HOL Light QE Overview

Patrick Laskowski

August 28, 2018

Contents

1 Introduction

This document aims to give a detailed overview of the changes made to John Harrison's HOL Light QE proof system. The location of each major change to the system in the code base will be provided, along with an explanation of what the change accomplishes and, where applicable, how it works.

2 type

The `type` type is an addition made in `define.ml`.
The `type` type is an inductive type defined in HOL Light's logic system that is responsible for storing type information. All members are either a `TyVar`, `TyBase`, `TyMonoCons`, or `TyBiCons`. The `type` type is only compatible with HOL types that contain up to and including two type parameters, as no default HOL type exceeds two type parameters. If compatability with more type parameters is desired, the `type` type may be modified accordingly. More information about each of the individual members is located below.

2.1 TyVar

`TyVar` takes a HOL string and is used to represent paramaeterized types (not to be confused with type paramaters in HOL types) in a term. For example, the type of equality in HOL is defined as `A->A->bool`. A `TyVar "A"` would be used to represent `A` in this type definition.

2.2 TyBase

`TyBase` takes a HOL string and is used to represent a HOL type that takes no type parameters. For example, if one wished to represent the HOL type `num`, they would use `TyBase "num"`.

2.3 TyMonoCons

`TyMonoCons` takes a HOL string and an instance of the `type` type, and is used to represent a HOL type that takes one type paramater. For example, expressing the type of a list of numbers, which would appear as `(num)list` inside HOL, can be accomplished with `TyMonoCons "list" (TyBase "num")`.

2.4 TyBiCons

`TyBiCons` takes a HOL string, an instance of the `type` type, and a second instance of the `type` type, and is used to represent a HOL type that takes two type parameters. For example, expressing the type of a function from the numbers to the booleans, which appears as `num->bool` inside HOL, can be accomplished with `TyBiCons "fun" (TyBase "num") (TyBase "bool")`.

3 epsilon

The `epsilon` type is an addition made in `define.ml`.
The `epsilon` type is an inductive type defined in HOL Light's logic system that is responsible for the representation of a term's syntax. All members are either a `QuoVar`, `QuoConst`, `App`, `Abs`, or `Quo`. Every quoted term should be eval-free i.e. does not contain an evaluation, therefore, there is no support for the representation of `Eval` in epsilon.

3.1 QuoVar

`QuoVar` takes a HOL string and an instance of the `type` type, and is used to represent a HOL variable. The HOL term `x:num` would be expressed in epsilon as `QuoVar "x" (TyBase "num")`.

3.2 QuoConst

`QuoConst` takes a HOL string and an instance of the `type` type, and is used to represent a HOL constant. The HOL term `F` would be expressed in epsilon as `QuoConst "F" (TyBase "bool")`.

3.3 App

`App` takes a member of type `epsilon` and another memeber of type `epsilon`, and is used to represent the application of a function (first argument) to a term (second argument). Although any term of type `epsilon` is accepted as the first argument, the term will be ill-formed if it does not represent the syntax of a function. The HOL term `T \/ F` would be expressed in epsilon as `App (App (QuoConst "\\/" (TyBiCons "fun" (TyBase "bool") (TyBiCons "fun" (TyBase "bool") (TyBase "bool")))) (QuoConst "T" (TyBase "bool"))) (QuoConst "F" (TyBase "bool"))`.

3.4 Abs

`Abs` takes a member of type `epsilon` and another member of type `epsilon`, and is used to represent a lambda expression. The first argument should be a `QuoVar` that represents the variable to be bound in the second term. Again, this restriction is not enforced, but the term will not be well formed unless the restriction is satisfied. The HOL term `\x.x:bool` would be expressed in epsilon as `Abs (QuoVar "x" (TyBase "bool")) (QuoVar "x" (TyBase "bool"))`.

3.5 Quo

`Quo` takes a member of type `epsilon`, and is used to represent a quotation. The HOL term `Q_x:bool _Q` would be expressed in epsilon as `Quo (QuoVar "x" (TyBase "bool"))`.

4 Kernel Modifications

This section covers additions and changes made to `fusion.ml`.

4.1 term

Three new members have been added to the definition of the `term` type.

4.1.1 Quote

This member takes a term and a HOL type that should be equivelant to the type of the term, and is used to signify a quoted term.

4.1.2 Hole

This member is used inside of `Quote` terms to indicate a usage of quasiquotation. It takes a single term and a HOL type that should be equivelant to the type of the term.

4.1.3 Eval

This member takes a term and an HOL type and expresses an evaluation of the syntax in the term to a value of the given type.

4.2 Term constructors

Term constructors have been added for the three additions to the `term` type.

4.2.1 mk_quote

`mk_quote` takes a term and returns a new term structurally equivelant to `Quote(term,type_of(term))`.

4.2.2 mk_hole

`mk_hole` takes a term and returns a new term structurally equivelant to `Hole(term,type_of(term))`.

4.2.3 mk_eval

`mk_eval` takes a pair `(term,HOL_type)` and returns a new term structurally equivelant to `Eval(term,HOL_type)`.

4.3 Term destructors

Term destructors have been added for the three additions to the `term` type.

4.3.1 dest_quote

`dest_quote` takes a term of the format `Quote(e,t)` and returns the pair `(e,t)`.

4.3.2 dest_hole

`dest_hole` takes a term of the format `Hole(e,t)` and returns the pair `(e,t)`.

4.3.3 dest_eval

`dest_eval` takes a term of the format `Eval(e,t)` and returns the pair `(e,t)`.

4.4 Rules of Inference

The following rules of inference have been added to the HOL kernel.

4.4.1 termToConstruction

`termToConstruction` takes a term of the form `Quote(e,t)` and returns a theorem stating that the term is equal to the representation of `e` in `epsilon`. This can also be done in the logic using the term `TTC`.

4.4.2 constructionToTerm

`constructionToTerm` takes a term of type `epsilon` and returns a theorem stating that the term is equal to the term that would have to be given to `termToConstruction` in order to generate the input term. `constructionToTerm` is the inverse of `termToConstruction`.

4.4.3 LAW_OF_QUO

`LAW_OF_QUO` takes a term of type `Quote(e,t)` and attempts to instantiate this term into the law of quotation.

4.4.4 VAR_DISQUO

`VAR_DISQUO` takes a term with the format `eval quo (QuoVar x ty)` and returns a theorem stating that the term is equal to x.

4.4.5 CONST_DISQUO

`CONST_DISQUO` takes a term with the format `eval quo (QuoConst x ty)` and returns a theorem stating that the term is equal to x.

4.4.6 QUOTABLE

`QUOTABLE` takes a term of type `epsilon` and returns an instantiated theorem for Axiom B10(5).

4.4.7 ABS_SPLIT

`ABS_SPLIT` takes a term of type `epsilon` and a second term of type `epsilon` and attempts to instantiate the first term in for `x` and the second in for `A` inside Axiom B10(4).

4.4.8 APP_SPLIT

`APP_SPLIT` takes a term of type `epsilon` and a second term of type `epsilon` and attempts to instantiate the first term in for `A` and the second in for `B` inside Axiom B10(3).

4.4.9 BETA_EVAL

`BETA_EVAL` takes a term of type `epsilon` and a second term of type `epsilon` and attempts to instantiate the first term in for `x` and the second in for `B` inside Axiom B11(a).

4.4.10 BETA_REVAL

`BETA_REVAL` takes three terms of type `epsilon`, and attempts to instantiate the first term in for `x`, the second term in for `A`, and the third in for `B` inside Axiom B11(b).

4.4.11 NOT_FREE_OR_EFFECTIVE_IN

`NOT_FREE_OR_EFFECTIVE_IN` takes a term of type `epsilon` and a second term of type `epsilon`, and attempts to instantiate the first term in for `x` and the second term in for `B` inside Axiom B12.

4.4.12 NEITHER_EFFECTIVE

`NEITHER_EFFECTIVE` takes four terms of type `epsilon` and attempts to instantiate the first in for `x`, the second in for `y`, the third in for `A`, and the fourth in for `B` inside Axiom B13.

4.4.13 effectiveIn

`effectiveIn` takes a term of type `epsilon` and a second term of type `epsilon` and attempts to instantiate the first in for `x` and the second in for `B` into the definition of `IS-EFFECTIVE-IN` (as defined in "Incorporating Quotation and Evaluation Into Churchs Type Theory").

4.4.14 EVAL_QUOTE

`EVAL_QUOTE` takes a term of the format `Eval(e,t)` and attempts to evaluate the syntax of e to a term of type t. If succesful, it returns a theorem asserting that this evaluated term and the input term are equal.

4.4.15 UNQUOTE

`UNQUOTE` takes a Hole with a constant epsilon term inside it, and returns a theorem that removes the hole to bring the epsilon term into the rest of the quotation. For example, `UNQUOTE ‘Q_2 + H_Q_3 _Q _H _Q‘` results in the theorem `Q_(2 + H_(Q_(3) _Q) _H) _Q = Q_(2 + 3) _Q`.

5 Parser changes

These changes have been made in `parser.ml` and `preterm.ml`. `printer.ml` has also been modified to support printing of these terms.

5.1 Q_ operator

To input a quoted term, one can wrap the term in `Q_` and `_Q`. For example: `Q_x * y _Q`.

5.2 H_ operator

To input a hole, one can wrap the term in `H_` and `_H`. For example: `Q_H_f:epsilon _H _Q`.

5.3 eval operator

To input an evaluation, one must use the format `eval term to type`. For example, `eval Q_3 + 2 _Q to num`.

6 Tactics

These changes have been made in `Constructions/ConstructionTactics.ml` and `Constructions/QuotationTactics.ml`.

6.1 TERM_TO_CONSTRUCTION_TAC

`TERM_TO_CONSTRUCTION_TAC` searches through the currently active goal to find the first instance it can apply `TERM_TO_CONSTRUCTION` to, and performs this rewrite.

6.2 UNQUOTE_TAC

`UNQUOTE_TAC` searches through the currently active goal to find the first instance it can apply `UNQUOTE` to, then performs the appropriate rewrite.

6.3 EVAL_QUOTE_TAC

`EVAL_QUOTE_TAC` searches through the currently active goal to find the first instance it can apply `EVAL_QUOTE` to, then performs the appropraite rewrite.

6.4 INTERNAL_TTC_TAC

`INTERNAL_TTC_TAC` performs the same task as `TERM_TO_CONSTRUCTION_TAC`, however, rather than seek out quoted terms, it attempts to turn any term applied to the `TTC` function into a representation of type `epsilon`.

6.5 ASM Modifier

The above tactics can all be used with the prefix `ASM_`, for example, `ASM_UNQUOTE_TAC`, in order to use terms in the goalstack's assumption list during the term rewrite process. These should be used when the success of a tactic relies on the truth of terms in the assumption list.

6.6 STRING_FETCH_TAC

`STRING_FETCH_TAC` is used to automatically resolve all possible string comparisons inside the goal. This is due to possible large amounts of string comparisons that would be tediuous to compare manually through HOL's native tactics.

7 Epsilon Additions

These are defined in `epsilon.ml`, and are a collection of theorems and HOL functions (not OCaml functions) defined on the `epsilon` and `type` types.

7.1 typeDistinct

`typeDistinct` is a theorem that asserts that different members of `type` cannot be equal to each other.

7.2 epsilonDistinct

`epsilonDistinct` is a theorem that asserts that different members of `epsilon` cannot be equal to each other.

7.3 ep_constructor

`ep_constructor` is a function that takes a member of type `epsilon` and returns which member it is as an HOL string.

7.4 ep_type

ep_type is a function that takes a memeber of type epsilon and returns it's type as a member of type type. ep_type is only defined for QuoVar, QuoConst, and Quo. combinatoryType should be used for more advanced terms.

7.5 stripFunc

stripFunc takes a TyBiCons representing a function type and returns a member of type denoting the type the function returns.

7.6 headFunc

headFunc takes a TyBiCons representing a function type and returns a member of type denoting the type of argument the function accepts.

7.7 combinatoryType

combinatoryType takes a member of epsilon and returns a member of type equivalent to the HOL type that would be obtained from running type_of on the HOL representation of the term.

7.8 isVar

isVar takes a member of epsilon and returns true or false as an HOL constant denoting whether or not the given term is a QuoVar

7.9 isConst

isConst takes a member of epsilon and returns true or false as an HOL constant denoting whether or not the given term is a QuoConst

7.10 isAbs

isAbs takes a member of epsilon and returns true or false as an HOL constant denoting whether or not the given term is a Abs

7.11 isApp

isApp takes a member of epsilon and returns true or false as an HOL constant denoting whether or not the given term is a App

7.12 isFunction

isFunction takes a member of type and returns a boolean expression to determine whether or not the term denotes the type of a function.

7.13 isValidConstName

`isValidConstName` takes an HOL string and determines whether or not the string is a valid constant name according to a list of all valid constant names. This does not automatically update, and new valid constant names can be added here.

7.14 isValidType

`isValidType` takes a member of `type` and determines whether or not the type represents a valid HOL type. Like `isValidConstantName`, the check is done with a static list of valid type names, and new valid type names can be added here if needed.

7.15 typeMismatch

`typeMismatch` takes a member of `epsilon` that must be a `QuoVar` and a second memeber of `epsilon`. `typeMismatch` then returns a HOL boolean constant denoting whether or not a variable of the same name as the first term but with a different type appears in the second term.

7.16 isExpr

`isExpr` takes a member of `epsilon` and returns a boolean constant denoting whether or not the given member of `epsilon` represents a well formed term in HOL.

7.17 isVarType

`isVarType` takes a member of `epsilon` and a member of `type` and returns a boolean constant denoting whether or not the given member of `epsilon` is a `QuoVar` with a type equivelant to the given memeber of `type`.

7.18 isConstType

`isConstType` takes a member of `epsilon` and a member of `type` and returns a boolean constant denoting whether or not the given member of `epsilon` is a `QuoConst` with a type equivelant to the given memeber of `type`.

7.19 isExprType

`isExprType` takes a member of `epsilon` and a memeber of `type` and returns a boolean constant denoting whether or not the given member of `epsilon` represents a well formed term in HOL and whether or not the term's type is equivelant to the given member of `type`.

7.20 isProperSubexpressionOf

`isProperSubexpressionOf` takes a member of `epsilon` and a second member of `epsilon` and returns a boolean constant whether or not the first given member of `epsilon` represents a well formed term in HOL, and whether or not the first given member of `epsilon` appears anywhere inside the second given member of `epsilon`.

7.21 e_abs

`e_abs` takes a member of type `epsilon` e and a second member of type `epsilon` f and returns `Abs e f`.

7.22 app

`app` takes a member of type `epsilon` e and a second member of type `epsilon` f and returns `App e f`.

7.23 quo

`quo` takes a member of type `epsilon` and returns `Quo e`.

7.24 eqTypes

`eqTypes` takes a member of `type` and a second member of `type` and returns a boolean constant denoting whether or not the types are equivalent.

7.25 isConstruction

`isConstruction` takes a member of `epsilon` and returns a boolean constant denoting whether or not the given member of `epsilon` is a valid construction i.e. has no holes in it.

7.26 appQuo

`appQuo` is Axiom 8.1 defined on members of type `epsilon`.

7.27 absQuo

`absQuo` is Axiom 8.2 defined on members of type `epsilon`.