcorner\langle\!\langle \mathscr{C}_\phi[\Lambda_1, \ldots \Lambda_n]\rangle\!\rangle\urcorner}$$

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

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 as a pattern. The second argument is a function that takes a list of n terms as its argument. The third argument is a function from a term to the same return type as the second argument. The fourth argument is a term. It then attempts to match the term to the context, producing a list of terms 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 term instead.

$$\begin{array}{l}\mathsf{match} : \\ \quad \mathit{context} \rightarrow (\mathit{term\ list} \rightarrow \alpha) \rightarrow (\mathit{term} \rightarrow \alpha) \rightarrow \mathit{term} \rightarrow \alpha\end{array}$$

$$\frac{(\langle\!\langle \mathscr{C}[\hat{}(v_1 {:} \mathit{term}) {:} \sigma_1, \ldots \hat{}(v_n {:} \mathit{term}) {:} \sigma_n]\rangle\!\rangle, [\langle\!\langle\Xi_1\rangle\!\rangle, \ldots \langle\!\langle\Xi_n\rangle\!\rangle / v_1, \ldots v_n])\ \mathit{matches}\ \langle\!\langle \mathrm{N}\rangle\!\rangle}{\begin{array}{l}\vdash \mathsf{match}\ \ulcorner\langle\!\langle \mathscr{C}[\_ {:} \sigma_1, \ldots \_ {:} \sigma_n]\rangle\!\rangle\urcorner\ \Lambda\ \mathrm{M}\ \ulcorner\langle\!\langle \mathrm{N}\rangle\!\rangle\urcorner \overset{\lambda}{\rightarrow} \Lambda\ [\ulcorner\langle\!\langle\Xi_1\rangle\!\rangle\urcorner, \ldots \ulcorner\langle\!\langle\Xi_n\rangle\!\rangle\urcorner] \\ \qquad \text{(where } v_1, \ldots\ v_n \text{ are fresh)}\end{array}}$$

$$\frac{\nexists\theta.\,(\langle\!\langle \mathscr{C}[\hat{}v_1 {:} \sigma_1, \ldots \hat{}v_n {:} \sigma_n]\rangle\!\rangle, \theta)\ \mathit{matches}\ \langle\!\langle \mathrm{N}\rangle\!\rangle}{\begin{array}{l}\vdash \mathsf{match}\ \ulcorner\langle\!\langle \mathscr{C}[\_ {:} \sigma_1, \ldots \_ {:} \sigma_n]\rangle\!\rangle\urcorner\ \Lambda\ \mathrm{M}\ \ulcorner\langle\!\langle \mathrm{N}\rangle\!\rangle\urcorner \overset{\lambda}{\rightarrow} \mathrm{M}\ \ulcorner\langle\!\langle \mathrm{N}\rangle\!\rangle\urcorner \\ \quad \text{(where } v_1, \ldots\ v_n \text{ are fresh)}\end{array}}$$

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

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

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

7.3 Compilation example

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

$$\begin{array}{l}\mathsf{E}\ulcorner\lambda\langle\!\langle \hat{}x + \hat{}y\rangle\!\rangle.\,\langle\!\langle \hat{}(comm\cdot y) + \hat{}(comm\cdot x)\rangle\!\rangle\\ \quad|\,\lambda\langle\!\langle \hat{}f\cdot\hat{}x\rangle\!\rangle.\,\langle\!\langle \hat{}(comm\cdot f)\cdot\hat{}(comm\cdot x)\rangle\!\rangle\\ \quad|\,\lambda\langle\!\langle \lambda\hat{}p.\hat{}b\rangle\!\rangle.\,\langle\!\langle \lambda\hat{}p.\hat{}(comm\cdot b)\rangle\!\rangle\\ \quad|\,\lambda\langle\!\langle \hat{}p.\hat{}b\,|\,\hat{}a\rangle\!\rangle.\,\langle\!\langle \lambda\hat{}p.\hat{}(comm\cdot b)\,|\,\hat{}(comm\cdot a)\rangle\!\rangle\\ \quad|\,\lambda x.\,x\urcorner\end{array}$$

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 *reFLect* pattern matching λs into calls to the pattern compiler P.

$$\begin{array}{l}\ \mathsf{P}\ulcorner\langle\!\langle \hat{}x + \hat{}y\rangle\!\rangle\urcorner\ (\mathsf{E}\ulcorner\langle\!\langle \hat{}(comm\cdot y) + \hat{}(comm\cdot x)\rangle\!\rangle\urcorner)\\ (\mathsf{P}\ulcorner\langle\!\langle \hat{}f\cdot\hat{}x\rangle\!\rangle\urcorner\ (\mathsf{E}\ulcorner\langle\!\langle \hat{}(comm\cdot f)\cdot\hat{}(comm\cdot x)\rangle\!\rangle\urcorner)\\ (\mathsf{P}\ulcorner\langle\!\langle \lambda\hat{}p.\hat{}b\rangle\!\rangle\urcorner\ (\mathsf{E}\ulcorner\langle\!\langle \lambda\hat{}p.\hat{}(comm\cdot b)\rangle\!\rangle\urcorner)\\ (\mathsf{P}\ulcorner\langle\!\langle \lambda\hat{}p.\hat{}b\,|\,\hat{}a\rangle\!\rangle\urcorner\ (\mathsf{E}\ulcorner\langle\!\langle \lambda\hat{}p.\hat{}(comm\cdot b)\,|\,\hat{}(comm\cdot a)\rangle\!\rangle\urcorner)\\ (\mathsf{P}\ulcorner x\urcorner\ (\mathsf{E}\ulcorner x\urcorner)\ \mathsf{error}))))\end{array}$$

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.

$$\begin{array}{l}\ \mathsf{P}\ulcorner\langle\!\langle \hat{}x + \hat{}y\rangle\!\rangle\urcorner\ (\mathsf{fill}\ulcorner\langle\!\langle \_ + \_\rangle\!\rangle\urcorner\ [\mathsf{E}\ulcorner comm\cdot y\urcorner, \mathsf{E}\ulcorner comm\cdot x\urcorner])\\ (\mathsf{P}\ulcorner\langle\!\langle \hat{}f\cdot\hat{}x\rangle\!\rangle\urcorner\ (\mathsf{fill}\ulcorner\langle\!\langle \_\cdot\_\rangle\!\rangle\urcorner\ [\mathsf{E}\ulcorner comm\cdot f\urcorner, \mathsf{E}\ulcorner comm\cdot x\urcorner])\\ (\mathsf{P}\ulcorner\langle\!\langle \lambda\hat{}p.\hat{}b\rangle\!\rangle\urcorner\ (\mathsf{fill}\ulcorner\langle\!\langle \lambda\_.\,\_\rangle\!\rangle\urcorner\ [\mathsf{E}\ulcorner p\urcorner, \mathsf{E}\ulcorner comm\cdot b\urcorner])\\ (\mathsf{P}\ulcorner\langle\!\langle \lambda\hat{}p.\hat{}b\,|\,\hat{}a\rangle\!\rangle\urcorner\ (\mathsf{fill}\ulcorner\langle\!\langle \lambda\_.\,\_\,|\,\_\rangle\!\rangle\urcorner\ [\mathsf{E}\ulcorner p\urcorner, \mathsf{E}\ulcorner comm\cdot b\urcorner, \mathsf{E}\ulcorner comm\cdot a\urcorner])\\ (\mathsf{P}\ulcorner x\urcorner\ x\ \mathsf{error}))))\end{array}$$

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.

$$\begin{array}{l}\ \mathsf{P}\ulcorner\langle\!\langle \hat{}x + \hat{}y\rangle\!\rangle\urcorner\ (\mathsf{fill}\ulcorner\langle\!\langle \_ + \_\rangle\!\rangle\urcorner\ [comm\ y, comm\ x])\\ (\mathsf{P}\ulcorner\langle\!\langle \hat{}f\cdot\hat{}x\rangle\!\rangle\urcorner\ (\mathsf{fill}\ulcorner\langle\!\langle \_\cdot\_\rangle\!\rangle\urcorner\ [comm\ f, comm\ x])\\ (\mathsf{P}\ulcorner\langle\!\langle \lambda\hat{}p.\hat{}b\rangle\!\rangle\urcorner\ (\mathsf{fill}\ulcorner\langle\!\langle \lambda\_.\,\_\rangle\!\rangle\urcorner\ [p, comm\ b])\\ (\mathsf{P}\ulcorner\langle\!\langle \lambda\hat{}p.\hat{}b\,|\,\hat{}a\rangle\!\rangle\urcorner\ (\mathsf{fill}\ulcorner\langle\!\langle \lambda\_.\,\_\,|\,\_\rangle\!\rangle\urcorner\ [p, comm\ b, comm\ a])\\ (\mathsf{P}\ulcorner x\urcorner\ x\ \mathsf{error}))))\end{array}$$

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

$$\begin{array}{l}\ \mathsf{match}\ulcorner\langle\!\langle \_ + \_\rangle\!\rangle\urcorner\ (\lambda k.\,\mathsf{L}\ [\ulcorner x\urcorner,\ulcorner y\urcorner]\ (\mathsf{fill}\ulcorner\langle\!\langle \_ + \_\rangle\!\rangle\urcorner\ [comm\ y, comm\ x])\ k)\\ (\mathsf{match}\ulcorner\langle\!\langle \_\cdot\_\rangle\!\rangle\urcorner\ (\lambda l.\,\mathsf{L}\ [\ulcorner f\urcorner,\ulcorner x\urcorner]\ (\mathsf{fill}\ulcorner\langle\!\langle \_\cdot\_\rangle\!\rangle\urcorner\ [comm\ f, comm\ x])\ l)\\ (\mathsf{match}\ulcorner\langle\!\langle \lambda\_.\,\_\rangle\!\rangle\urcorner(\lambda m.\,\mathsf{L}\ [\ulcorner p\urcorner,\ulcorner b\urcorner]\ (\mathsf{fill}\ulcorner\langle\!\langle \lambda\_.\,\_\rangle\!\rangle\urcorner\ [p, comm\ b])\ m)\\ (\mathsf{match}\ulcorner\langle\!\langle \lambda\_.\,\_\,|\,\_\rangle\!\rangle\urcorner\\ \ (\lambda n.\,\mathsf{L}\ [\ulcorner p\urcorner,\ulcorner b\urcorner,\ulcorner a\urcorner]\ (\mathsf{fill}\ulcorner\langle\!\langle \lambda\_.\,\_\,|\,\_\rangle\!\rangle\urcorner\ [p, comm\ b, comm\ a])\ n)\\ (\lambda x.\,x))))\end{array}$$

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

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

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.[7] Suppose we use the notation $\Lambda \Rightarrow \Lambda'$ to mean that Λ is evaluated to produce Λ'. Certainly $\Lambda \Rightarrow \Lambda'$ implies $\Lambda \overset{*}{\rightarrow} \Lambda'$, 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:

$$\vdash \mathsf{eval} : \mathit{term} \rightarrow \mathit{term}$$

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

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. A call to value removes the quotes from around a term and interprets the result as a value of the appropriate type.

$$\vdash \mathsf{value}_\sigma : \mathit{term} \rightarrow \sigma$$

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

Antiquotation and value may appear similar in some respects, but there are several important differences between them:

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

- 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 a term is different from the actual type, then value may instantiate the type of the term. Antiquotation may instead instantiate the type of its context.
- Antiquotation does not alter the level of the quoted term, but value moves the term from level one to level zero.

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 (Taha & Sheard, 2002) or the static check for closed code in the system λ^{BN} (Benaissa *et al.*, 1999).

8.2 Value reification

The *reFL^ect* language also supports a partial inverse of value 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.

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

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 logical inconsistencies. For example, $\lambda x.\,x+1$ and $\lambda x.\,1+x$ are equal and extensionality therefore requires $\mathsf{lift}\cdot(\lambda x.\,x+1)$ and $\mathsf{lift}\cdot(\lambda x.\,1+x)$ to be equal. But the terms $\langle\!\langle\lambda x.\,x+1\rangle\!\rangle$ 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.

$$\mathsf{lift}\cdot\Lambda = \langle\!\langle [\![\Lambda]\!]\mathrel{:}\sigma\rangle\!\rangle$$

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 and types.[8] When we do this, we say that we have put the expression in

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

a *black box*. Since black boxes are just a kind of constant they require no special treatment.

Note that eval is not simply the composition of value and lift. Consider the expressions

$$\langle\!\langle(\lambda x.\, \lambda y.\, x + y)\cdot 1)\rangle\!\rangle \quad \text{and} \quad \langle\!\langle(\lambda x.\, \lambda y.\, y + x)\cdot 1)\rangle\!\rangle$$

Applying eval to these 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 a term to a term 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

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

This maps n to an expression that sums all the numbers up to n. For example, sum·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 implementation more transparent.

9 Application example

This section presents an example application of *reFLect* in which we develop the core of an LCF-architecture theorem prover (Gordon *et al.*, 1979). The logic of the theorem prover is a variant of higher-order logic. The construction is similar to that of Church's formulation of simple type theory (Church, 1940), except that our basis is *reFLect* rather than λ-calculus. In particular, we follow Harrison's formulation of Church's logic used in the HOL Light system (Harrison, 2000).

The example illustrates how the term manipulation features of *reFLect* make for a short and transparent theorem prover implementation. The code here is similar to the core of the actual *reFLect* theorem prover used at Intel. The `Eval` inference rule illustrates how reflection can be used for efficient reasoning by evaluation. We take the liberty of sugaring the syntax somewhat so that we may reuse without explanation the more mathematical notation already employed in this paper.

We begin by defining universal and existential quantifiers as higher-order functions. These are given the special 'binder' fixity so that, for example, '∀`x. P`' is taken to abbreviate '∀ `(`λ`x. P)`'. Note that, unlike some languages, *reFLect* does not restrict equality to 'equality types' on which equality may be decided. In *reFLect* equality may be used at all types, but a runtime exception occurs if an attempt is made to evaluate an equality that can not be decided. With this relaxation, the definitions of the quantifiers are as follows:

```
let ∀ P = (P = λx. T);
binder ∀;

let ∃ P = ∀ q. (∀x. P x ==> q) ==> q;
binder ∃;
```

We also define some predicates on functions that we will use later to formulate the axiom of infinity.

```
let one_one f = ∀x1 x2. (f x1 = f x2) ==> (x1 = x2);
let onto f = ∀y. ∃x. y = f x;
```

The following module defines the abstract type of theorems and operations on them. Only those identifiers nominated at the end of the module are exported, thus establishing the secure, LCF style, core of the theorem prover.

```
begin_abstype;
  // Theorem data-type and projection functions
  lettype thm = ⊢ (term list) term;
  infix 4 ⊢;

  let asms  (as ⊢ c) = as;
  let concl (as ⊢ c) = c;

  // Inference rules corresponding to reFLect reduction rules
  let Defn k =
    let l = definition k in
    let ph = match_type (type l) (type k) in
      if ¬(well_founded k) then error "Not well founded"
      else [] ⊢ ⟨⟨^k = ^(inst ph l)⟩⟩;
  let Beta tm =
    let ⟨⟨(λ^l. ^m) ^x⟩⟩ = tm in
      if ¬(pattern l) then error "Invalid pattern"
      else if ¬(l ready x) then error "Not ready"
      else let (SOME th) = match l x in
        [] ⊢ ⟨⟨^tm = ^(subst th m)⟩⟩;
  let Gamma tm =
    let ⟨⟨(λ^l. ^m | ^n) ^x⟩⟩ = tm in
      if ¬(pattern l) then error "Invalid pattern"
      else if ¬(l ready x) then error "Not ready"
      else let (SOME th) = match l x in
        [] ⊢ ⟨⟨^tm = ^(subst th m)⟩⟩;
  let Zeta tm =
    let ⟨⟨(λ^l. ^m | ^n) ^x⟩⟩ = tm in
      if ¬(pattern l) then error "Invalid pattern"
      else if  ¬(l ready x) then error "Not ready"
      else let NONE = match l x in
        [] ⊢ ⟪^tm = ^n ^x⟫;
  let Psi tm =
    let (c, ls) = dest_quote tm in
      if ¬(every quote ls) then error "Not ready"
      else [] ⊢ ⟪^tm = ^(fill c ls)⟫;

  // The effect of this rule may be derived by chaining the above rules,
  // but using reflection here gives us proof at the speed of evaluation.
  let Eval tm =  [] ⊢ ⟪^tm = ^(eval tm)⟫;

  // Equality properties
  let Refl tm = [] ⊢ ⟪^tm = ^tm⟫;
  let Apply (as1 ⊢ ⟪^l1 = ^l2⟫) (as2 ⊢ ⟪^m1 = ^m2⟫) =
    as1 U as2 ⊢ ⟪^l1 ^m1 = ^l2 ^m2⟫;
  let Abs1 l (as ⊢ ⟪^m1 = ^m2⟫) =
    if some (free_in l) as then error "Variable free in assumptions"
    else as ⊢ ⟪(λ^l.^m1) = (λ^l. ^m2)⟫;
  let Abs2 l (as1 ⊢ ⟪^m1 = ^m2⟫) (as2 ⊢ ⟪^n1 = ^n2⟫) =
    if some (free_in l) as1 then error "Variable free in assumptions"
    else as1 U as2 ⊢ ⟪(λ^l.^m1 | ^n1) = (λ^l. ^m2 | ^n2)⟫;
  let Eta = [] ⊢ ⟪∀f. (λx. f x) = f⟫;

  // Deduction rules
  let Assume tm = [tm] ⊢ ⟪^tm⦂bool⟫;
  let Eq_Mp (as1 ⊢ ⟪^l1 = ^m⟫) (as2 ⊢ l2) =
    if ¬(l1 alpha l2) then error "Not alpha equivalent"
    else as1 U as2 ⊢ m;
  let Deduct_Antisym_Rule (as1 ⊢ l1) (as2 ⊢ l2) =
    let as1' = filter (λt. ¬(t alpha l2)) as1 in
    let as2' = filter (λt. ¬(t alpha l1)) as2 in
      as1' U as2' ⊢ ⟪^l1 = ^l2⟫;

  // Type and term instantiation
  let Inst ph (as ⊢ l) =  map (inst ph) as ⊢ inst ph l;
  let Subst th (as ⊢ l) = map (subst th) as ⊢ subst th l;

  // Axiom of infinity
  let Infinity =
    [] ⊢ ⟪∃f⦂(num → num). one_one f ∧ ¬(onto f)⟫;
end_abstype
  asms concl Defn Beta Gamma Zeta Psi Eval Refl Apply Abs1 Abs2 Eta
  Assume Eq_Mp Deduct_Antisym_Rule Inst Subst Infinity;
```

The quotation and antiquotation facilities of *reFLect* are used to good effect here to produce a theorem prover core that transparently implements logical rules. The inference rules, for example, use pattern matching on the structure of their premises – making their expected form clearly evident. Likewise, term-splicing is used to construct readable inference rule conclusions. Reflection is used too; in the `Eval` rule, the `eval` function is invoked to implement proof by evaluation. Finally, since the logical core of the theorem prover is itself expressed in *reFLect*, we could contemplate using *reFLect* to formally analyze this core.

10 Related work

The *reFLect* language can been seen as an application-specific contribution to the field of *meta-programming*. In Sheard's taxonomy of meta-programming (Sheard, 2001), *reFLect* 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, *reFLect* has a family resemblance to languages for run-time code generation such as MetaML (Taha & Sheard, 2002) and Template Haskell (Sheard & Peyton Jones, 2002). 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 *reFLect*, 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 given, *reFLect* also differs from MetaML in typing all quotations with a universal type *term*. Template Haskell is similar to *reFLect* 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 *reFLect* is the system described by Shields *et al.* (1998). 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 *reFLect* simple, and the implementation of time critical theorem proving algorithms, like rewriting, efficient.

The *reFLect* language extends the notion of quotation and antiquotation, which have been used for term construction since the LCF system (Gordon *et al.*, 1979), 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 (Aasa *et al.*, 1988) who proposed this mechanism for constructing and destructing object-language expressions within a meta-language. The other reflective languages discussed here (Sheard & Peyton Jones, 2002; Shields *et al.*, 1998; Taha & Sheard, 2002) 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.

The Camlp4 pre-processor-pretty-printer (de Rauglaudre, 2003) implements ideas from Mauny & de Rauglaudre (1994) by adding an initial phase to the Objective Caml compiler (Leroy *et al.*, 2003) in which a system of quotations and antiquotations for expressions in a specified grammar can be used to extend the Objective Caml language. This system for compile time code generation differs from systems like Template Haskell in that it is designed to facilitate syntactic extensions to the host language. Camlp4 is adept at providing support for language embeddings. Shallow embeddings and deep embeddings can both be performed. In the case of a deep embedding, quotations can be used to construct from the concrete syntax of the embedded language Objective Caml expressions with values in a data-type representing the abstract syntax of the embedded language. Quotations can also be used to construct Objective Caml patterns that can be used to match values in the data-type representing the abstract syntax of the embedded language. This application of Camlp4 implements a system like that described by Aasa, Petersson and Synek (Aasa *et al.*, 1988).

The Objective Caml system is itself implemented in Objective Caml, and the interfaces to Objective Caml abstract syntax trees and the compiler are exposed. These interfaces can be combined with the Camlp4 to yield a system with some of the features of *reFLect*. Objective Caml syntax trees are a more complex data-type than those for *reFLect*. They are well suited for their role as the representation of an advanced programming language with an efficient compiler and good error reporting. The abstract syntax trees of *reFLect* are simple, their target application being the expression of a simple and easily manipulated logical calculus. The data-type of Objective Caml abstract syntax trees is not restricted to manipulation in type-safe ways as values of the *reFLect* term type are. Although available for intrepid users, Objective Caml syntax trees are intended to be constructed by the system, before a final type-checking pass confirms their well-formedness. In contrast, *reFLect* terms are optimized for interactive manipulation by users in such a way that they are guaranteed to be well formed at all times. The type-safety of the value mechanism of *reFLect* is not present in the Objective Caml equivalent. This reflects the intended use of these facilities in Objective Caml by language developers. That said, the type safe value mechanism of *reFLect* was influenced by work of the Objective Caml developers on dynamic typing (Leroy & Mauny, 1993).

Lava (Bjesse *et al.*, 1998) is another framework for describing circuits in a functional language, Haskell. In the spirit of the approach pioneered by Sheeran in μFP (Sheeran, 1983), Lava circuit descriptions are built up from primitives using higher-order functions that implement various ways of composing sub-circuits together.

Lava also employs non-standard interpretation to allow the same circuit description to be executed in different ways. For example, under the normal interpretation as a functional program, execution of the circuit is simulation. But under a different interpretation, execution carries out the construction of a circuit net-list. This provides some of the facilities for which reflection is intended in *reFLect*. The net-lists resulting under non-standard interpretation are just data structures – with 'observable sharing' (Claessen & Sands, 1999) – and can be inspected and transformed by other Haskell programs. So Lava circuit descriptions can both be executed and analyzed syntactically, though the latter is possible only at the net-list level.

11 Conclusion

In this paper we presented the language *reFLect*, 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 *reFLect*, as well as using them to describe a method of compiling away the new syntactic features of *reFLect*.

We have completed an implementation of *reFLect* using the compilation technique described to translate *reFLect* into λ-calculus, which is then evaluated using essentially the same combinator compiler and run-time system as the previous *FL* system (Aagaard *et al.*, 1999). The performance of *FL* programs that do not use the new features of *reFLect* has not been impacted.

We have used *reFLect* to implement a mechanized reasoning system based on inspirations from HOL (Gordon & Melham, 1993) and the Forte (Aagaard *et al.*, 1999; Jones *et al.*, 2001) 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 *reFLect* 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 have proved these properties for the *reFLect* language features for expression construction and analysis, though not those that relate to evaluation of expressions (Krstić & Matthews, 2003). Their proofs cover the language presented here up to, but not including, section 7.

Acknowledgements

The authors are grateful to the following people for their helpful discussions on *reFLect* and their comments on an earlier version of this work (Grundy *et al.*, 2003): Rob Gerth, Sava Krstić, Tim Leonard and Carl Seger of Intel; Mike Gordon of the University of Cambridge; Tim Sheard of Portland State University; and John Matthews and Mark Shields of Galois Connections. We would also like to thank the anonymous reviewers of the Journal of Functional Programming for their helpful reviews.

References

Aagaard, M., Jones, R. B., Kaivola, R., Kohatsu, K. R. & Seger, C. (2000) Formal verification of iterative algorithms in microprocessors. *Design Automation: 37th ACM/IEEE conference DAC*, pp. 201–206. ACM.

Aagaard, M. D., Jones, R. B. & Seger, C.-J. H. (1999) *Lifted-FL*: A pragmatic implementation of combined model checking and theorem proving. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C. and Théry, L. (editors), *Theorem Proving in Higher Order Logics: 12th international conference TPHOLs*, pp. 323–340. LNCS, vol. 1690. Springer.

Aasa, A., Petersson, K. & Synek, D. (1988) Concrete syntax for data objects. *LISP and Functional Programming: ACM Conference LFP*, pp. 96–105. ACM.

Barras, B. (2000) Proving and computing in HOL. In: Aagaard, M. and Harrison, J. (editors), *Theorem Proving in Higher Order Logics: 13th International Conference TPHOLs*, pp. 17–37. LNCS, vol. 1869. Springer.

Benaissa, Z. El-Abidine, Moggi, E., Taha, W. & Sheard, T. (1999) Logical modalities and multi-stage programming. *Intuitionsitic Modal Logics and Applications: Federated Logic Conference Satellite Workshop IMLA.*

Berghofer, S. & Nipkow, T. (2000) Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J. and Pollack, R. (editors), *Types for Proofs and Programs: International Workshop, TYPES*, pp. 24–40. LNCS, vol. 2277. Springer.

Bjesse, P., Claessen, K., Sheeran, M. & Singh, S. (1998) Lava: Hardware design in Haskell. *Functional Programming: International Conference ICFP*, pp. 174–184. ACM.

Cardelli, L. & Wegner, P. (1985) On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.* **17**(4), 471–522.

Church, A. (1940). A formulation of the simple theory of types. *J. Ssymbolic Logic*, **5**, 56–68.

Claessen, K. & Sands, D. (1999) Observable sharing for functional circuit description. *Advances in Computing Science: 5th Asian Computing Science Conference ASIAN*, pp. 62–73. Springer.

Coquand, T. (1986) An analysis of Girard's paradox. *Logic in Computer Science: 1st IEEE Symposium LICS*, pp. 227–236. IEEE.

Coquand, T. (1991) A new paradox in type theory. In: Prawitz, D., Skyrms, B. and Westerståhl, D. (editors), *Logic, Methodology and Philosophy of Science: 9th International Congress*, pp. 555–570. Studies in Logic and the Foundations of Mathematics, vol. 134. North-Holland.

de Rauglaudre, D. (2003) *Camlp4: Reference manual.* 3.07 edn. INRIA.

Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C. & Werner, B. (1993) *The Coq Proof Assistant User's Guide.* Rapport technique, Programme 2 154. INRIA.

Girard, J.-Y., Lafont, Y. & Taylor, P. (1989) *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science, vol. 7. Cambridge University.

Gordon, M. J., Milner, A. J. & Wadsworth, C. P. (1979) *Edinburgh LCF: A Mechanised Logic of Computation*. LNCS, vol. 78. Springer.

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

Gordon, M. J. C. & Melham, T. F. (editors) (1993) *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University.

Greve, D., Wilding, M. & Hardin, D. (2000) High-speed, analyzable simulators. *In:* Kaufmann, M., Manolios, P. and Moore, J. S. (editorss), *Computer-aided Reasoning: ACL2 case studies*. Kluwer.

Grundy, J., Melham, T. & O'Leary, J. (2003) *A reflective functional language for hardware design and theorem proving*. Technical report PRG-RR-03-16, University of Oxford, Computing Laboratory.

Harper, R., MacQueen, D. & Milner, R. (1986) *Standard ML*. Report 86-2, University of Edinburgh, Laboratory for Foundations of Computer Science.

Harrison, J. (1995) *Metatheory and reflection in theorem proving: A survey and critique*. Technical Report CRC-053, SRI Cambridge.

Harrison, J. (2000) *The HOL Light manual*. 1.1 edn. University of Cambridge, Computer Laboratory.

Johnson, S. D. (1984) *Synthesis of Digital Designs from Recursion Equations*. MIT.

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

Kaivola, R., & Aagaard, M. (2000) Divider circuit verification with model checking and theorem proving. In: Aagaard, M. and Harrison, J. (editors), *Theorem Proving in Higher Order Logics: 13th International Conference TPHOLs*, pp. 338–355. LNCS, vol. 1869. Springer.

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

Kaivola, R. & Narasimhan, N. (2001) Formal verification of the Pentium® 4 multiplier. *High-level Design Validation and Test: 6th International Workshop HLDVT*, pp. 115–122.

Kaufmann, M., Manolios, P. & Moore, J. S. (editors) (2000a) *Computer-aided Reasoning: ACL2 case studies*. Kluwer.

Kaufmann, M., Manolios, P. & Moore, J. S. (2000b) *Computer-aided Reasoning: An approach*. Kluwer.

Krstić, S. & Matthews, J. (2003) *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.

Leisenring, A. C. (1969) *Mathematical Logic and Hilbert's ε-symbol*. Macdonald.

Leroy, X. & Mauny, M. (1993) Dynamics in ML. *J. Funct. Program.* **3**(4), 431–463.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D. & Vouillon, J. (2003) *The Objective Caml System: Documentation and user's manual*. 3.07 edn. INRIA.

Matthews, J., Cook, B. & Launchbury, J. (1998) Microprocessor specification in Hawk. *Computer Languages: International Conference*, pp. 90–101. IEEE.

Mauny, M. & de Rauglaudre, D. (1994) A complete and realistic implementation of quotations for ML. *ML and its Applications: ACM-SIGPLAN Workshop ML*, pp. 70–78.

Melham, T. F. (1993) *Higher Order Logic and Hardware Verification*. Cambridge University.

Moore, J. S. (1998) Symbolic simulation: An ACL2 approach. In: Gopalakrishnan, G. and Windley, P. (editors), *Formal Methods in Computer-aided Design: 2nd International Conference FMCAD*, pp. 334–350. LNCS, vol. 1522. Springer.

Müller, O. & Slind, K. (1997) Treating partiality in a logic of total functions. *The Comput. J.* **40**(10), 640–651.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002) *Isabelle/HOL: A proof assistant for higher-order logic*. LNCS, vol. 2283. Springer.

Paulson, L. C. (1983) A higher-order implementation of rewriting. *Sci. Comput. program.* **3**(2), 119–149.

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

Sheard, T. (2001) Accomplishments and research challenges in meta-programming. In: Taha, W. (eitor), *Semantics, Applications, and Implementation of Program Generation: 2nd International Workshop SAIG*, pp. 2–44. LNCS, vol. 2196. Springer.

Sheard, T. & Peyton Jones, S. (2002) Template meta-programming for Haskell. *Haskell: Workshop*, pp. 1–16. ACM.

Sheeran, M. (1983) *μFP: An algebraic VLSI design language*. PhD thesis, University of Oxford.

Shields, M., Sheard, T. & Peyton Jones, S. (1998) Dynamic typing as staged type inference. *Principles of Programming Language: 25th Annual Symposium POPL*, pp. 289–302.

Spirakis, G. (2003) Leading-edge and future design challenges: Is the classical EDA ready? *Design Automation: 40th ACM/IEEE Conference DAC*, p. 416. ACM.

Stump, A. (1999) *On Coquand's "An analysis of Girard's paradox"*. MTC qualifying exam, Stanford University, Department of Computer Science.

Suppes, P. C. (1957) *Introduction to Logic*. Van Nostrand Reinhold.

Taha, W. (1999) *Multi-stage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology.

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