Manual for CL Watson

M. Randall Holmes

June 8, 2020

1 Introduction

CL Watson is a version of the Watson theorem prover. Watson is an interactive equational higher-order prover whose higher-order logic is based on the weakly extensional version *NFU* of Quine's set theory "New Foundations", presented as a stratified lambda-calculus.

CL Watson is based on combinatory logic instead of lambda-calculus. The intention is to implement stratified lambda-calculus without actually explicitly implementing either stratification or lambda-calculus. Part of the intention is that all manipulations of terms made by the computer represent logical moves which are directly, locally justifiable without any appeal to such nonlocal conditions as stratification of terms or position in terms defined by cases.

CL Watson uses variable binding notation of the usual kind, unlike Watson. It uses the metaphor of navigation in terms to handle application of theorems (which are all understood as rewrite rules). Navigation is extended to include navigation to formal values as well as to subterms: this is how "lambda-terms" are implemented. A term $(\lambda x.T)$ has no subterm of the form T: it is a combinator which evaluates to T when applied to the variable x. The display functions of the prover recognize the term as an "abstraction", and so display it as a λ-term, and the navigation functions allow one to navigate to the "subterm" T, but what actually happens when T is rewritten involves abstraction from the rewritten form.

2 The Logic

The basic logical rules are those of equational logic which are assumed to be familiar. All theorems are equations, supposed universally quantified over any variables present.

Primitives are constants `Abst`, `Id`, and `Eq`. If T and U are terms, then $T(U)$ is a term (the application of T to U) and $|T|$ is a term (the constant function with value T).

The combinatory logic axioms proper are

Identity: $\texttt{Id}(x) = x$

Constant: $|T|(U) = T$

Distribution: $\texttt{Abst}(T)(U)(V) = T(|V|)(U(V))$

Constant Expansion: $|T(U)| = \texttt{Abst}(|T|)(|U|)$

These axioms support a meta-theorem.

Definition: If A is a set of integers, define $A+1$ as the set of all successors of elements of A and $A-1$ as the set of all predecessors of elements of A. Where U is a term and T is a term, define $\texttt{type}(U,T)$, a set of integers, as follows. $\texttt{type}(T,T) = \{0\}$. $\texttt{type}(U,T) = \mathcal{Z}$ (the set of all integers) if T is atomic and U is distinct from T. $\texttt{type}(U,|T|) = \texttt{type}(U,T) - 1$. $\texttt{type}(V,T(U)) = (\texttt{type}(V,T)+1) \cap \texttt{type}(V,U)$. It should be clear that $\texttt{type}(,T)$ is always either $\mathcal{Z}$ (if U does not occur in T), a singleton set $\{n\}$ (in which case we say that U has type n in T), or $\emptyset$ (in which case we say that U is ill-typed in T).

Abstraction Theorem: If $0 \in \texttt{type}(x,T)$, then there is a term $(\lambda x.T)$, not containing the variable x, such that $(\lambda x.T)(x) = T$ is a theorem.

Proof: For any term T, we can construct a term T' which contains no subterm of the form $|T(U)|$ by repeated rewriting using the Constant Expansion axiom.

For any terms T and U, we define a term $(\lambda U.T)$ as follows: if $T = U$, define $(\lambda U.T)$ as `Id`. If T is an atomic term or constant function term distinct from U define $(\lambda U.T)$ as $|T|$. If T is a composite term $V(W)$ define $(\lambda U.T)$ as $\texttt{Abst}(\lambda|U|.V)(\lambda U.W)$.

Induction on the structure of terms, along with the defining axioms for the combinators, establishes that $(\lambda U.T)(U) = T$ is a theorem for any T and U. Our further claim is that if T' is chosen (as it always can be) such that $T = T'$ is a theorem and T' contains no subterm of the form $|V(W)|$ and if $\mathtt{type}(x.T)$ contains 0, and if U is either atomic or an iterated constant function of an a tomic term, then $(\lambda U.T')$ (which is already seen to satisfy $(\lambda U.T')(x) = T' = T$) will further not have U as a subterm.

We run through the induction again. If $T' = U$, then $(\lambda U.T') = \mathtt{Id}$ contains no occurrence of U. If T' is an atomic term distinct from U, then $(\lambda U.T') = |T'|$ contains no occurrence of U. If T' is a constant function, it must be an iterated constant function of an atomic term. If this term is U, then $\mathtt{type}(U, T)$ is the singleton set of a negative integer, and so does not contain 0. If this term is not U, then $(\lambda U.T') = |T'|$ does not contain an occurrence of U. If T' is of the form $V(W)$ then $(\lambda U.T') = \mathtt{Abst}(\lambda|U|.V)(\lambda U.W)$ and we need to argue that neither $(\lambda|U|.V)$ nor $(\lambda U.W)$ contain x if $\mathtt{type}(U, T)$ contains 0. We must have 0 in $\mathtt{type}(U, W)$, so induction tells us that $(\lambda U.W)$ contains no occurrence of U. We must have -1 in $\mathtt{type}(U, V)$. Any occurrence of U in V must be in the context $|U|$, and all these occurrences of $|U|$ must have type 0 in V. From this it follows that $(\lambda|U|.V)$ contains no occurrence of $|U|$ by induction, and so can contain no occurrence of U, since any occurrence of U in $(\lambda|U|.V)$ would need to be derived from an occurrence of U in V, which would appear as part of a subterm $|U|$ eliminated by the abstraction algorithm.

We introduce a weak extensionality axiom. The motivation is that we allow rewrites inside abstraction terms $(\lambda x.T)$.

Extension Axiom: If $T = U$ is a theorem, so is $(\lambda x.T) = (\lambda x.U)$. Further, $(\lambda x.T(x)) = T$, where T is of one of the forms $\mathtt{Abst}$, $\mathtt{Abst}(\mathtt{U})$.

Refinement of the Abstraction Algorithm: The abstraction algorithm can be modified as follows: when it happens that $(\lambda|U|.V) = |V|$ and $(\lambda U.W) = |W|$, define $(\lambda U.V(W))$ as $|V(W)|$ (this is equal to the abstraction obtained under the original algorithm by the axiom of Constant Expansion); define $(\lambda x.U(x))$ as U in case U is of one of the forms $\mathtt{Id}$, $|V|$, $\mathtt{Abst}$, $\mathtt{Abst}(V)$ or $\mathtt{Abst}(V)(W)$ (justified by the Extension Axiom), and otherwise define it as usual. Further, replace $(\lambda x.T(x)$ with

T in the contexts U, V in terms $\texttt{Abst}(U)$, $\texttt{Abst}(U)(V)$ (also justified by the Extension Axiom).

[Internally to CL Watson, an "inertia operator" is applied to U when it is chosen as $(\lambda x.U(x))$: this helps make the reduction algorithm inverse to the abstraction algorithm in case U itself is a "lambda-term".]

The CL Watson logic of equality will be discussed as it is implemented. The intention is that $\texttt{Eq}(|||T|||)(||U||)(|V|)(W)$ (which actually can be written `V {T = U} W` in CL Watson infix notation, though it will initially be written `V {||T|| = ||U||} W`) means "if $T = U$ then V else W". The axioms are

True Equation:

$$\texttt{Eq}(|||X|||)(||X||)(|Y|)(Z) = Y$$

False Equation:

$$\texttt{Eq}(|||\texttt{Id}|||)(|||\texttt{Id}|||)(|Y|)(Z) = Z$$

The nesting of constant functions obscures the fact that this axiom asserts that `Id` and `|Id|` are distinct. This is a generic choice for drawing a distinction because these are in effect the projection operators: $\texttt{Id}(|X|)(Y) = X$ and $|\texttt{Id}|(|X|)(Y) = Y$.

Application Distribution:

$$\texttt{Eq}(|||T|||)(||U||)(|A(B)|)(C(D))$$

=

$$\texttt{Eq}(|||T|||)(||U||)(|A|)(C)(\texttt{Eq}(|||T|||)(||U||)(|B|)(D))$$

Constant Function Distribution:

$$\texttt{Eq}(|||T|||)(||U||)(||A||)(|B|)$$

=

$$|\texttt{Eq}(|||T|||)(||U||)(A)(B)|$$

Substitution under Hypotheses:

$$\texttt{Eq}(|||T|||)(||U||)(|T|)(B)$$

=

$$\texttt{Eq}(|||T|||)(||U||)(|U|)(B)$$

Weak Extensionality/Choice:

$$\texttt{Eq}(|||(\lambda x.T(x))||||)(|||(\lambda x.U(x))|||)(|V|)(W)$$

$=$

$$\texttt{Eq}(|||T(\texttt{Diff}(|T|)(U)(X))||||)(|||U(\texttt{Diff}(|T|)(U)(X))|||)(|V|)(W)$$

3 The Program

3.1 The Term Language

Constant atomic terms of the language of CL Watson are either strings of letters with the initial capitalized and any others lower case (`A` is a term) or strings of special characters (precise definition is the function `isspecial` in the source).

Variable atomic terms of the language of CL Watson have the letter `x` followed by a numeral.

If `T` is a term and `U` is a term then `|T|` and `T(U)` are terms. These constructions correspond to the application and constant function constructions of the logic.

CL Watson supports infix notation. `T U V` is syntactic sugar for `U(|T|)(V)` (this resembles the usual "currying" used to implement functions of two variables in combinatory logic, but notice the adjustment of the type of `T` so that it is the same as the type of `U`). If `T` is an infix term, it will be enclosed in parentheses. If `U` is an infix term, it will be enclosed in braces. If `T` is an infix term in the context `T(U)`, it will be enclosed in braces. Redundant braces and parentheses are all right. This all has the effect of enforcing APL operator precedence: we expect to eventually implement some sort of user-defined operator precedence. CL Watson draws an internal distinction between constant function constructors explicitly supplied by the user and those which appear in the internals of infix notation, so as to avoid terms acquiring unexpected infix forms.

CL Watson supports lambda-notation. If `x1` represents x and `T` represents `T`, then `[x1=>T]` represents $(\lambda x.T)$. Note that the internal representation of `[x1=>T]` contains no variable analogous to `x1`, and the bound variable displayed by the system when displaying a term may be different from the bound variable the user enters.

CL Watson displays `F([x1=>T])` as `[F(x1)=>T]` where F is an atomic constant (in order to support standard notation for quantifiers and other binders). This sometimes has odd effects where atoms not normally thought of as binders are applied to abstractions. CL Watson will sometimes also display terms in a way involving infix notation "accidentally".

One must be careful with whitespace: spaces actually have meaning in the syntax of CL Watson (they separate infixes from neighboring terms). In certain cases CL Watson may supply these spaces when they are omitted (next to special characters; never before an opening parenthesis). It is safer to write them and they are always displayed. Redundant spaces in locations not next to infixes should always be avoided.

A very new feature is the presence of "quoted" terms `"T"`, which represent interpretations of terms in a model of the logic. These are intended for use in supporting unstratified quantification.

3.2 The Proof Model

The model of prover use is this: the user enters a term, applies theorems to it as rewrite rules until a new term is obtained, then records the equation of the original term and the final term as a new theorem usable as a rewrite rule.

At any point, one views the right side of the theorem under construction, and a selected subterm where rewrite rules will be applied.

The command

`Start "T"`

sets up a new proof environment with the term `T` as both left and right term in the "theorem under construction". `s` is an abbreviation for `Start`.

The command

`Look()`

allows one to see the right side of the current term. Variants `Look1()` and `Look2()` support different term views. View 1 supports neither abstraction nor infix notation. View 2 supports abstraction but not infix notation. Commands `view1()` and `view2()` will reset the default view to view 1 or 2; they also change the effect of the navigation commands. `view3()` restores default behavior.

The commands

`workback()`

interchanges the left and right sides of the theorem under construction.

The command

`lookback()`

allows one to look at the left side of the theorem under construction and return to the right side.

The command

`startover()`

sets both left and right sides of the theorem to the current left side.

Now we consider navigation commands, which reset the current subterm.

`right()`

`left()`

These commands are multipurpose. They will go to the right or left subterm of a function application term, to the right or left subterm of an infix term (leaving aside the infix). They will go to the body of an abstraction (that is,to a formal value of the abstraction term).

`middle()`

`function()`

These commands are dedicated to infix terms. `middle()` goes to the infix subterm of an infix term. `function()` goes to the left subterm of an infix term when it is considered as a function application term (the subterm `U(|T|)` which disappears from view when `U(|T|)(V)` is presented as `T U V`). The use of the `function()` command can be avoided by using `toggleinfix()` to change the way the term is viewed.

`value()`

This goes to the formal value of any "abstract" term: it works where an "abstract" is not actually presented in abstract form (this is true of terms `Id`, `|T|`, and `Abst`, and may be true of more terms in later forms of CL Watson).

Terms dedicated to infix terms will not work in view 2; terms dedicated to abstraction terms will not work in view 1. In view 1 or 2, `right()` and `left()` will not do the same thing as in view 3. Any movement command at all takes one to the interior of the quotes in a quoted term.

`up()`

`top()`

The `up()` command reverses the effect of the most recent movement command; the `top()` command takes one to the top of the term.

We now consider the application of theorems.

If `T` and `U` are terms, then `T :>: U` and `T :<: U` are terms, with the same reference as `U`. The effect is to signal the intent to apply a rewrite represented by the term `T`; the alternative form applies the theorem in the

opposite direction. The head of the term `T` will be the name of a theorem (or theorem-like internal function of the prover); the rest of the term can supply parameters to the theorem.

The command `ri "T"` introduces an embedded theorem `T` at the current selected subterm. The command `rri "T"` introduces an embedded theorem using the alternative infix `:<:`.

The command `Execute()` carries out all rewrites signalled by embedded theorems in the selected subterm. Rewrites are carried out in a depth-first manner, and new rewrites introduced by executed rewrites are carried out in their turn until all are eliminated. `ex()` abbreviates `Execute()`.

The command `Onestep()` carries out a single step of the reduction process which is carried out aggressively by `Execute()`.

The command `assign1 n "T"` implements a global substitution of `T` for the variable `xn` (n is a numeral) in both the left and right sides of the current theorem under construction. The command `assigninto n "T"` substitutes the left and right sides of the current term for `xn` in `T` to get new left and right sides.

The command `Extract "T"` converts the selected term `U` to `[T=>U](U)`, where possible. The command `Red()` is the reverse operation of beta-reduction.

The command `makeinert()` makes the selected term "inert" (this is a technicality in the reduction algorithm). The command `toggleinfix` will toggle the display of a potential infix term between the forms `T(|U|)(V)` and `U T V` (it resets the hidden bit in the representation of the constant function between user-supplied and implicit).

The command `prove "T"` proves a new theorem, with left side the left side of the current theorem under construction and right side the right side of the current theorem under construction. The head of the term `T` will be the name of the new theorem: it may have arguments which can be used to pass information to the theorem. In examples, we will see that theorems may have quite complex execution behavior, using built-in higher-order matching facilities, information passed to them in parameters, and introducing new embedded theorems which will in their turn be executed.

The command `axiom "name" "T" "U"` introduces an axiom named `name` asserting `T = U`.

The command `reflect "name1" "name2"` converts a theorem called name1 of the form `"T" = "U"` to a theorem called name2 (if this name is free) of the form `T = U`: it allows theorems proved about the object model to be exported to the meta-model. This is logically strong: we'll see if it is useful.

The command `thmdisplay "T"` displays the theorem with name `T`.

Further commands analogous to those in the Watson user manual may be expected to appear.

4 Examples