Completion as a Derived Rule of Inference

Konrad Slind
University of Calgary

Preliminary Draft

October 10, 1990

Introduction.

Gordon's HOL (Higher Order Logic) system [Gordon89] is a descendant of LCF [GMW79] that implements a version of Church's Simple Type Theory [Church40]. One of the notable features of HOL, indeed all descendants of LCF, is that it enforces adherence to the logic by encapsulating the axioms and primitive rules of inference of the logic inside the *:thm* abstract data type. The type system of the ML programming language in which the logic is implemented ensures that the only way to get an object of type *:thm* is by primitive or derived rules of inference. This is in contrast to the majority of proof systems in which a "theorem" is just something that pops out at the end of the run of some monolithic piece of code.

An important procedure for equality reasoning is Knuth-Bendix completion [KB70], so it is interesting and useful to add completion to HOL. There are several issues raised by this. First, what does it mean to do completion in a typed higher order logic, where algorithms for matching, unification, and term ordering no longer have the nice properties of the first order case, if they exist at all? Second, how can one provide completion in the logic — should it be a primitive rule of inference, a derived rule, or an extralogical tool? Third, what are the possible uses of completion in a higher order logic? The work of Hsiang [HsDer83] and Kapur and Narendran [KN84] shows how refutation theorem proving can be accomplished by a completion algorithm, but we are more interested in how completion can be used in a natural deduction style logic. That is, how does completion exist side by side with other derived rules of inference, rather than being the inference engine running underneath?

This paper advances the view that completion should be a derived rule of inference, and demonstrates how this is possible by implementing completion in HOL. An important part of this development is the characterization of the first order terms of a given type. The completion rule presented here is easily adaptable to other natural deduction logics with equality.

1 In and out of logic.

Meta-theorems are not directly usable in the LCF approach to proof. For example, in an LCF-style implementation of natural deduction propositional logic, we would not be able to directly use the truth table method - the type system enforces adherence to natural deduction. We might use the truth table method to give us the knowledge that a formula is provable by natural deduction, but we wouldn't be allowed to assert that knowledge as a theorem. Similarly, an LCF-style system would only accept the results of a completion algorithm as theorems if they had been produced by rules in that system.

To implement a theorem proving procedure in HOL, therefore, requires that one distinguish between what tasks must be done by inference and what may be done by computation. Hence we draw a distinction between being "inside" the logic, i.e., using inference, and being "outside" the logic, i.e., using other computation.

HOL implements rewriting inside the logic, as a derived rule of inference [Paulson83]. This implies that rewrite rules, the end product of a completion implementation, must be theorems of the logic, which in turn implies that every critical pair must be a theorem. Therefore, critical pairs must be obtained by inference. In general, if we replace "rule", "equation", and "critical pair" by "theorem" in an exposition of Knuth-Bendix completion (as we will), we get what we are after. This reasoning gives us the following breakdown:

inside — inference	*outside — computation*
rewriting by a set of rewrite rules	unification
computing critical pairs	term ordering
renaming variables apart	matching
normalizing the variables in a rule	checking if reduced to identity
orienting an equation to a rule	deciding if a term is first order

In the HOL system, we use computation in two ways: to guide inference and to supply inference rules with information they need — the logic ensures consistency while it is up to the computation, in this case, to provide completeness.

2 An implementation of Simple Type Theory.

Most of this section is paraphrased from the development in the HOL manual [Gordon89].

2.1 Types and terms.

The set of types τ is formed from the following disjoint sets of primitive symbols:

- $\mathcal{V} = \{v_1, v_2, \ldots\}$, an infinite set of variables;
- $\mathcal{C} = \{(bool, 0), (ind, 0), (s_1, n_1), \ldots (s_k, n_k)\}$, a set of *type constants.* A type constant is a (name,arity) pair, where the arity must be a natural number. If two type constants have the same name, they must have the same arity, i.e., no two distinct type constants may have the same name. We distinguish $\mathcal{A} = \{(c, a) \in \mathcal{C} : a = 0\} \cup \mathcal{V}$, the *atomic types.*

The set τ is the smallest set such that:

1. All elements of $\mathcal{A}$ are members of τ.
2. $t_1 \rightarrow t_2 \in \tau$ if t_1 and t_2 are members of τ
3. $(f, n)(t_1, \ldots, t_n) \in \tau$ if $t_1, \ldots, t_n$ are members of τ and $(f, n) \in \mathcal{C}$.

The atomic types *bool* and *ind* represent the (built in) type of boolean values and an infinite set of individuals, respectively. In HOL, the set $\mathcal{C}$ is initially restricted to these two, but can be augmented by means of the *new_type* function, which takes a name and an arity and adds a new type constant to $\mathcal{C}$. Type variables provide polymorphism - a theorem with type variables in it represents a family of theorems derivable by instantiating those type variables. An *instance* δ' of a type δ is obtained by replacing all occurrences of a type variable in δ by a type.

A *signature over* $\mathcal{C}$, $\Sigma_{\mathcal{C}}$, is a set of (name,type) pairs, where the type is a member of τ. The set of HOL terms over a signature, $Terms_{\Sigma_{\mathcal{C}}}$, is defined to be the smallest set such that:

- variables – $v : \gamma$ is a term if v is a name and $\gamma \in \tau$
- constants – $c : \gamma'$ is a term if $(c, \gamma) \in \Sigma_{\mathcal{C}}$ and γ' is an instance of γ
- combinations – if $tm_1 : (\gamma_1 \rightarrow \gamma_2)$ is a term and $tm_2 : \gamma_1$ is a term, then $(tm_1\ tm_2)$ is a term of type γ_2

- abstractions – if $v : \gamma_1$ is a term and *body* $: \gamma_2$ is a term, then $\lambda v.body$ is a term of type $\gamma_1 \rightarrow \gamma_2$.

It is possible for constants and variables in a term to have the same name. It is also possible for different variables in a term to have the same name, in which case they have different types.

2.2 The logic.

HOL is a natural deduction style logic. There are three primitive logical constants: $=$ (equality), $\supset$ (implication), and ε (Hilbert's choice operator). Because of the limitations of the ASCII character set, the HOL system uses `\` for λ, `!` and `?` for universal and existential quantification, respectively, `==>` for implication, and `@` for choice. There are eight basic inference rules:

```
ASSUME --------          DISCH          Gamma |- t2
        tm |- tm                 ---------------------------
                                  Gamma - {t1} |- t1 ==> t2

REFL   ----------        MP       Gamma |- t1 ==> t2    Delta |- t1
        |- t = t                 ----------------------------------
                                         Gamma U Delta |- t2

BETA_CONV    ----------------------------
              |- (\x. t1) t2 = t1[t2/x]

ABS   A |- t1 = t2               INST_TYPE A |- tm  type_subst
    ---------------------------            ------------------
      A |- (\x. t1) = (\x. t2)             A |- type_subst(tm)

SUBST   {(A1 |- l1=r1)/v1; ... ; (An |- ln=rn)/vn}
        tm[v1,...,vn]
        B |- tm[l1,...,ln]
      ----------------------------------------------
            A1 U ... U An U B |- tm[r1, ... rn]
```

There are also five axioms:

```
BOOL_CASES_AX  |- !b:bool. b = T \/ b = F
IMP_ANTISYM_AX |- !b1 b2. (b1 ==> b2)==>(b2 ==> b1)==>(b1 = b2)
ETA_AX         |- !f:(* -> **). (\x. f x) = f
SELECT_AX      |- !P:(*->bool). !x:*. (P x) ==> P (@ P)
INFINITY_AX    |- ?f:(ind -> ind). One_One f /\ ~(Onto f)
```

The only rule of interest to us is **SUBST**; none of the axioms concern us directly. Because we restrict our attention to first order terms, we can ignore **INST_TYPE**, which applies a substitution to the type variables of a term. **SUBST** requires more explanation: its first argument $[\vdash l_1 = r_1/v_1, \ldots, \vdash l_n = r_n/v_n]$ is a list of (theorem,variable) pairs; its second argument is assumed to be a template with (some of) its free variables found in $\{v_1, \ldots, v_n\}$; and the third argument $B \vdash tm[l_1, \ldots, l_n]$ is the theorem that is going to be substituted into. Conceptually, **SUBST** traverses the template and tm in parallel, replacing l_i by r_i in tm when v_i is encountered in the template and l_i is (simultaneously) encountered (free) in tm (automatic renaming takes care of the variable capture problem).

2.3 Formulae as terms.

Terms of type *bool* are called formulas. The standard logical connectives can be defined with the three primitive constants; however, for completion, we are only interested in (possibly universally quantified) equations, so we describe only $\forall$, which is an abbreviation for $\lambda\ (P : * \rightarrow bool).\ P = (\lambda\ x.\ T)$. ($T$ itself abbreviates $(\lambda\ (x : bool).\ x) = (\lambda\ (x : bool).\ x)$.) Although this looks odd, the definition allows the derivation of all the usual rules for universal quantification, e.g., generalization and specialization, so it has no impact on further developments.

3 The first order restriction on terms.

Definition. A *curried type of sort* γ is a function type $\gamma \rightarrow \gamma'$ where γ' is either γ or a curried type of sort γ. The *width* of $\gamma_1 \rightarrow \gamma_2 \rightarrow \ldots \rightarrow \gamma_n$, a curried type of sort γ, is n.

Definition. The set of first order terms of type $\gamma, \mathcal{F}_\gamma$, is the subset of $Terms_{\Sigma_C}$ defined by the following rules:

- variables – if $v : \gamma \in Terms_{\Sigma_C}$, then $v : \gamma$ is in $\mathcal{F}_\gamma$
- constants – if $c : \gamma \in Terms_{\Sigma_C}$, then $c : \gamma \in \mathcal{F}_\gamma$
- combinations – if combination $t : \gamma \in Terms_{\Sigma_C}$, then, since we wish to exclude partial applications, we strip $t = (\ldots(f : \delta\ t_1)\ \ldots\ t_m)$ to f and an argument list $[t_1, \ldots, t_m]$. If each member of the argument list is in $\mathcal{F}_\gamma$ and δ is the curried type of sort γ of width $m + 1$, then t is in $\mathcal{F}_\gamma$.
- abstractions – are not allowed.

Definition A ***first order equality theorem of type*** α is a (possibly universally quantified) theorem $\Gamma \vdash t_1 = t_2$ where t_1 and t_2 are both members of $\mathcal{F}_\alpha$. A ***homogeneous list of first order equality theorems***
$[\Gamma_1 \vdash (l_1 : \alpha) = r_1; \ldots; \Gamma_n \vdash l_n = r_n]$ is a list of first order equality theorems, all of type α.

For any $\alpha \in \tau$ the first order matching, unification, and term ordering algorithms retain their properties in $\mathcal{F}_\alpha$, since there is an easy isomorphism between the set of first order terms and $\mathcal{F}_\alpha$.

Proof. Simple.

4 The components of completion.

We will consider the application of a substitution to a term and the computation of critical pairs as inference rules, since they are important components of completion that need to be inside the logic. As stated above, the rewriting of theorems is done by inference; we will not cover that here since full details can be found in Paulson's paper [Paulson83], in which he not only deals with term rewriting but also formula rewriting. Term rewriting suffices for HOL because HOL formulas are merely terms of boolean type.

4.1 Applying a substitution

The derived rule of inference **INST** applies a substitution to a theorem and is thereby the basis of term replacement in the HOL system. **INST** is already available in the HOL system.

```
INST     {A1, ..., An} |- tm     theta
         -----------------------------
         {A1, ..., An} |- theta(tm)
```

Since this is not a primitive facility in HOL, it is effected by

1. $\theta' \;=\; \{t/v \in \theta : v$ is not free in the assumptions $\}$
2. Converting $\theta' \;=\; \{t_1/v_1, \ldots, t_m/v_m\}$ into two sequences:
   - $(v_1, \ldots, v_m)$ — a generalization sequence
   - $(t_m, \ldots, t_1)$ — a specialization sequence

3. Generalizing (in left-to-right order) on the variables in $(v_1, \ldots, v_m)$ to get $\{A_1, \ldots, A_n\} \vdash \forall v_m \ldots v_1.\, tm$
4. Specializing (in left-to-right order) with all the terms in $(t_m, \ldots, t_1)$ to get $\{A_1, \ldots, A_n\} \vdash \theta(tm)$

To get the effect of applying a substitution, the last-generalized variable must correspond to the first term specialized. Further, all generalizations must be done first, so that a specialization doesn't get done and a subsequent generalization bind variables introduced by the specialization. Note that this routine depends on the substitution being idempotent.

4.2 Critical pair formation.

The heart of the completion algorithm is the production of critical pairs. As already mentioned, critical pairs must be theorems, hence they must follow from an inference rule. The split between inference and computation comes with the computation of occurrences of critical pairs, or overlaps. An *overlap* between rules r_1 and r_2 is a pair (θ, occ) of a substitution (produced by the first order unification algorithm) and an occurrence. The occurrence defines the path to the non-variable subterm of r_1 that unified with r_2. The following derived rule of inference, given two rules, and an overlap between the first rule and the second rule, returns the critical pair corresponding to that overlap.

```
CRITICAL_PAIR   A |- t1 = u1      B |- t2 = u2     (theta, occ)
               ----------------------------------------------------
                A U B |- (theta(t1))[occ := theta(u2)] = theta(u1)
```

(The notation $tm_1[occ := tm_2]$ denotes the term identical with tm_1 except that the subterm denoted by occ has been replaced by tm_2.)

In detail, **CRITICAL_PAIR** works as follows:

1. $r_1' \;=\;$ **INST** $\theta\ r_1\ (=\ A \vdash (\theta t_1) = (\theta u_1))$
2. $r_2' \;=\;$ **RENAME** r_2. **RENAME** is a derived rule of inference that merely changes all the free variables of a theorem to be new to the system.
3. $r_2'' \;=\;$ **INST** $\theta\ r_2'\ (=\ B \vdash (\theta t_2) = (\theta u_2))$
4. Generate v, a brand new variable of the right type, and form a template by replacing the subterm of θt_1 at occ by v:
   $template \;=\; (\theta t_1)[occ := v] = (\theta u_1)$

5. *critical_pair* $=$ **SUBST** $[r_2''/v]$ *template* r_1'

Example. [Huet80] We step through the inference rule. Assume that we have already derived $r1$ and $r2$. Notice that we will not have to rename $r2$ since its varaibles are already disjoint from those in $r1$.

```
r1 = |- f x (g x a) = h x
and
r2 = |- g b y = k y

#let [(theta,occ)] = overlap r1 r2 [];;
theta = [("b", "x"); ("a", "y")] : (term # term) list
occ = [2] : int list

#let r1' = INST theta r1;;
r1' = |- f b (g b a) = h b

#let r2'' = INST theta r2;;
r2'' = |- g b a = k a

#let (v,template) = mk_template r1' occ;;
v = "v" : term
template = "f b v = h b" : term

#let critical_pair = SUBST [(r2'',v)] template r1';;
critical_pair = |- f b (k a) = h b
```

The ML function *critical_pairs* that incorporates **CRITICAL_PAIR** has the type : *rule* $\rightarrow$ *rule* $\rightarrow$ *thm list*, and is thus a derived rule of inference. The ML function *kb* implements Huet's version of Knuth-Bendix completion [Huet81] and calls *critical_pairs*. It has type
: (*term* $\rightarrow$ *term* $\rightarrow$ *bool*) $\rightarrow$ *thm list* $\rightarrow$ *thm list*, hence is also a derived rule of inference. Its first argument should be a term ordering, and it checks that its second argument is a homogeneous list of first order equality theorems.

5 Example.

We use group theory, the factorial of the term rewriting world, for an example. The term order is the recursive path ordering with status [Dersh87]. Notice that the declared constants are polymorphic, as are all the returned theorems: the resulting set of theorems can be instantiated to any type by use of **INST_TYPE**.

```
#new_theory "group";
#new_infix ("op",':* -> * -> *') ;
#new_constant ("inv", ':* -> *');
#new_constant ("i", ':*');
#val e1 = new_axiom ("e1", '(i op x) = x')
##and e2 = new_axiom ("e2", '((inv x) op x) = i')
##and e3 = new_axiom ("e3", '((x op y) op z) = (x op (y op z))');
e1 = |- !x. i op x = x
e2 = |- !x. (inv x) op x = i
e3 = |- !x y z. (x op y) op z = x op (y op z)
() : void
#close_theory();
() : void

#kb (rpos status inv_op_i) ex1;
[|- i op x1 = x1,
 |- (inv x1) op x1 = i,
 |- (x1 op x2) op x3 = x1 op (x2 op x3),
 |- (inv x1) op (x1 op x2) = x2,
 |- x1 op i = x1,
 |- inv i = i,
 |- inv(inv x1) = x1,
 |- x1 op (inv x1) = i,
 |- x1 op ((inv x1) op x2) = x2,
 |- inv(x1 op x2) = (inv x2) op (inv x1)]
: thm list
Time: 16.2s
Intermediate theorems generated: 17436
```

We note that the non-logical version of the Knuth-Bendix completion algorithm took approximately 7 seconds to complete the group axioms.

In the HOL system, one develops *theories* by establishing some definitions and proving theorems about the constants introduced by the definitions. Once a theory is completed, it can be saved on disc and its definitions, theorems, and specialized proof procedures used in the development of other theories. The more theories that are developed, the higher level of support a person has in attempting to prove something. The completion procedure given here has many applications, among them the standard one of providing a decision procedure for equality for (some) equational theories. This would be an aid to those developing such theories [Gunter89], although there is typically much more than just rewrite rules to provide for a theory.

6 Conclusions and further research.

As can be seen, implementing completion in the logic is about twice as slow as an equally naive non-logical implementation. This is made bearable by virtue of the advantage conferred by having completion in the logic: any use of it will not require justification by "external" metatheorems; the user can rely on its output to be theorems.

There are two obvious paths to follow with this work: extend the first order work to equational completion and proof by consistency; and investigate completion in which the set of terms is not so restrictive. Another, due to an offhand remark by Tobias Nipkow, is to realize that *types* are a first order structure and to do completion on type equations. I don't know of any applications for this.

The impetus behind this research was to investigate the introduction of automatic theorem proving techniques into the HOL system — an inside-the-logic implementation of resolution or of term rewriting theorem proving [HsDer83] may be too slow to be useful; in that case the research of Miller and Felty [Miller, Felty86] on porting proofs between logics may be useful in translating refutation proofs to tactical proofs in a sound manner.

Acknowledgements.

I am grateful to Graham Birtwistle and Paliath Narendran: Graham has provided me with a great deal of financial support in the course of this work and suggested improvements in the paper; in the year that he was in Calgary, Dran taught me about term rewriting systems and has been a constant source of encouragement.

References

[Church40] Alonzo Church, *A Formulation of the Simple Theory of Types*, Journal of Symbolic Logic, Volume 5, 1940, pp. 56-68.

[Dersh87] Nachum Dershowitz, *Termination of Rewriting*, Journal of Symbolic Computation, Volume 3, 1987, pp. 69-116.

[Felty86] Amy Felty, *Using Extended Tactics to do Proof Transformations*, MSc. Thesis, Department of Computer and Information Science, University of Pennsylvania, December 1986, 85 pages.

[Gordon89] Michael Gordon, *The HOL System: Description*, Cambridge Research Center, SRI International, 1989.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, LNCS 78, Springer-Verlag, 1979.

[Gunter89] Elsa Gunter, *Doing Algebra in Simple Type Theory*, Technical Report MS-CIS-89-38, Logic & Computation 09, Department of Computer and Information Science, University of Pennsylvania, 1989.

[HsDer83] Jieh Hsiang and Nachum Dershowitz, *Rewrite Methods for Clausal and Non-Clausal Theorem Proving*, Proc. 10th ICALP, Springer LNCS 154, July 1983, pp. 331-346.

[Huet80] Gerard Huet, *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, nJACM, Volume 27, Number 4, October 1980, pp. 797-821.

[Huet81] Gerard Huet, *A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm*, Journal of Computer and System Sciences, Volume 21, 1981, pp. 11-21.

[KB70] Donald Knuth and Peter Bendix, *Simple Word Problems in Universal Algebras*, in Computational Problems in Abstract Algebra, edited by J. Leech, Pergamon Press, Oxford, 1970.

[KN84] Deepak Kapur and Paliath Narendran, *An Equational Approach to Theorem Proving in First Order Predicate Calculus* Computer Science Branch, General Electric Company, Schenectady, 1984.

[Miller] Dale Miller, *A Compact Representation of Proofs*, Studia Logica, Volume 56, Number 4, pp 347-370.

[Paulson83] Lawrence Paulson, *A Higher Order Implementation of Rewriting*, Science of Computer Programming, Volume 3, 1983, pp. 119-149.