HOL Light QE[⋆]

Jacques Carette, William M. Farmer, and Patrick Laskowski

Computing and Software, McMaster University, Canada
`http://www.cas.mcmaster.ca/~carette`
`http://imps.mcmaster.ca/wmfarmer`

1 February 2018

Abstract. We are interested in algorithms that manipulate mathematical expressions in mathematically meaningful ways. Expressions are syntactic, but most logics do not allow one to discuss syntax. CTT_{qe} is a version of Church's type theory that includes quotation and evaluation operators, akin to quote and eval in the Lisp programming language. Since the HOL logic is also a version of Church's type theory, we decided to add quotation and evaluation to HOL Light to demonstrate the implementability of CTT_{qe} and the benefits of having quotation and evaluation in a proof assistant. The resulting system is called HOL Light QE. Here we document the design of HOL Light QE and the challenges that needed to be overcome. The resulting implementation is freely available.

1 Introduction

A *syntax-based mathematical algorithm (SBMA)* manipulates mathematical expressions in a meaningful way. SBMAs are commonplace in mathematics. Examples include algorithms that compute arithmetic operations by manipulating numerals, linear transformations by manipulating matrices, and derivatives by manipulating functional expressions. Reasoning about the mathematical meaning of an SBMA requires reasoning about the relationship between how the expressions are manipulated by the SBMA and what the manipulations mean.

We argue in [24] that the combination of quotation and evaluation, along with appropriate inference rules, provides the means to reason about the interplay between syntax and semantics, which is what is needed for reasoning about SBMAs. *Quotation* is an operation that maps an expression e to a special value called a *syntactic value* that represents the syntax tree of e. Quotation enables expressions to be manipulated as syntactic entities. *Evaluation* is an operation that maps a syntactic value s to the value of the expression that is represented by s. Evaluation enables meta-level reasoning via syntactic values to be reflected into object-level reasoning. Quotation and evaluation thus form an infrastructure for integrating meta-level and object-level reasoning. Quotation gives a form of *reification* of object-level values which allows introspection. Along with inference

⋆ This research was supported by NSERC.

rules, this gives a certain amount of *logical reflection*; evaluation adds to this some aspects of *computational reflection* [21,34].

Incorporating quotation and evaluation operators — like quote and eval in the Lisp programming language — into a traditional logic like first-order logic or simple type theory is not a straightforward task. Several challenging design problems stand in the way. The three design problems that most concern us are the following. We will write the quotation and evaluation operators applied to an expression e as $\ulcorner e \urcorner$ and $[\![e]\!]$, respectively.

1. *Evaluation Problem.* An evaluation operator is applicable to syntactic values that represent formulas and thus is effectively a truth predicate. Hence, by the proof of Tarski's theorem on the undefinability of truth [50], if the evaluation operator is total in the context of a sufficiently strong theory (like first-order Peano arithmetic), then it is possible to express the liar paradox. Therefore, the evaluation operator must be partial and the law of disquotation cannot hold universally (i.e., for some expressions e, $[\![\ulcorner e \urcorner]\!] \neq e$). As a result, reasoning with evaluation can be cumbersome and leads to undefined expressions.
2. *Variable Problem.* The variable x is not free in the expression $\ulcorner x + 3 \urcorner$ (or in any quotation). However, x is free in $[\![\ulcorner x + 3 \urcorner]\!]$ because $[\![\ulcorner x + 3 \urcorner]\!] = x + 3$. If the value of a constant c is $\ulcorner x + 3 \urcorner$, then x is free in $[\![c]\!]$ because $[\![c]\!] = [\![\ulcorner x + 3 \urcorner]\!] = x + 3$. Hence, in the presence of an evaluation operator, whether or not a variable is free in an expression may depend on the values of the expression's components. As a consequence, the substitution of an expression for the free occurrences of a variable in another expression depends on the semantics (as well as the syntax) of the expressions involved and must be integrated with the proof system for the logic. That is, a logic with quotation and evaluation requires a semantics-dependent form of substitution in which side conditions, like whether a variable is free in an expression, are proved within the proof system. This is a major departure from traditional logic.
3. *Double Substitution Problem.* By the semantics of evaluation, the value of $[\![e]\!]$ is the *value* of the expression whose syntax tree is represented by the *value* of e. Hence the semantics of evaluation involves a double valuation. This is most apparent when the value of a variable involves a syntax tree that refers to the name of that same variable. For example, if the value of a variable x is $\ulcorner x \urcorner$, then $[\![x]\!] = [\![\ulcorner x \urcorner]\!] = x = \ulcorner x \urcorner$. Hence the substitution of $\ulcorner x \urcorner$ for x in $[\![x]\!]$ requires one substitution inside the argument of the evaluation operator and another substitution after the evaluation operator is eliminated. This double substitution is another major departure from traditional logic.

$\textsc{ctt}_{\mathrm{qe}}$ [25,26] is version of Church's type theory [16] with quotation and evaluation that solves these three design problems. It is based on $\mathcal{Q}_0$ [2], Peter Andrews' version of Church's type theory. We believe $\textsc{ctt}_{\mathrm{qe}}$ is the first readily implementable version of simple type theory that includes *global* quotation and evaluation operators. We show in [25] that it is suitable for defining, applying, and reasoning about SBMAs.

To demonstrate that $\textsc{ctt}_{\text{qe}}$ is indeed implementable, we have done so by modifying HOL Light [35], a compact implementation of the HOL proof assistant [32]. The resulting version of HOL Light is called HOL Light QE. Here we present its design, implementation, and the challenges encountered.

The rest of the paper is organized as follows. Section 2 presents the key ideas underlying $\textsc{ctt}_{\text{qe}}$ and explains how $\textsc{ctt}_{\text{qe}}$ solves the three design problems. Section 3 offers a brief overview of HOL Light. The HOL Light QE implementation is described in section 4, and examples of how quotation and evaluation are used in it are discussed in section 5. Section 6 is devoted to related work. And the paper ends with some final remarks including a brief discussion on future work.

The major contributions of the work presented here are:

1. We show that the logical machinery for quotation and evaluation embodied in $\textsc{ctt}_{\text{qe}}$ can be straightforwardly implemented by modifying HOL Light.
2. We produce an HOL-style proof assistant with a built-in global reflection infrastructure for defining, applying, and proving properties about SBMAs.
3. We demonstrate how this reflection infrastructure can be used to express formula schemas, such as the induction schema for first-order Peano arithmetic, as single formulas.

2 CTT$_{\text{qe}}$

The syntax, semantics and proof system of $\textsc{ctt}_{\text{qe}}$ are defined in [25]. Here we will only introduce the definitions and results of that are key to understanding how HOL Light QE implements $\textsc{ctt}_{\text{qe}}$. The reader is encouraged to consult [25] when additional details are required.

2.1 Syntax

$\textsc{ctt}_{\text{qe}}$ has the same machinery as $\mathcal{Q}_0$ plus an inductive type ϵ of syntactic values, a partial quotation operator, and a typed evaluation operator.

A *type* of $\textsc{ctt}_{\text{qe}}$ is defined inductively by the following formation rules:

1. *Type of individuals*: ι is a type.
2. *Type of truth values*: o is a type.
3. *Type of constructions*: ϵ is a type.
4. *Function type*: If α and β are types, then $(\alpha \to \beta)$ is a type.

Let $\mathcal{T}$ denote the set of types of $\textsc{ctt}_{\text{qe}}$. A *typed symbol* is a symbol with a subscript from $\mathcal{T}$. Let $\mathcal{V}$ be a set of typed symbols such that, for each $\alpha \in \mathcal{T}$, $\mathcal{V}$ contains denumerably many typed symbols with subscript α. A *variable of type* α of $\textsc{ctt}_{\text{qe}}$ is a member of $\mathcal{V}$ with subscript α. $\mathbf{x}_\alpha, \mathbf{y}_\alpha, \mathbf{z}_\alpha, \ldots$ are syntactic variables ranging over variables of type α. Let $\mathcal{C}$ be a set of typed symbols disjoint from $\mathcal{V}$. A *constant of type* α of $\textsc{ctt}_{\text{qe}}$ is a member of $\mathcal{C}$ with subscript α. $\mathbf{c}_\alpha, \mathbf{d}_\alpha, \ldots$ are syntactic variables ranging over constants of type α. $\mathcal{C}$ contains a set of *logical constants* that include $\mathsf{app}_{\epsilon \to \epsilon \to \epsilon}$, $\mathsf{abs}_{\epsilon \to \epsilon \to \epsilon}$, and $\mathsf{quo}_{\epsilon \to \epsilon}$.

An *expression of type* α of $\textsc{ctt}_{\text{qe}}$ is defined inductively by the formation rules below. $\mathbf{A}_\alpha, \mathbf{B}_\alpha, \mathbf{C}_\alpha, \ldots$ are syntactic variables ranging over expressions of type α. An expression is *eval-free* if it is constructed using just the first five rules.

1. *Variable*: $\mathbf{x}_\alpha$ is an expression of type α.
2. *Constant*: $\mathbf{c}_\alpha$ is an expression of type α.
3. *Function application*: $(\mathbf{F}_{\alpha\to\beta}\,\mathbf{A}_\alpha)$ is an expression of type β.
4. *Function abstraction*: $(\lambda\,\mathbf{x}_\alpha\,.\,\mathbf{B}_\beta)$ is an expression of type $\alpha \to \beta$.
5. *Quotation*: $\ulcorner\mathbf{A}_\alpha\urcorner$ is an expression of type ϵ if $\mathbf{A}_\alpha$ is eval-free.
6. *Evaluation*: $[\![\mathbf{A}_\epsilon]\!]_{\mathbf{B}_\beta}$ is an expression of type β.

The sole purpose of the second component $\mathbf{B}_\beta$ in an evaluation $[\![\mathbf{A}_\epsilon]\!]_{\mathbf{B}_\beta}$ is to establish the type of the evaluation; we will thus write $[\![\mathbf{A}_\epsilon]\!]_{\mathbf{B}_\beta}$ as $[\![\mathbf{A}_\epsilon]\!]_\beta$.

A *construction* of $\textsc{ctt}_{\text{qe}}$ is an expression of type ϵ defined inductively by:

1. $\ulcorner\mathbf{x}_\alpha\urcorner$ is a construction.
2. $\ulcorner\mathbf{c}_\alpha\urcorner$ is a construction.
3. If $\mathbf{A}_\epsilon$ and $\mathbf{B}_\epsilon$ are constructions, then $\mathsf{app}_{\epsilon\to\epsilon\to\epsilon}\,\mathbf{A}_\epsilon\,\mathbf{B}_\epsilon$, $\mathsf{abs}_{\epsilon\to\epsilon\to\epsilon}\,\mathbf{A}_\epsilon\,\mathbf{B}_\epsilon$, and $\mathsf{quo}_{\epsilon\to\epsilon}\,\mathbf{A}_\epsilon$ are constructions.

The set of constructions is thus an inductive type whose base elements are quotations of variables and constants, and whose constructors are $\mathsf{app}_{\epsilon\to\epsilon\to\epsilon}$, $\mathsf{abs}_{\epsilon\to\epsilon\to\epsilon}$, and $\mathsf{quo}_{\epsilon\to\epsilon}$. As we will see shortly, constructions serve as syntactic values.

Let $\mathcal{E}$ be the function mapping eval-free expressions to constructions that is defined inductively as follows:

1. $\mathcal{E}(\mathbf{x}_\alpha) = \ulcorner\mathbf{x}_\alpha\urcorner$.
2. $\mathcal{E}(\mathbf{c}_\alpha) = \ulcorner\mathbf{c}_\alpha\urcorner$.
3. $\mathcal{E}(\mathbf{F}_{\alpha\to\beta}\,\mathbf{A}_\alpha) = \mathsf{app}_{\epsilon\to\epsilon\to\epsilon}\,\mathcal{E}(\mathbf{F}_{\alpha\to\beta})\,\mathcal{E}(\mathbf{A}_\alpha)$.
4. $\mathcal{E}(\lambda\,\mathbf{x}_\alpha\,.\,\mathbf{B}_\beta) = \mathsf{abs}_{\epsilon\to\epsilon\to\epsilon}\,\mathcal{E}(\mathbf{x}_\alpha)\,\mathcal{E}(\mathbf{B}_\beta)$.
5. $\mathcal{E}(\ulcorner\mathbf{A}_\alpha\urcorner) = \mathsf{quo}_{\epsilon\to\epsilon}\,\mathcal{E}(\mathbf{A}_\alpha)$.

When $\mathbf{A}_\alpha$ is eval-free, $\mathcal{E}(\mathbf{A}_\alpha)$ is the unique construction that represents the syntax tree of $\mathbf{A}_\alpha$. That is, $\mathcal{E}(\mathbf{A}_\alpha)$ is a syntactic value that represents how $\mathbf{A}_\alpha$ is syntactically constructed. For every eval-free expression, there is a construction that represents its syntax tree, but not every construction represents the syntax tree of an eval-free expression. For example, $\mathsf{app}_{\epsilon\to\epsilon\to\epsilon}\,\ulcorner\mathbf{x}_\alpha\urcorner\ulcorner\mathbf{x}_\alpha\urcorner$ represents the syntax tree of $(\mathbf{x}_\alpha\,\mathbf{x}_\alpha)$ which is not an expression of $\textsc{ctt}_{\text{qe}}$ since the types are mismatched. A construction is *proper* if it is in the range of $\mathcal{E}$, i.e., it represents the syntax tree of an eval-free expression.

The purpose of $\mathcal{E}$ is to define the semantics of quotation: the meaning of $\ulcorner\mathbf{A}_\alpha\urcorner$ is $\mathcal{E}(\mathbf{A}_\alpha)$.

2.2 Semantics

The semantics of $\textsc{ctt}_{\text{qe}}$ is based on Henkin-style general models [36]. An expression $\mathbf{A}_\epsilon$ of type ϵ denotes a construction, and when $\mathbf{A}_\epsilon$ is a construction, it denotes itself. The semantics of the quotation and evaluation operators are defined so that the following two theorems hold:

Theorem 2.21 (Law of Quotation) $\ulcorner\boldsymbol{A}_\alpha\urcorner = \mathcal{E}(\boldsymbol{A}_\alpha)$ *is valid in* $\textsc{ctt}_{\text{qe}}$.

Corollary 2.22 $\ulcorner \boldsymbol{A}_\alpha \urcorner = \ulcorner \boldsymbol{B}_\alpha \urcorner$ *iff* $\boldsymbol{A}_\alpha$ *and* $\boldsymbol{B}_\alpha$ *are identical expressions.*

Theorem 2.23 (Law of Disquotation) $[\![\ulcorner \boldsymbol{A}_\alpha \urcorner]\!]_\alpha = \boldsymbol{A}_\alpha$ *is valid in* $\textsc{ctt}_{\rm qe}$.

Remark 2.24 Notice that this is not the full Law of Disquotation, since only eval-free expressions can be quoted. As a result of this restriction, the liar paradox is not expressible in $\textsc{ctt}_{\rm qe}$ and the Evaluation Problem mentioned above is effectively solved.

2.3 Quasiquotation

Quasiquotation is a parameterized form of quotation in which the parameters serve as holes in a quotation that are filled with expressions that denote syntactic values. It is a very powerful syntactic device for specifying expressions and defining macros. Quasiquotation was introduced by Willard Van Orman Quine in 1940 in the first version of his book *Mathematical Logic* [48]. It has been extensively employed in the Lisp family of programming languages [4],[1] and from there to other families of programming languages, most notably the ML family.

In $\textsc{ctt}_{\rm qe}$, constructing a large quotation from smaller quotations can be tedious because it requires many applications of the syntax constructors $\mathsf{app}_{\epsilon\to\epsilon\to\epsilon}$, $\mathsf{abs}_{\epsilon\to\epsilon\to\epsilon}$, and $\mathsf{quo}_{\epsilon\to\epsilon}$. Quasiquotation alleviates this problem. It can be defined straightforwardly in $\textsc{ctt}_{\rm qe}$. However, quasiquotation is not part of the official syntax of $\textsc{ctt}_{\rm qe}$; it is just a notational device used to write $\textsc{ctt}_{\rm qe}$ expressions in a compact form.

As an example, consider $\ulcorner \neg(\mathbf{A}_o \wedge \lfloor \mathbf{B}_\epsilon \rfloor)\urcorner$. Here $\lfloor \mathbf{B}_\epsilon \rfloor$ is a *hole* or *antiquotation.* Assume that $\mathbf{A}_o$ contains no holes. $\ulcorner \neg(\mathbf{A}_o \wedge \lfloor \mathbf{B}_\epsilon \rfloor)\urcorner$ is then an abbreviation for the verbose expression

$$\mathsf{app}_{\epsilon\to\epsilon\to\epsilon} \ulcorner \neg_{o\to o}\urcorner\, (\mathsf{app}_{\epsilon\to\epsilon\to\epsilon}\, (\mathsf{app}_{\epsilon\to\epsilon\to\epsilon} \ulcorner \wedge_{o\to o\to o}\urcorner \ulcorner \mathbf{A}_o \urcorner)\, \mathbf{B}_\epsilon).$$

$\ulcorner \neg(\mathbf{A}_o \wedge \lfloor \mathbf{B}_\epsilon \rfloor)\urcorner$ represents the the syntax tree of a negated conjunction in which the part of the tree corresponding to the second conjunct is replaced by the syntax tree represented by $\mathbf{B}_\epsilon$. If $\mathbf{B}_\epsilon$ is a quotation $\ulcorner \mathbf{C}_o \urcorner$, then the quasiquotation $\ulcorner \neg(\mathbf{A}_o \wedge \lfloor \ulcorner \mathbf{C}_o \urcorner \rfloor)\urcorner$ is *equivalent* to the quotation $\ulcorner \neg(\mathbf{A}_o \wedge \mathbf{C}_o)\urcorner$.

2.4 Proof System

The proof system for $\textsc{ctt}_{\rm qe}$ consists of the axioms for $\mathcal{Q}_0$, the single rule of inference for $\mathcal{Q}_0$, and additional axioms [25, B1–B13] that define the logical constants of $\textsc{ctt}_{\rm qe}$ (B1–B4,B5,B7), specify ϵ as an inductive type (B4,B6), state the properties of quotation and evaluation (B8,B10), and extend the rules for beta-reduction (B9,B11–13). We prove in [25] that this proof system is sound for all formulas and complete for eval-free formulas.

The axioms that express the properties of quotation and evaluation are:

[1] In Lisp, the standard symbol for quasiquotation is the backquote (‘) symbol, and thus in Lisp, quasiquotation is usually called *backquote*.

B8 (Properties of Quotation)

1. $\ulcorner \mathbf{F}_{\alpha\rightarrow\beta}\,\mathbf{A}_\alpha \urcorner = \mathsf{app}_{\epsilon\rightarrow\epsilon\rightarrow\epsilon} \ulcorner \mathbf{F}_{\alpha\rightarrow\beta} \urcorner \ulcorner \mathbf{A}_\alpha \urcorner$.
2. $\ulcorner \lambda\, \mathbf{x}_\alpha \,.\, \mathbf{B}_\beta \urcorner = \mathsf{abs}_{\epsilon\rightarrow\epsilon\rightarrow\epsilon} \ulcorner \mathbf{x}_\alpha \urcorner \ulcorner \mathbf{B}_\beta \urcorner$.
3. $\ulcorner \ulcorner \mathbf{A}_\alpha \urcorner \urcorner = \mathsf{quo}_{\epsilon\rightarrow\epsilon} \ulcorner \mathbf{A}_\alpha \urcorner$.

B10 (Properties of Evaluation)

1. $[\![\ulcorner \mathbf{x}_\alpha \urcorner]\!]_\alpha = \mathbf{x}_\alpha$.
2. $[\![\ulcorner \mathbf{c}_\alpha \urcorner]\!]_\alpha = \mathbf{c}_\alpha$.
3. $(\mathsf{is\text{-}expr}^{\alpha\rightarrow\beta}_{\epsilon\rightarrow o}\,\mathbf{A}_\epsilon \wedge \mathsf{is\text{-}expr}^{\alpha}_{\epsilon\rightarrow o}\,\mathbf{B}_\epsilon) \supset [\![\mathsf{app}_{\epsilon\rightarrow\epsilon\rightarrow\epsilon}\,\mathbf{A}_\epsilon\,\mathbf{B}_\epsilon]\!]_\beta = [\![\mathbf{A}_\epsilon]\!]_{\alpha\rightarrow\beta}\,[\![\mathbf{B}_\epsilon]\!]_\alpha$.
4. $(\mathsf{is\text{-}expr}^{\beta}_{\epsilon\rightarrow o}\,\mathbf{A}_\epsilon \wedge \neg(\mathsf{is\text{-}free\text{-}in}_{\epsilon\rightarrow\epsilon\rightarrow o} \ulcorner \mathbf{x}_\alpha \urcorner \ulcorner \mathbf{A}_\epsilon \urcorner)) \supset$
   $[\![\mathsf{abs}_{\epsilon\rightarrow\epsilon\rightarrow\epsilon} \ulcorner \mathbf{x}_\alpha \urcorner\,\mathbf{A}_\epsilon]\!]_{\alpha\rightarrow\beta} = \lambda\,\mathbf{x}_\alpha \,.\, [\![\mathbf{A}_\epsilon]\!]_\beta$.
5. $\mathsf{is\text{-}expr}^{\epsilon}_{\epsilon\rightarrow o}\,\mathbf{A}_\epsilon \supset [\![\mathsf{quo}_{\epsilon\rightarrow\epsilon}\,\mathbf{A}_\epsilon]\!]_\epsilon = \mathbf{A}_\epsilon$.

The axioms for extending the rules for beta-reduction are:

B9 (Beta-Reduction for Quotations)

$$(\lambda\,\mathbf{x}_\alpha \,.\, \ulcorner \mathbf{B}_\beta \urcorner)\,\mathbf{A}_\alpha) = \ulcorner \mathbf{B}_\beta \urcorner.$$

B11 (Beta-Reduction for Evaluations)

1. $(\lambda\,\mathbf{x}_\alpha \,.\, [\![\mathbf{B}_\epsilon]\!]_\beta)\,\mathbf{x}_\alpha = [\![\mathbf{B}_\epsilon]\!]_\beta$.
2. $(\mathsf{is\text{-}expr}^{\beta}_{\epsilon\rightarrow o}\,((\lambda\,\mathbf{x}_\alpha \,.\, \mathbf{B}_\epsilon)\,\mathbf{A}_\alpha) \wedge \neg(\mathsf{is\text{-}free\text{-}in}_{\epsilon\rightarrow\epsilon\rightarrow o} \ulcorner \mathbf{x}_\alpha \urcorner\,((\lambda\,\mathbf{x}_\alpha \,.\, \mathbf{B}_\epsilon)\,\mathbf{A}_\alpha))) \supset$
   $(\lambda\,\mathbf{x}_\alpha \,.\, [\![\mathbf{B}_\epsilon]\!]_\beta)\,\mathbf{A}_\alpha = [\![(\lambda\,\mathbf{x}_\alpha \,.\, \mathbf{B}_\epsilon)\,\mathbf{A}_\alpha]\!]_\beta$.

B12 ("Not Free In" means "Not Effective In")

$\neg\mathsf{IS\text{-}EFFECTIVE\text{-}IN}(\mathbf{x}_\alpha, \mathbf{B}_\beta)$
where $\mathbf{B}_\beta$ is eval-free and $\mathbf{x}_\alpha$ is not free in $\mathbf{B}_\beta$.

B13 (Beta-Reduction for Function Abstractions)

$(\neg\mathsf{IS\text{-}EFFECTIVE\text{-}IN}(\mathbf{y}_\beta, \mathbf{A}_\alpha) \vee \neg\mathsf{IS\text{-}EFFECTIVE\text{-}IN}(\mathbf{x}_\alpha, \mathbf{B}_\gamma)) \supset$
$(\lambda\,\mathbf{x}_\alpha \,.\, \lambda\,\mathbf{y}_\beta \,.\, \mathbf{B}_\gamma)\,\mathbf{A}_\alpha = \lambda\,\mathbf{y}_\beta \,.\, ((\lambda\,\mathbf{x}_\alpha \,.\, \mathbf{B}_\gamma)\,\mathbf{A}_\alpha)$
where $\mathbf{x}_\alpha$ and $\mathbf{y}_\beta$ are distinct.

Substitution is performed using the properties of beta-reduction as Andrews does in the proof system for $\mathcal{Q}_0$ [2, p. 213]. The following three beta-reduction cases require discussion:

1. $(\lambda\,\mathbf{x}_\alpha \,.\, \lambda\,\mathbf{y}_\beta \,.\, \mathbf{B}_\gamma)\,\mathbf{A}_\alpha$ where $\mathbf{x}_\alpha$ and $\mathbf{y}_\beta$ are distinct.
2. $(\lambda\,\mathbf{x}_\alpha \,.\, \ulcorner \mathbf{B}_\beta \urcorner)\,\mathbf{A}_\alpha$.
3. $(\lambda\,\mathbf{x}_\alpha \,.\, [\![\mathbf{B}_\epsilon]\!]_\beta)\,\mathbf{A}_\alpha$.

The first case can normally be reduced when either (1) $\mathbf{y}_\beta$ is not free in $\mathbf{A}_\alpha$ or (2) $\mathbf{x}_\alpha$ is not free in $\mathbf{B}_\gamma$. However, due to the Variable Problem mentioned before, it is only possible to syntactically check whether a "variable is not free in an expression" when the expression is eval-free. Our solution is to replace the syntactic notion of "a variable is free in an expression" by the semantic notion of "a variable is effective in an expression" when the expression is not necessarily eval-free, and use Axiom B13 to perform the beta-reduction.

"$\mathbf{x}_\alpha$ is effective in $\mathbf{B}_\beta$" means the value of $\mathbf{B}_\beta$ depends on the value of $\mathbf{x}_\alpha$. Clearly, if $\mathbf{B}_\beta$ is eval-free, "$\mathbf{x}_\alpha$ is effective in $\mathbf{B}_\beta$" implies "$\mathbf{x}_\alpha$ is free in $\mathbf{B}_\beta$". However, "$\mathbf{x}_\alpha$ is effective in B_β" is a refinement of "$\mathbf{x}_\alpha$ is free in $\mathbf{B}_\beta$" on eval-free expressions since $\mathbf{x}_\alpha$ is free in $\mathbf{x}_\alpha = \mathbf{x}_\alpha$, but $\mathbf{x}_\alpha$ is not effective in $\mathbf{x}_\alpha = \mathbf{x}_\alpha$. "$\mathbf{x}_\alpha$ is effective in $\mathbf{B}_\beta$" is expressed in CTT$_{\text{qe}}$ as $\mathsf{IS\text{-}EFFECTIVE\text{-}IN}(\mathbf{x}_\alpha, \mathbf{B}_\beta)$, an abbreviation for

$$\exists\, \mathbf{y}_\alpha \,.\, ((\lambda\, \mathbf{x}_\alpha \,.\, \mathbf{B}_\beta)\, \mathbf{y}_\alpha \neq \mathbf{B}_\beta)$$

where $\mathbf{y}_\alpha$ is any variable of type α that differs from $\mathbf{x}_\alpha$.

The second case is simple since a quotation cannot be modified by substitution — it is effectively the same as a constant. Thus beta-reduction is performed without changing $\ulcorner \mathbf{B}_\beta \urcorner$ as shown in Axiom B9 above.

The third case is handled by Axioms B11.1 and B11.2. B11.1 deals with the trivial case when $\mathbf{A}_\alpha$ is the bound variable $\mathbf{x}_\alpha$ itself. B11.2 deals with the other much more complicated situation. The condition

$$\neg(\mathsf{is\text{-}free\text{-}in}_{\epsilon\to\epsilon\to o} \ulcorner \mathbf{x}_\alpha \urcorner\, ((\lambda\, \mathbf{x}_\alpha \,.\, \mathbf{B}_\epsilon)\, \mathbf{A}_\alpha))$$

guarantees that there is no *double substitution*. $\mathsf{is\text{-}free\text{-}in}_{\epsilon\to\epsilon\to o}$ is a logical constant of CTT$_{\text{qe}}$ such that $\mathsf{is\text{-}free\text{-}in}_{\epsilon\to\epsilon\to o} \ulcorner \mathbf{x}_\alpha \urcorner \ulcorner \mathbf{B}_\beta \urcorner$ says that the variable $\mathbf{x}_\alpha$ is free in the (eval-free) expression $\mathbf{B}_\beta$.

Thus we see that substitution in CTT$_{\text{qe}}$ in the presence of evaluations may require proving semantic side conditions of the following two forms:

1. $\neg\mathsf{IS\text{-}EFFECTIVE\text{-}IN}(\mathbf{x}_\alpha, \mathbf{B}_\beta)$.
2. $\mathsf{is\text{-}free\text{-}in}_{\epsilon\to\epsilon\to o} \ulcorner \mathbf{x}_\alpha \urcorner \ulcorner \mathbf{B}_\beta \urcorner$.

2.5 The Three Design Problems

To recap, CTT$_{\text{qe}}$ solves the three design problems given in section 1. The Evaluation Problem is avoided by restricting the quotation operator to eval-free expressions and thus making it impossible to express the liar paradox. The Variable Problem is overcome by modifying Andrews' beta-reduction axioms. The Double Substitution Problem is eluded by using a beta-reduction axiom for evaluations that excludes beta-reductions that embody a double substitution.

3 HOL Light

HOL Light [35] is an open-source proof assistant developed by John Harrison. It implements a logic (HOL) which is a version of Church's type theory. It is a simple implementation of the HOL proof assistant [32] written in OCaml and hosted on GitHub at `https://github.com/jrh13/hol-light/`. Although it is a relatively small system, it has been used to formalize many kinds of mathematics and to check many proofs including the lion's share of Tom Hale's proof of the Kepler conjecture [23].

HOL Light is very well suited to serve as a foundation on which to build an implementation of CTT_{qe}: First, it is an open-source system that can be freely modified as long as certain very minimal conditions are satisfied. Second, it is an implementation of a version of simple type theory that is essentially $\mathcal{Q}_0$, the version of Church's type theory underlying CTT_{qe}, plus (1) polymorphic type variables, (2) an axiom of choice expressed by asserting that the Hilbert ϵ operator is a choice (indefinite description) operator, and (3) an axiom of infinity that asserts that `ind`, the type of individuals, is infinite [35]. The type variables in the implemented logic are not a hindrance; they actually facilitate the implementation of CTT_{qe}. The presence of the axioms of choice and infinity in HOL Light alter the semantics of CTT_{qe} without compromising in any way the semantics of quotation and evaluation. And third, HOL Light supports the definition of inductive types so that ϵ can be straightforwardly defined.

4 Implementation

4.1 Overview

HOL Light QE was implemented in four stages:

1. The set of terms was extended so that CTT_{qe} expressions could be mapped to HOL Light terms. This required the introduction of `epsilon`, the type of constructions, and term constructors for quotations and evaluations. See subsection 4.2.
2. The proof system was modified to include the machinery in CTT_{qe} for reasoning about quotations and evaluations. This required adding new rules of inference and modifying the `INST` rule of inference that simultaneously substitutes terms $t_1, \ldots, t_n$ for the free variables $x_1, \ldots, x_n$ in a sequent. See subsection 4.3.
3. Machinery — consisting of HOL function definitions, tactics, and theorems — was created for supporting reasoning about quotations and evaluations in the new system. See subsection 4.4.
4. Examples were developed in the new system to test the implementation and to demonstrate the benefits of having quotation and evaluation in higher-order logic. See section 5.

The first and second stages have been completed; both stages involved modifying the kernel of HOL Light. The third stage is sufficiently complete that our examples in 5 work well, and did not involve any further changes to the HOL Light kernel. We do expect that adding further examples, which is ongoing, will require additional machinery but no changes to the kernel.

The HOL Light QE system was developed by the third author under the supervision of the first two authors on an undergraduate NSERC USRA research project at McMaster University and is available at

`https://github.com/JacquesCarette/hol-light`.

It should be further remarked that our fork, from late April 2017, is not fully up-to-date with respect to HOL Light. In particular, this means that it is best to compile it with OCaml 4.03.0 and camlp5 6.16, both available from opam.

To run HOL Light QE, execute the following commands in HOL Light QE top-level directory named `hol_light`:

```
1) install opam
2) opam init --comp 4.03.0
3) opam install "camlp5=6.16"
5) opam ‘eval config env‘
5) cd hol_light
6) make
7) run ocaml via
     ocaml -I ‘camlp5 -where‘ camlp5o.cma
8) #use "hol.ml";;
   #use "Constructions/epsilon.ml";;
   #use "Constructions/pseudoquotation.ml";;
   #use "Constructions/QuotationTactics.ml";;
```

Each test can be run by an appropriate further `#use` statement.

4.2 Mapping of $\mathbf{CTT_{qe}}$ Expressions to HOL Terms

Tables 1 and 2 illustrate how the $\textsc{ctt}_{\text{qe}}$ types and expressions are mapped to the HOL types and terms, respectively. The HOL types and terms are written in the the internal representation employed in HOL Light QE. The type `epsilon` and the term constructors `Quote` and `Eval` are additions to HOL Light explained below. Since $\textsc{ctt}_{\text{qe}}$ does not have type variables, it has a logical constant $=_{\alpha\to\alpha\to o}$ representing equality for each $\alpha \in \mathcal{T}$. The members of this family of constants are all mapped to a single HOL constant with the polymorphic type `a_ty_var->a_ty_var->bool` where `a_ty_var` is any chosen HOL type variable.

The other logical constants of $\textsc{ctt}_{\text{qe}}$ [25, Table 1] are not mapped to primitive HOL constants. $\mathsf{app}_{\epsilon\to\epsilon\to\epsilon}$, $\mathsf{abs}_{\epsilon\to\epsilon\to\epsilon}$, and $\mathsf{quo}_{\epsilon\to\epsilon}$ are implemented by `App`, `Abs`, and `Quo`, constructors for the inductive type `epsilon` given below. The remaining logical constants are predicates on constructions that are implemented by HOL functions. The $\textsc{ctt}_{\text{qe}}$ type ϵ is the type of constructions, the syntactic values that represent the syntax trees of eval-free expressions. ϵ is formalized as an inductive type `epsilon`. Since types are components of terms in HOL Light, an

$\mathbf{CTT_{qe}}$ **Type** α	**HOL Type** $\mu(\alpha)$	**Abbreviation for** $\mu(\alpha)$
o	`Tyapp("bool",[])`	`bool`
ι	`Tyapp("ind",[])`	`ind`
ϵ	`Tyapp("epsilon",[])`	`epsilon`
$\beta \to \gamma$	`Tyapp("fun",[`$\mu(\beta), \mu(\gamma)$`])`	$\mu(\beta)$`->`$\mu(\gamma)$

Table 1. Mapping of $\textsc{ctt}_{\text{qe}}$ Types to HOL Types

$\mathbf{CTT_{qe}}$ **Expression** e	**HOL Term** $\nu(e)$
$\mathbf{x}_\alpha$	`Var("x",`$\mu(\alpha)$`)`
$\mathbf{c}_\alpha$	`Const("c",`$\mu(\alpha)$`)`
$=_{\alpha\rightarrow\alpha\rightarrow o}$	`Const("=",a_ty_var->a_ty_var->bool)`
$(\mathbf{F}_{\alpha\rightarrow\beta}\,\mathbf{A}_\alpha)$	`Comb(`$\nu(\mathbf{F}_{\alpha\rightarrow\beta}), \nu(\mathbf{A}_\alpha)$`)`
$(\lambda\,\mathbf{x}_\alpha\,.\,\mathbf{B}_\beta)$	`Abs(Var("x",`$\mu(\alpha)$`),`$\nu(\mathbf{B}_\beta)$`)`
$\ulcorner\mathbf{A}_\alpha\urcorner$	`Quote(`$\nu(\mathbf{A}_\alpha), \mu(\alpha)$`)`
$[\![\mathbf{A}_\epsilon]\!]_{\mathbf{B}_\beta}$	`Eval(`$\nu(\mathbf{A}_\epsilon), \mu(\beta)$`)`

Table 2. Mapping of $\mathrm{CTT_{qe}}$ Expressions to HOL Terms

inductive type `type` of syntax values for HOL Light QE types (which are the same as HOL types) is also needed. Specifically:

```
define_type "type = TyVar string
                  | TyBase string
                  | TyMonoCons string type
                  | TyBiCons string type type"

define_type "epsilon = QuoVar string type
                     | QuoConst string type
                     | App epsilon epsilon
                     | Abs epsilon epsilon
                     | Quo epsilon"
```

Terms of type `type` denote the syntax trees of HOL Light QE types, while the terms of type `epsilon` denote the syntax trees of HOL Light QE terms that are eval-free.

The OCaml type of HOL types in HOL Light QE

```
type hol_type = Tyvar of string
              | Tyapp of string * hol_type list
```

is the same as in HOL Light, but the OCaml type of HOL terms in HOL Light QE

```
type term = Var of string * hol_type
          | Const of string * hol_type
          | Comb of term * term
          | Abs of term * term
          | Quote of term * hol_type
          | Hole of term * hol_type
          | Eval of term * hol_type
```

has three new constructors: `Quote`, `Hole`, and `Eval`.

`Quote` constructs a quotation of type `epsilon` with components t and α from a term t of type α that is is eval-free. `Eval` constructs an evaluation of type α with components t and α from a term t of type `epsilon` and a type α. `Hole` is used to construct "holes" of type `epsilon` in a quasiquotation as described in [25]. A HOL Light QE quotation that contains holes is a quasiquotation, while a quotation without any holes is a normal quotation. The construction of terms has been modified to allow a hole (of type `epsilon`) to be used where a term of some other type is expected.

The external representation of a quotation `Quote(t,ty)` is `Q_ t _Q`. Similarly, the external representation of a hole `Hole(t,ty)` is `H_ t _H`. The external representation of an evaluation `Quote(t,ty)` is `eval t to ty`.

4.3 Modification of the HOL Light Proof System

We said in 2.4 that the proof system for CTT_{qe} is obtained by extending the proof system for $\mathcal{Q}_0$ with additional axioms B1–B13. Since $\mathcal{Q}_0$ and HOL Light are both complete (with respect to the semantics of Henkin-style general models), HOL Light includes the reasoning capabilities of the proof system for $\mathcal{Q}_0$ but obviously not the reasoning capabilities embodied in the B1–B13 axioms. These capabilities must be implemented in HOL Light QE.

This is done in several ways. First, the logical constants defined by Axioms B1–B4, B5, and B7 are defined in HOL Light QE as HOL functions. Second, the no junk (B6) and no confusion (B4) requirements for ϵ are automatic consequences of defining `epsilon` as an inductive type. Third, Axiom B9 is implemented directly in the HOL Light code for substitution. Fourth, the remaining axioms, B8 and B10–B13 are implemented by new rules of inference in HOL Light QE as shown in Table 3.

CTT$_{\text{qe}}$ Axioms	**NewRules of Inference**
B8 (Properties of Quotation)	`LAW_OF_QUO`
B10 (Properties of Evaluation)	
B10.1	`VAR_DISQUO`
B10.2	`CONST_DISQUO`
B10.3	`APP_SPLIT`
B10.4	`ABS_SPLIT`
B10.5	`QUOTABLE`
B11 (Beta-Reduction for Evaluations)	
B11.1	`BETA_EVAL`
B11.2	`BETA_REVAL`
B12 ("Not Free In" means "Not Effective In")	`NOT_FREE_OR_EFFECTIVE_IN`
B13 (Beta-Reduction for Function Abstractions)	`NEITHER_EFFECTIVE`

Table 3. New Inference Rules in HOL Light QE

The HOL Light `INST` rule of inference is also modified. This rule simultaneously substitutes a list of terms for a list of variables in a sequent. The substitution function `vsubst` defined in the HOL Light kernel is modified so that it works like substitution (via beta-reduction rules) does in CTT_{qe}. These are the main changes to the function:

1. A substitution of a term `t` for a variable `x` in a function abstraction `Abs(y,s)` is performed as usual if (1) `t` is eval-free and `x` is not free in `t`, (2) there is a theorem that says `x` is not effective in `t`, (3) `s` is eval-free and `x` is not free

in `s`, or (4) there is a theorem that says `x` is not effective in `s`. Otherwise, if `s` or `t` is not eval-free, the substitution fails and if `s` and `t` are eval-free, the variable `x` is renamed and the substitution is continued.

2. A substitution of a term `t` for a variable `x` in a quotation `Quote(e,ty)` where `e` does not contain any holes (i.e., terms of the form `Hole(e',ty')`) returns `Quote(e,ty)` unchanged (as stated in Axiom B9). If `e` does contain holes, then `t` is substituted for the variable `x` in the holes in `Quote(e,ty)`.
3. A substitution of a term `t` for a variable `x` in an evaluation `Eval(e,ty)` returns (1) `Eval(e,ty)` when `t` is `x` or `x` does not occur `e` and (2) the function abstraction application `Comb(Abs(x,Eval(e,ty)),t)` otherwise. When (2) happens, this part of the substitution is finished and the user can possibly continue it by applying `BETA_EVAL` or `BETA_REVAL`, the rules of inference corresponding to Axioms B11.1 and B11.2 for beta-reduction of evaluations.

4.4 Creation of Support Machinery

The HOL Light QE contains a number of HOL functions, tactics, and theorems that are useful for reasoning about constructions, quotations, and evaluations. An important example is the HOL function `isExprType` that implements the $\mathrm{CTT}_{\mathrm{qe}}$ family of logical constants $\textsf{is-expr}^{\alpha}_{\epsilon\rightarrow o}$ where α ranges over members of $\mathcal{T}$. This function takes terms s_1 and s_1 of type `epsilon` and `type`, respectively, and returns true iff s_1 represents the syntax tree of a term t, s_2 represents the syntax tree of a type α, and t is of type α.

5 Examples

We present two examples that test the implementation of HOL Light QE and illustrate its capabilities by expressing, instantiating, and proving formula schemas.

5.1 Law of Excluded Middle

The *law of excluded middle (LEM)* is expressed as the formula schema $A \vee \neg A$ where A is a syntactic variable ranging over all formulas. Each instance of LEM is a theorem of HOL, but LEM cannot be expressed in HOL as a single formula. However, LEM can be formalized in $\mathrm{CTT}_{\mathrm{qe}}$ as the universal statement

$$\forall\, x_\epsilon \,.\, \textsf{is-expr}^{o}_{\epsilon\rightarrow o}\, x_\epsilon \supset [\![x_\epsilon]\!]_o \vee \neg[\![x_\epsilon]\!]_o.$$

An instance of this statement of LEM is any instance of the universal formula. It can be directly written in HOL Light QE as the formula

```
`!x:epsilon. isExprType (x:epsilon) (TyBase "bool")
   ==> ((eval x to bool) \/ ~(eval x to bool))`
```

that is readily proved. An instance of this theorem is obtained by applying the rule of inference `INST` followed by `BETA_EVAL` or `BETA_REVAL`, the beta-reduction rules for evaluations.

5.2 Induction Schema

The (first-order) *induction schema for Peano arithmetic* is usually expressed as the formula schema

$$(P(0) \wedge \forall\, x \,.\, (P(x) \supset P(S(x)))) \supset \forall\, x \,.\, P(x)$$

where $P(x)$ is a parameterized syntactic variable that ranges over all formulas of first-order Peano arithmetic. If we assume that the domain of the type ι is the natural numbers and $\mathcal{C}$ includes the usual constants of natural number arithmetic (including a constant $S_{\iota\to\iota}$ representing the successor function), then this schema can be formalized in $\textsc{ctt}_{\rm qe}$ as

$$\begin{array}{l} \forall\, f_\epsilon \,.\, ((\textsf{is-expr}_{\epsilon\to o}^{\iota\to o}\, f_\epsilon \wedge \textsf{is-peano}_{\epsilon\to o}\, f_\epsilon) \supset \\ \quad ((\llbracket f_\epsilon \rrbracket_{\iota\to o}\, 0 \wedge (\forall\, x_\iota \,.\, \llbracket f_\epsilon \rrbracket_{\iota\to o}\, x_\iota \supset \llbracket f_\epsilon \rrbracket_{\iota\to o}\, (S_{\iota\to\iota}\, x_\iota))) \supset \forall\, x_\iota \,.\, \llbracket f_\epsilon \rrbracket_{\iota\to o}\, x_\iota)) \end{array}$$

where $\textsf{is-peano}_{\epsilon\to o}\, f_\epsilon$ holds iff f_ϵ represents the syntax tree of a predicate of first-order Peano arithmetic. The *induction schema for Presburger arithmetic* is exactly the same as the induction schema for Peano arithmetic except that the predicate $\textsf{is-peano}_{\epsilon\to o}$ is replaced by an appropriate predicate $\textsf{is-presburger}_{\epsilon\to o}$.

It should be noted that the induction schemas for Peano and Presburger arithmetic are weaker that the full induction principle for the natural numbers:

$$\forall\, p_{\iota\to o} \,.\, ((p_{\iota\to o}\, 0 \wedge (\forall\, x_\iota \,.\, p_{\iota\to o}\, x_\iota \supset p_{\iota\to o}\, (S_{\iota\to\iota}\, x_\iota))) \supset \forall\, x_\iota \,.\, p_{\iota\to o}\, x_\iota)$$

The full induction principle states that induction holds for all properties of the natural numbers (which is an uncountable set), while the induction schemas for Peano and Presburger arithmetic hold only for properties that are definable in Peano and Presburger arithmetic (which are countable sets).

The full induction principle is expressed in HOL Light as the theorem

```
‘!P. P(_0) / (!n. P(n) ==> P(SUC n)) ==> !n. P n‘
```

named `num_INDUCTION`. However, it is not possible to directly express the Peano and Presburger induction schemas in HOL Light without adding new rules of inference to its kernel.

The induction schema for Peano arithmetic can be written in HOL Light QE just as easily as in $\textsc{ctt}_{\rm qe}$:

```
‘!f:epsilon.
   (isExprType (f:epsilon)
      (TyBiCons "fun" (TyVar "num") (TyBase "bool")))
   /\ (isPeano f)
   ==>
   (eval (f:epsilon) to (num->bool)) 0
    /\
   (!n:num. (eval (f:epsilon) to (num->bool)) n
      ==>
      (eval (f:epsilon) to (num->bool)) (SUC n))
   ==> (!n:num. (eval (f:epsilon) to (num->bool)) n)‘
```

This statement named `peanoInduction` is proved from `num_INDUCTION` in HOL Light QE by the following steps:

1. Instantiate `num_INDUCTION` with the predicate `‘P:num->bool‘` to obtain the formula `indinst`.
2. Prove and install the theorem `nei_peano` that says the variable `(n:num)` is not effective in `(eval (f:epsilon) to (num->bool))`.
3. Logically reduce `peanoInduction` and then prove the result by instantiating `‘P:num->bool‘` in `indinst` with `‘eval (f:epsilon) to (num->bool)‘` using the `INST` rule of inference. This instantiation requires the previously proved theorem `nei_peano`.

The induction schema for Presburger arithmetic is stated and proved in the very same way.

By virtue of being able to express the Peano and Presburger induction schemas, we can properly define the first-order theories of Peano arithmetic and Presburger arithmetic in HOL Light QE.

6 Related Work

Quotation, evaluation, reflection, reification, issues of intensionality versus extensionality, metaprogramming and metareasoning each have extensive literature — sometimes in more than one field. For example, one can find a vast literature on reflection in logic, programming languages, and theorem proving. Due to space restrictions, we cannot do justice to the full breadth of issues. For a full discussion, please see the related work section in [25]. The surveys of Costantini [21], Harrison [34] are excellent. From a programming perspective, the discussion and extensive bibliography of Kavvos' D.Phil. thesis [42] are well worth reading.

Focusing just on interactive proof assistants, we find that Robert Boyer and J Moore developed a global infrastructure [6] for incorporating symbolic algorithms into the Nqthm [7] theorem prover. This approach is also used in ACL2 [41], the successor to Nqthm; see [40]. Over the last 30 years, the Nuprl group lead by Robert Constable has produced a large body of work on metareasoning and reflection for theorem proving [1,3,18,38,43,44,54] that has been implemented in the Nuprl [19] and MetaPRL [37] systems. Proof by reflection has become a mainstream technique in the Coq [20] proof assistant with the development of tactics based on symbolic computations like the Coq ring tactic [5,33] and the formalizations of the *four color theorem* [29] and the *Feit-Thompson odd-order theorem* [30] led by Georges Gonthier. See [5,8,12,31,33,39,46] for a selection of the work done on using reflection in Coq. Many other systems also support metareasoning and reflection: Agda [45,51,52], Idris [14,13,15] Isabelle/HOL [11], Lean [22], Maude [17], PVS [53], and Theorema [28,9].

The semantics of the quotation operator $\ulcorner\cdot\urcorner$ is based on the *disquotational theory of quotation* [10]. According to this theory, a quotation of an expression e is an expression that denotes e itself. In $\textsc{ctt}_{\rm qe}$, $\ulcorner\mathbf{A}_\alpha\urcorner$ denotes a value that represents the syntactic structure of $\mathbf{A}_\alpha$. Polonsky [47] presents a set of axioms for quotation operators of this kind. Other theories of quotation have been proposed — see [10] for an overview. For instance, quotation can be viewed as an operation that constructs literals for syntactic values [49].

It is worth quoting Boyer and Moore [6] here:

> The basic premise of all work on extensible theorem-provers is that it should be possible to add new proof techniques to a system without endangering the soundness of the system. It seems possible to divide current work into two broad camps. In the first camp are those systems that allow the introduction of arbitrary new procedures, coded in the implementation language, but require that each application of such a procedure produce a formal proof of the correctness of the transformation performed. In the second camp are those systems that contain a formal notion of what it means for a proof technique to be sound and require a machine-checked proof of the soundness of each new proof technique. Once proved, the new proof technique can be used without further justification.

This remains true to this day. The systems in the LCF tradition (Isabelle/HOL, Coq, HOL Light) are in the "first camp", while Nqthm, ACL2, Nuprl, MetaPRL, Agda, Idris, Lean, Maude and Theorema, as well as our approach broadly fall in the "second camp". However, all systems in the first camp have started to offer some reflection capabilities on top of their tactic facilities. Below we give some additional details for each system, leveraging information from the papers already cited above as well as the documentation of each system[2].

In more detail, the Coq extension for reflection, SSReflect [31], which stands for *small scale reflection*, works by locally reflecting the syntax of particular kinds of objects — such as decidable predicates and finite structures. It is the pervasive use of decidability and computability which gives small scale reflection its power, and at the same time, its limitation. An extension to PVS allows reasoning much in the style of SSReflect. Isabelle/HOL offers a nonlogical `reify` function (aka quotation), while its `interpret` function is in the logic; it uses global datatypes to represent HOL Light terms.

The approach for the second list of systems also varies quite a bit. Nqthm, ACL2, Theorema (as well as now HOL Light QE) have global quotation and evaluation operators in the logic, as well as careful restrictions on their use to avoid paradoxes. Idris also has global quotation and evaluation, and the *totality checker* is used to avoid paradoxes. MetaPRL have evaluation but no global quotation. Agda has global quotation and evaluation, but their use are mediated by a built-in `TC` (TypeChecking) monad which ensures soundness. Lean works similarly: all reflection must happen in the `tactic` monad, from which one cannot escape. Maude appears to offer a global quotation operator, but it is unclear if there is a global evaluation operator; quotations are offered by a built-in module, and those are extra-logical.

[2] And some personal communication with some of system authors.

7 Conclusion

CTT_{qe} [25,26] is a version of Church's type theory with global quotation and evaluation operators that is intended for defining, applying, proving properties about syntax-based mathematical algorithms (SBMAs), algorithms that manipulate expressions in a mathematically meaningful ways. HOL Light QE is an implementation of CTT_{qe} obtained by modifying HOL Light [35], a compact implementation of the HOL proof assistant [32]. In this paper, we have presented the design and implementation of HOL Light QE. We have discussed the challenges that needed to be overcome. And we have given some examples that test the implementation and show the benefits of having quotation and evaluation in higher-order logic.

The implementation of HOL Light QE was very straightforward since the logical issues were worked out in CTT_{qe} and HOL Light provides good support for inductive types. Pleasingly, and surprisingly, no new issues arose during the implementation. HOL Light QE works in exactly the same way as HOL Light except that, in the presence of evaluations, the instantiation of free variables may require proving side conditions that say (1) a variable is not effective in a term or (2) that a variable represented by a construction is not free in a term represented by a construction (see subsections 2.4 and 4.3). This is the only significant cost we see for using HOL Light QE in place of HOL Light.

HOL Light QE provides a built-in global reflection infrastructure [25]. This infrastructure can be used to reason about the syntactic structure of terms and, as we have shown, to express formula schemas as single formulas. More importantly, the infrastructure provides the means to define, apply, and prove properties about SBMAs. An SBMA can be defined as a function that manipulates constructions. The *meaning formula* that specifies its mathematical meaning can be stated using the evaluation of constructions. And the SBMA's meaning formula can be proved from the SBMA's definition. In other words, the infrastructure provides a unified framework for formalizing SBMAs in a proof assistant.

We plan to continue the development of HOL Light QE and to show that it can be effectively used to develop SBMAs as we have just described. In particular, we intend to formalize in HOL Light QE the example on the symbolic differentiation in CTT_{qe} that is presented in [25]. This will require defining the algorithm for symbolic differentiation, writing its meaning formula, and finally proving the meaning formula from the algorithm's definition and standard facts about derivatives.

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, pages 95–105. IEEE Computer Society, 1990.
2. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof, Second Edition.* Kluwer, 2002.

3. E. Barzilay. *Implementing Reflection in Nuprl*. PhD thesis, Cornell University, 2005.
4. A. Bawden. Quasiquotation in Lisp. In O. Danvy, editor, *Proceedings of the 1999 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12, 1999. Technical report BRICS-NS-99-1, University of Aarhus, 1999.
5. S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997.
6. R. Boyer and J Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 103–185. Academic Press, 1981.
7. R. Boyer and J Moore. *A Computational Logic Handbook*. Academic Press, 1988.
8. T. Braibant and D. Pous. Tactics for reasoning modulo AC in Coq. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs (CPP 2011)*, volume 7086 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2011.
9. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4:470–504, 2006.
10. H. Cappelen and E. LePore. Quotation. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2012 edition, 2012.
11. A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41:33–59, 2008.
12. A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.
13. David Christiansen and Edwin Brady. Elaborator reflection: Extending idris in idris. *SIGPLAN Not.*, 51(9):284–297, September 2016.
14. David Raymond Christiansen. Type-directed elaboration of quasiquotations: A high-level syntax for low-level reflection. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages*, IFL '14, pages 1:1–1:9, New York, NY, USA, 2014. ACM.
15. David Raymond Christiansen. *Practical Reflection and Metaprogramming for Dependent Types*. PhD thesis, IT University of Copenhagen, 2016.
16. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
17. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285:245–288, 2002.
18. R. L. Constable. Using reflection to explain and enhance type theory. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO ASI Series*, pages 109–144. Springer, 1995.
19. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
20. Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.5*, 2016. Available at `https://coq.inria.fr/distrib/current/refman/`.
21. S. Costantini. Meta-reasoning: A survey. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408 of *Lecture Notes in Computer Science*, pages 253–288, 2002.

22. Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):34, 2017.
23. T. Hales et al. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017.
24. W. M. Farmer. The formalization of syntax-based mathematical algorithms using quotation and evaluation. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, volume 7961 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2013.
25. W. M. Farmer. Incorporating quotation and evaluation into Church's type theory. *Computing Research Repository (CoRR)*, abs/1612.02785 (72 pp.), 2016.
26. W. M. Farmer. Incorporating quotation and evaluation into Church's type theory: Syntax and semantics. In M. Kohlhase, M. Johansson, B. Miller, L. de Moura, and F. Tompa, editors, *Intelligent Computer Mathematics*, volume 9791 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2016.
27. W. M. Farmer. Theory morphisms in Church's Type theory with quotation and evaluation. *Computing Research Repository (CoRR)*, abs/1703.02117 (15 pp.), 2017.
28. M. Giese and B. Buchberger. Towards practical reflection for formal mathematics. RISC Report Series 07-05, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, 2007.
29. G. Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *Computer Mathematics (ASCM 2007)*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2008.
30. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2013.
31. Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
32. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
33. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005.
34. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, 1995. Available at `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.ps.gz`.
35. J. Harrison. HOL Light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009.
36. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.
37. J. Hickey, A. Nogin, R. L. Constable, B. E. Aydemir, E. Barzilay, Y. Bryukhov, R. Eaton, A. Granicz, A. Kopylov, C. Kreitz, V. Krupski, L. Lorigo, S. Schmitt, C. Witty, and X. Yu. MetaPRL — A modular logical environment. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303, 2003.

38. D. Howe. Reflecting the semantics of reflected proof. In P. Aczel, H. Simmons, and S. Wainer, editors, *Proof Theory*, pages 229–250. Cambridge University Press, 1992.
39. D. W. H. James and R. Hinze. A reflection-based proof tactic for lattices in Coq. In Horváth Z, V. Zsók, P. Achten, and P. W. M. Koopman, editors, *Proceedings of the Tenth Symposium on Trends in Functional Programming (TFP 2009)*, volume 10 of *Trends in Functional Programming*, pages 97–112. Intellect, 2009.
40. W. A. Hunt Jr., M. Kaufmann, R. B. Krug, J S. Moore, and E. W. Smith. Meta reasoning in ACL2. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2005.
41. M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23:203–213, 1997.
42. G. A. Kavvos. On the Semantics of Intensionality and Intensional Recursion. Available from `http://arxiv.org/abs/1712.09302`, December 2017.
43. T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86)*, pages 237–248. IEEE Computer Society, 1986.
44. A. Nogin, A. Kopylov, X. Yu, and J. Hickey. A computational approach to reflective meta-reasoning about languages with bindings. In R. Pollack, editor, *ACM SIGPLAN International Conference on Functional Programming, Workshop on Mechanized Reasoning about Languages with Variable Binding, (MERLIN 2005)*, pages 2–12. ACM, 2005.
45. U. Norell. Dependently typed programming in Agda. In A. Kennedy and A. Ahmed, editors, *Proceedings of TLDI'09*, pages 1–2. ACM, 2009.
46. M. Oostdijk and H. Geuvers. Proof by computation in the Coq system. *Theoretical Computer Science*, 272, 2002.
47. A. Polonsky. Axiomatizing the Quote. In M. Bezem, editor, *Computer Science Logic (CSL'11) — 25th International Workshop/20th Annual Conference of the EACSL*, volume 12 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 458–469. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2011.
48. W. V. O. Quine. *Mathematical Logic: Revised Edition.* Harvard University Press, 2003.
49. F. Rabe. Generic literals. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics*, volume 9150 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2015.
50. A. Tarski. The concept of truth in formalized languages. In J. Corcoran, editor, *Logic, Semantics, Meta-Mathematics*, pages 152–278. Hackett, second edition, 1983.
51. P. van der Walt. Reflection in Agda. Master's thesis, Universiteit Utrecht, 2012.
52. P. van der Walt and W. Swierstra. Engineering proof by reflection in Agda. In R. Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2012.
53. F. W. von Henke, S. Pfab, H. Pfeifer, and H. Rueß. Case studies in meta-level theorem proving. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs'98*, volume 1479, pages 461–478. Springer, 1998.
54. X. Yu. *Reflection and Its Application to Mechanized Metareasoning about Programming Languages.* PhD thesis, California Institute of Technology, 2007.