A few exercises in theorem processing

F. Lockwood Morris

Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244–4100, United States

Abstract

The realization of inference rules as the primitive operations of a type “theorem” in a type-safe programming language that has so well served LCF and its descendants may, it is suggested, be of interest aside from any immediate context of theorem proving or hardware or software verification. Using the general “conversions” introduced by Paulson, a couple of simple programming exercises with theorem data, imitative of list processing, are presented. An example of a potentially useful notational definition in the HOL object language is given as an application.

Keywords: Theorem processing; Derived inference rules; Conversions; List processing

1. Introduction

The proof assistant LCF [3] and its descendants, notably HOL, share two outstanding characteristics. The first is that the underlying programming environment is a type-safe metalanguage, typically some dialect of ML [4,7], and that logical inference consists in the application of functions with result type “theorem.” If the built-in axioms and primitive inference rules are sound, and the introduction of new axioms is barred, then consistency is assured.

The second characteristic, which for most users may overshadow the first, is that individual proofs are all but invariably constructed in goal-directed fashion—that is, by a tree of backward moves, each notionally annotated by “It is enough to show...” Such steps are made by invoking metalanguage routines called “tactics,” (some recognizably inverse to single inference rules but many not), which not only construct the zero or more subgoals said to be “enough to show,” but are also prepared to carry out the forwards inferences substantiating that claim after the subgoals have been transformed into corresponding theorems.

The construction of routines that directly manipulate theorems as data with inference rules as operations, which might be called “theorem processing,” has, then, pretty much been left to a small group of tactic writers. This need not, however, be the case. A system like HOL is uniquely suited to be open-source software: because any item of type “theorem” produced is guaranteed actually to be a truth, independent contributions—some by not very expert practitioners—can be assimilated with only light central coordination to harmonize notation and suppress blatant
silliness. As a consequence, although the justifying applications of HOL to date have been mainly for verification of hardware and protocols, some of these applications have led to the codification of a surprising amount of elementary mathematics in diverse areas.

To the best of my knowledge, the most substantial such activity to date by far—a construction of the reals and the formalization of a portion of real analysis with applications which include the verification of floating-point algorithms for square root and logarithm—has been reported by Harrison [5]. By contrast, and to give an idea of typical small-scale mathematical theories in HOL, a theory in the HOL library ascribed to Laurent Théry develops the concept of greatest common divisor for natural numbers, and proves—besides some elementary lemmas—the existence and uniqueness of the gcd function, and culminates in a theorem that expresses the Euclidean algorithm in a familiar form:

$$\forall a. \text{gcd } a \ b = \text{if } a = 0 \text{ then } b \text{ else } \text{gcd}(b \mod a) \ a$$

all in about two hundred and fifty lines of definitions and (tactic-invoking) proofs—presumably, within an order of magnitude, the work of a few days. Building on this library theory, similar amounts of time and space suffice to define the dual concept of least common multiple and to prove, besides a number of other lemmas, that the function on natural numbers defined by

$$\text{lcm } a \ b = \text{if } 0 < a \land 0 < b \text{ then } a \times b \div \text{gcd } a \ b \text{ else } 0$$

yields the unique least common multiple.

Particularly in setting up usual mathematical notations which cannot be directly read off as terms of simple type theory (the example to be developed below is the iterated inequality notation, e.g. \( a \leq b < c \leq d \)) and in justifying the shorthand rules of calculation which they invite, the developer of the elements of some mathematical area may find reason to program in ML some quite elaborate rules of inference.

In one sense, a programmed inference rule is just a program—it is subject to mistakes, such as producing a theorem that is not the one desired, or looping, or crashing. From another point of view, however, an output of an inference rule (in a secure system like HOL) may be seen as the result of a self-verifying computation. A rule for arithmetic simplification might, for example, yield, for suitable inputs, the theorem \( \vdash 40 \times 9 \div 5 + 32 = 104 \). As this example suggests, computation by deduction may be thought of as having a relation to computation by an ordinary program analogous with the relation of a printing calculator to the kind where one merely punches buttons and views the numeric answer.

Something should be said to connect the topic of the present note to the work of John Reynolds (besides the debt, acknowledged by Gordon et al. [4], that the design of ML owes to his GEDANKEN [8]). I take it that a major theme for Reynolds has been proving what it is that programs compute—both program-by-program by way of Hoare-style assertions, and generalities deduced by reasoning about the denotational semantics of a programming language. To the extent that the view of inference rules as self-checking programs may afford a sort of end run around the necessity of proving program correctness, “theorem processing” would seem to be relevant to this theme, in at least a contrarian way. Moreover, although it is as yet (so far as I know) purely a somewhat plausible conjecture that computation by derivation might find applications outside the improvement and extension of proof assistants—in some class of computations calling less for speed of execution than for the greatest possible confidence in the results—it may nevertheless be considered an interesting variation on the craft of (correct) programming, being able to imitate some familiar motifs from list processing, as will be seen below.

The logic of HOL is a form of the simple theory of types, differing from the presentation given by Church [1] in that type polymorphism has been made part of the object language. That is, it is a typed lambda calculus (the types being allowed to contain instantaneous type variables), equipped in the beginning with no other constants than polymorphic equality and, on the type \textbf{bool}, the connective \( \supset \) of implication. (Formulæ are the terms of type \textbf{bool}, and material equivalence is equality at that type.) The general form of a theorem in HOL, as in LCF, is the \textit{sequent}, consisting of a set of assumptions \( A_1, \ldots, A_k \), and a \textit{conclusion} \( B \); although such a sequent has the same logical force as the implication \( A_1 \land \cdots \land A_k \supset B \), it accords with the natural-deduction style of proof to make rules and tactics focus on the top-level structure of the conclusion, leaving the assumptions in the background. A \textit{goal}, or would-be theorem, has of course the same parts as a theorem, but may be created at will. A rule drawing an inference from two or more theorems typically unites the assumption sets of its arguments to form that of its result. An empty assumption set is generally not written; the examples here will make no particular use of assumptions, and so we may at times write “\( \vdash P \)” where a more general “\( A \vdash P \)” is really meant.
There are a handful of primitive axioms and rules of inference, the latter including β-conversion, rules of substitution and type instantiation, and modus ponens. Two primitive definition principles permit the introduction of a new constant, equated to a term in which it does not occur, and of a new type in one-to-one correspondence with any given subset of an existing type. Besides bool there is a primitive type ind of “individuals” with an axiom of infinity, and of course higher types can be built up with the function arrow. The Introduction to HOL gives a set-theoretic model for all this, and proves the soundness of the primitive rules [2].

At this point the axiomatic method, in the sense stigmatized by Russell, is once and for all abandoned in favor of “honest toil”: the other familiar connectives and quantifiers of predicate logic, and on top of this the type of natural numbers, with the induction principle and facility of definition by primitive recursion, are all introduced by definitions, and the relevant properties proved as theorems. I stress this point because in the remainder of this note, in which space forbids me from giving a connected description of the HOL system, various theorems and rules are said to be “provided by HOL,” risking giving the impression of this being just a gee-whiz story about a piece of software, with a new feature introduced to meet every difficulty. While that may not be an entirely unjustified criticism of the present exposition, the reader is asked at least to realize that the theorems are intended to be recognized as true and the rules as sound, and that the theorems have indeed been proved in the process of building the system, and the rules composed from sound primitives.

In Paulson’s 1983 paper “A Higher-Order Implementation of Rewriting” [6] are to be found much more serious examples of theorem processing (though not there so called) than in the present note; he constructs powerful rewriting tactics, which provide the main engine of most LCF and HOL proofs, from a primitive rule for replacing equals by equals. After two decades, this may still be as substantial a demonstration of the utility of functional programming techniques as has appeared.

The next section will recapitulate some basic parts of Paulson’s paper (though not its capsule description of ML itself, which it is hoped may, with the passage of time, have become widely readable). Additional notations and functions specific to HOL will, it is intended, all be explained as they arise. Subsequent sections will display a couple of easy theorem-processing exercises imitative of list processing, and will show how, by suitable definitions of some operators, the HOL object language can be endowed with something like the usual mathematical notation for multi-term inequalities, and also how the appeals to transitivity implicit in the notation can be automated.

2. Conversions

A major contribution of “A Higher-Order Implementation of Rewriting” is the introduction of the general notion of “conversion.” A conversion is a function applicable to some class of terms which, whenever its application to a term succeeds, yields an equational theorem \( \vdash t = t' \). (All the theorems produced by the conversions to be considered here will have genuinely empty assumption sets.) HOL abbreviates as \( \texttt{conv} \) the type \( \texttt{term} \rightarrow \texttt{thm} \) shared by all conversions.

Conversions take their name from β-conversion; in HOL this is the primitive rule \texttt{BETA_CONV}, which maps any term \( t \) of the form \( (\lambda x. M)N \) to the theorem \( \vdash (\lambda x. M)N = \texttt{M}[N/x] \). Paulson observed the great utility of the same format for a large class of inference rules. In particular, he introduced the \( \texttt{thm} \rightarrow \texttt{conv} \) function, which he called \texttt{REWRITE_CONV} (now HOL’s \texttt{REWR_CONV}), that turns any theorem having the form of a universally quantified equation, say \( \vdash \forall x. t = u \), into a conversion applicable to any term \( t' \) that is an instance of \( t \) under some substitution for the vector of variables \( \vec{x} \), and returning the theorem \( \vdash t' = u' \), where \( u' \) is the corresponding instance of \( u \).

Especially attractive are conversions applicable to Boolean terms, producing logical equivalences. Let \( c \) be such a conversion and \( c\ p = \vdash p = q \) an instance of its use; then the forward direction of the biconditional justifies the deduction of \( \vdash q \) from \( \vdash p \) (suppressing an arbitrary assumption set), while the backward direction justifies the reduction of a goal with conclusion \( p \) to one with conclusion \( q \). HOL provides \texttt{CONV_RULE} : \texttt{conv} \rightarrow \texttt{thm} \rightarrow \texttt{thm} to produce the corresponding inference rule from any conversion (supposed to be applicable to some Boolean terms), and \texttt{CONV_TAC} : \texttt{conv} \rightarrow \texttt{tactic} to produce the corresponding tactic. Thus, for example, \texttt{CONV_RULE (REWR_CONV (\vdash \forall x. t = u))} will infer, from any theorem whose conclusion is an instance of \( t \), the corresponding instance of \( u \); \texttt{CONV_TAC} with the same conversion as the argument will reduce a goal whose conclusion is an instance of \( t \) to one with the corresponding instance of \( u \) in its place.

(One may, of course, wish to make similar use of a theorem which is a quantified one-way implication, say \( \vdash \forall x. t \supset u \). HOL provides the rule \texttt{MATCH_MP} : \texttt{thm} \rightarrow \texttt{thm} \rightarrow \texttt{thm} (whose construction, as it happens, Paulson
describes) such that MATCH_MP (⊢ ∀x. t ⊔ u) has the same effect as would CONV_RULE (REWR_CONV (⊢ ∀x. t = u)). The related tactic, MATCH_MP_TAC, by contrast, given the same implicative theorem, yields a tactic which will reduce a goal whose conclusion is an instance of u to one which is the corresponding instance of t.)

The most important higher order function for combining conversions is sequential composition, written as infixed THENC (plain THEN having been taken for a similar composition of tactics): if c₁ and c₂ are conversions and t a term such that c₁ t = ⊢ t = t' and c₂ t' = ⊢ t' = t'' then (c₁ THENC c₂) t = ⊢ t = t''.

The composition of conversions of course appeals to the transitivity of equality; there is naturally also an identity conversion exploiting reflexivity: ALL_CONV t = ⊢ t = t, for any term t.

Almost equally prominent are functions mapping conversions to conversions which form the building blocks for general rewriting: we take note here of RATOR_CONV and RAND_CONV. If c is a conversion and t a term such that c t₁ = ⊢ t₁ = t₁', then for any term, t₁ t₁₂ say, whose rator is t₁, we will have

\[ \text{RATOR_CONV } c (t₁ t₂) = \vdash t₁ t₂ = t₁' t₂. \]

Similarly, if c t₂ = ⊢ t₂ = t₂',

\[ \text{RAND_CONV } c (t₁ t₂) = \vdash t₁ t₂ = t₁ t₂'. \]

(There is also ABS_CONV, not needed here, similarly passing a conversion down from a λ-abstraction to its body; since all non-atomic HOL terms are built up by just application and abstraction—surface syntactic appearance to the contrary—RAND_CONV, RATOR_CONV, and ABS_CONV provide a complete set of decompositional conversions.\(^2\))

To show some of these pieces working together, we write a definition of BINOP_CONV (already provided by HOL) which should satisfy

\[ \text{BINOP_CONV } c (f t₁ t₂) = \vdash f t₁ t₂ = f t₁' t₂', \]

where c t₁ = ⊢ t₁ = t₁', \(i = 1, 2\). The definition is simple, but we make two steps of it because LAND_CONV, acting on the left operand of a binary operator, is sometimes useful by itself:

\[ \text{fun LAND_CONV } c = \text{RATOR_CONV } (\text{RAND_CONV } c); \]

(Observe that RATOR_CONV (RAND_CONV c) (f t₁ t₂) will be ⊢ f t₁ t₂ = f t₁' t₂ provided that (RAND_CONV c) (f t₁) = ⊢ f t₁ = f t₁'.)

\[ \text{fun BINOP_CONV } c = \text{LAND_CONV } c \text{ THENC RAND_CONV } c. \]

Since terms whose main connective is an infix binary operator are construed by HOL as curried applications, for infix \(\oplus\) the effect of BINOP_CONV appears as

\[ \text{BINOP_CONV } c (t₁ \oplus t₂) = \vdash t₁ \oplus t₂ = t₁' \oplus t₂'. \]

Paulson shows how the function of type \(\text{thm list} \rightarrow \text{conv}\) that HOL calls REWRITE_CONV, a general-purpose rewriting conversion parameterized by a list of theorems, may be built up on REWR_CONV as a basis, using RATOR_CONV, RAND_CONV, ABS_CONV, and recursion. REWRITE_CONV \([th₁ \ldots thₙ]\) is a conversion which persistently searches its argument term for subterms that it can rewrite by any of \(th₁, \ldots, thₙ\) (or by any of a set of standard simplifications, such as \(\vdash \forall x. x \land T = x\)), continuing until no subterm of the result can be further rewritten—or possibly forever if \(th₁, \ldots, thₙ\) have been chosen imprudently. As with REWR_CONV, composing CONV_TAC and CONV_RULE with REWRITE_CONV yields two functions of types \(\text{thm list} \rightarrow \text{tactic}\) and \(\text{thm list} \rightarrow \text{thm} \rightarrow \text{thm}\) which HOL calls REWRITE_TAC and REWRITE_RULE. Variants whose names start with “PURE_” omit the standard simplifications.

\(^2\) A referee remarks: “System builders not in the HOL or LCF tradition can’t begin to understand how important for implementation clarity and ease it is that there are only four syntactic forms [i.e., constants, variables, applications, and abstractions]. One regularly sees a complete ad hoc mess made of syntax representation in other systems (e.g. decision procedures—not only deductive theorem provers), with much unnecessary multiplication of syntactic kinds.”
3. Inference as “theorem processing”

The only analogy between theorems and data structures to be used in the examples presented here is entirely trite: to know (or to have proved) several propositions is the same as to know their conjunction. Thus ordered pairs in general may be modeled by binary conjunctions, and finite lists by right-associated multiple conjunctions, with a vacuous proposition as their terminator—call these “cj-lists.” The constructor and selectors are already present as the HOL rules \textsc{CONJ} ($\wedge$-introduction) and \textsc{CONJUNCT1}, \textsc{CONJUNCT2} ($\wedge$-elimination).

Introduce a constant \textsc{NOF} to serve as an explicit “empty cj-list” by the definition

\[
\textsc{NOF} = \text{df} \top.
\]

Using \textsc{T} itself for the purpose would run the risk that the simplifications in HOL's default rewriting could rub out "$\wedge \top$" from the ends of formulae. “\textsc{NOF},” reminiscent of “\textsc{NIL},” is short for “not false”—before the wisdom of a separate definition was seen, “$\sim F$” was briefly used as a stopgap in the same role.

(There is no reason to follow here the concrete invocations by which one makes definitions, introducing new constants into the HOL object language. The basic logic accepts, as a definition, any equation whose whole left-hand side is its only occurrence of a new constant; formally this is an axiom whose consistency is guaranteed. Much of the infrastructure of HOL as a usable system consists in the programmed inferences by which various classes of more elaborate definitions, notably definitions by primitive recursion, may be reduced to this form; in general the user is returned what may be called a “defining theorem” rather than the new primitive axiom. Definitions will be shown here with “$=$\text{df}” where practical, otherwise they are described by showing the defining theorem; in either case it may be arranged that the same sequence of characters that constitutes the new object language constant is also bound, as an ML identifier, to the defining theorem; thus we now have \textsc{NOF} naming the theorem \thm{\textsc{NOF} = \top}.

Although inference in general throws information away, there are so many inferences which do not that it is attractive to do as much theorem processing as practicable with Boolean conversions. As examples, we may imitate the first two exercises of anybody’s introduction to list processing, \textit{reverse} and \textit{append}, defining conversions \textsc{CJ\_REVERSE\_CONV} and \textsc{CJ\_APPEND\_CONV}.

The particular theorems needed for the conversions developed in this note are of such triviality that it is easier to give a HOL declaration to actually produce each one than to invent variations on “is easily shown to hold.”

HOL’s \texttt{prove: term \times tactic \rightarrow thm} is the usual way of recording, for non-interactive replaying, a tactic proof that has (usually) been found interactively: if a tactic $\Phi$ (ordinarily a composition from many individual tactics by \texttt{THEN} and other connectives) successfully proves a Boolean term $p$ (without assumptions) as an interactive goal, then \texttt{prove} ($p$, $\Phi$) $\vdash p$. HOL provides \texttt{TAUT\_TAC}, which can prove any propositional tautology; although in principle an exponential-time abomination to be shunned, \texttt{TAUT\_TAC} is fast and practical for formulae like \texttt{conj_pop_lem} below.

For the sake of brevity, in the samples of ML given here, any condition which is detected, but ought not to arise, is met by “\textit{fail ()},” which raises an exception from HOL.

\textsc{CJ\_REVERSE\_CONV} works by the usual Slinky-like U-turn of one list onto another, initially empty. Introduce a hand-built conversion \texttt{PUT\_NOF\_CONV} to pair (by conjunction) an empty cj-list with any proposition:

\[
\begin{align*}
\texttt{val} & \quad \texttt{put\_nof\_lem} = \texttt{prove (``\forall x . x = (x \land \textsc{NOF})\''}, \texttt{REWRITE\_TAC [\textsc{NOF}]})
\texttt{;} \\
\texttt{val} & \quad \texttt{DROP\_NOF\_CONV t} = \texttt{SPEC t put\_nof\_lem}
\end{align*}
\]

(HOL’s \texttt{SPEC: term \rightarrow thm \rightarrow thm} being the rule for specialization of a universally quantified theorem) and one to drop an empty cj-list as first component of such a pair:

\[
\begin{align*}
\texttt{val} & \quad \texttt{drop\_nof\_lem} = \texttt{prove (``\forall x . \textsc{NOF} \land x = x\''}, \texttt{REWRITE\_TAC [\textsc{NOF}]})
\texttt{;} \\
\texttt{val} & \quad \texttt{DROP\_NOF\_CONV t} = \\
& \quad \texttt{if rand (rator t) = ``\textsc{NOF}''}
\texttt{then SPEC (rand t) drop\_nof\_lem else fail ()};
\end{align*}
\]

and one to iterate the basic step of reversing as long as it is applicable:
Rewriting the conjunction of two cj-lists (the first of them non-empty) with \texttt{conj} this yields of cj-lists which may be appended by recursion. Thus we may write whose left conjunct is the correct first element of the appended cj-list, and whose right conjunct again models a pair

Then the reversing conversion is simply

\texttt{val CJ\_REVERSE\_CONV} = \texttt{PUT\_NOF\_CONV}

\texttt{THENC CJ\_REV THENC DROP\_NOF\_CONV}.

A conversion \texttt{CJ\_APPEND\_CONV} for appending cj-lists, mapping a conjunction of two such lists into the theorem asserting its equivalence to one combined cj-list, may be written in imitation of the usual recursive definition of append as follows: the argument term is supposed to be of the form \(l \land r\), where \(l\) and \(r\) are both cj-lists; when \(l\) is of the form \(A \land l'\), we have only to set \(A\) aside as the first element of the final result, and solve the sub-problem of appending \(l'\) to \(r\).

We happen to need the mirror image of the pre-proved theorem, \texttt{CONJ\_ASSOC}, asserting the associativity of \(\land\):

\texttt{val conj\_assoc\_lem} = \texttt{GSYM CONJ\_ASSOC};

this yields

\(\vdash \forall t_1 t_2 t_3 . (t_1 \land t_2) \land t_3 = t_1 \land t_2 \land t_3\).

Rewriting the conjunction of two cj-lists (the first of them non-empty) with \texttt{conj\_assoc\_lem} produces a conjunction whose left conjunct is the correct first element of the appended cj-list, and whose right conjunct again models a pair of cj-lists which may be appended by recursion. Thus we may write

\texttt{fun CJ\_APPEND\_CONV} \(t\) =

\texttt{(if is\_conj (rand (rator \(t\)))

then \texttt{REWR\_CONV conj\_assoc\_lem THENC RAND\_CONV CJ\_APPEND\_CONV

else \texttt{DROP\_NOF\_CONV}) \(t\)).

Alternatively, \texttt{CJ\_APPEND\_CONV} may be more briefly programmed with the help of reversing, as:

\texttt{val CJ\_APPEND\_CONV} = \texttt{LAND\_CONV CJ\_REVERSE\_CONV

\texttt{THENC CJ\_REV THENC DROP\_NOF\_CONV}.

4. Multi-term inequalities

We may apply cj-lists, together with HOL’s facilities for definition and for mild extensions to its object language syntax, to attack one of the minor nuisances that beset beginning programmers: the necessity to write, say, \(l \leq m \land m < r\) rather than the usual mathematical shorthand \(l \leq m < r\). (Call the shorthand forms, in their HOL realization, (inequality) chains.)

Introduce a suffix operator \(\vdash\), infix operators \(\leq\), \(<\), \(\geq\); and \(>\) (the decorative colons allow us to avoid any complications about overloading here) and a prefix operator \(|\vdash|\), with precedences chosen the same for the four infixes, which moreover are to associate to the right, lower for \(|\vdash|\) and higher for \(\vdash\), all to the purpose that, for example, \(\vdash a \vdash b \leq c \vdash d\vdash\) should have for successive right-hand sub-phrases \(a \vdash b \leq c \vdash d\vdash\), \(b \leq c \vdash d\vdash\), \(c \vdash d\vdash\), and \(d\vdash\).

Now define \(|\vdash|\) as a unary function of type \texttt{nat \rightarrow nat \times bool}, \(\leq\), \(<\), \(\geq\), and \(>\) as binary functions of type \texttt{nat \times bool \rightarrow nat \times bool}, and \(|\vdash|\) as a unary function of type \texttt{nat \times bool \rightarrow bool} as follows:

\(n : \texttt{num}) \vdash |\vdash n, \texttt{NOF})

\(m \leq (n, P) = \texttt{df} (m, m \leq n \land P)

\(m < (n, P) = \texttt{df} (m, m < n \land P)

\(m \geq (n, P) = \texttt{df} (m, m \geq n \land P)\)
\[ m >: (n, P) \equiv (m, m > n \land P) \]
\[ |: (n : \text{num}, P : \text{bool}) \equiv P. \]

Without further ado, any chain, for example
\[
\text{val } t_1 = \text{`|: } a \leq b <: c \leq d : |\text{`|};
\]
is now a meaningful propositional term; to translate from the new notation into a cj-list of two-term inequalities, we have only to rewrite it with the definitions of the new symbols:
\[
\text{val } \text{CHAIN CJ CONV} = \text{PURE_REWRITE CONV \[ |:, \leq, <:, \geq, >:, |\].}
\]

Now \text{CHAIN CJ CONV } t_1, for example, evaluates to the theorem
\[
\vdash |: a \leq b <: c \leq d : | = a \leq b \land b < c \land c \leq d \land \text{NOF}.
\]

4.1. An alternative scheme

There is an alternative way to introduce multi-term inequalities which uses heavier machinery, but may be thought more natural. HOL has been equipped, as a natural generalization of the machinery developed for the construction of the natural numbers, with a facility for the convenient definition of recursive datatypes in the object language—very similar to the datatypes of ML, type-homogeneous lists are the leading example—and for accepting definitions of functions by primitive recursion over such types. There is no need to show the concrete details of invoking the facility; suffice it to say that we may define a datatype “\text{chain}” with the unary constructor \(|:|\) taking a \text{num} argument, and four binary constructors \(\leq, <:, \geq, >:\) each taking one \text{num} and one \text{chain} argument. (The same syntactic extensions as were supposed before may be used to make postfix and infix operators of the constructor names.)

Then we can put off all the semantics for chains onto a recursive definition of the prefix operator \(|:|\), now of type \text{chain} → \text{bool}. First one defines by cases (i.e., primitive recursion that doesn’t recurse) an auxiliary function \text{CHF} (for “chain front”) of type \text{chain} → \text{num}, for which the defining theorem states
\[
\forall m : \text{num} \ . \ z : \text{chain} \ . \ (\text{CHF}(m |:) = m) \land
\begin{align*}
(\text{CHF}(m \leq: z) = m) & \land (\text{CHF}(m <: z) = m) \land \\
(\text{CHF}(m \geq: z) = m) & \land (\text{CHF}(m >: z) = m).
\end{align*}
\]

One may then define \(|:|\) by primitive recursion so as to have the defining theorem
\[
\vdash \forall m z . \ . \ (|: (m |:) = T) \land
\begin{align*}
(\ |: (m \leq: z) = (m \leq \text{CHF} z \land |: z)) & \land (\ |: (m <: z) = (m < \text{CHF} z \land |: z)) \land \\
(\ |: (m \geq: z) = (m \geq \text{CHF} z \land |: z)) & \land (\ |: (m >: z) = (m > \text{CHF} z \land |: z)).
\end{align*}
\]

Under this treatment, \text{CHAIN CJ CONV} may be defined by
\[
\text{val } \text{CHAIN CJ CONV} = \text{PURE_REWRITE CONV \[ |:, \text{CHF}\]}
\]
and will yield the same behavior as before.

5. Conversions on chains

Conversions acting on chains are typically easier to program as manipulations of the equivalent cj-lists, since we already have some infrastructure for that; thus we need an inverse of \text{CHAIN CJ CONV}, to recover a chain from a non-empty cj-list to which it was, or might have been, converted by \text{CHAIN CJ CONV}. Rather than make an independent

\(^3\) The author is in no position to prefer either of the two approaches described in this section. In a project, still very much under development, to support elementary category theory in HOL, chain-like notations have been introduced both for sequences of objects and arrows within any one category, and for “horizontal” compositions of functors and natural transformations linking several categories. At present there seem to be sufficient reasons for following the separate-datatype approach to the semantics of the former, but the something-paired-with-a-proposition approach for the latter. There is a similarity here to the distinction between what have come to be called “deep embedding” and “shallow embedding,” if not actually an instance of it.
recursive definition of this conversion, call it \( \text{CJC\_CHAIN\_CONV} \), it makes sense to foresee what chain term will appear in the answer and apply \( \text{CHAIN\_CJC\_CONV} \) to that, obtaining an equation that needs only to have its sides interchanged (by the rule \( \text{SYM} \)) to be what we need. The ML function to construct the desired chain term is most uninteresting, but we may exhibit it to make sure. (The caret symbol \( ^\wedge \), which should be followed by an ML expression of type \( \text{term} \), denotes anti-quotation inside the HOL quotes \( "\ldots " \). A dollar sign before a HOL infix operator makes it an ordinary identifier for the nonce.)

\[
\begin{align*}
\text{fun } \text{chain\_of\_cj\_list } t &= \text{if } t = "\text{NOF}" \text{ then fail } () \text{ else let val } (a, b, rl, rest) = \\
& (\text{rand (rator (rand (rator t))), rand (rator (rand (rator t)))), \text{rand t}); \\
& \text{val } rl' = \text{if } rl = "\text{$\leq$}" \text{ then } "\text{$\leq:$}" \text{ else if } rl = "\text{$<$}" \text{ then } "\text{$<$:}" \text{ else if } rl = "\text{$\geq$}" \text{ then } "\text{$\geq:$}" \text{ else if } rl = "\text{$>$}" \text{ then } "\text{$>$:}" \text{ else fail } () \text{ in } \text{if } rest = "\text{NOF}" \text{ then } " : | (\text{\textasciicircum \textcircled{rl'} \textasciicircum a} (\text{\textasciicircum b} : |))" \text{ else if } b = \text{rand (rator (rand (rator rest)))} \text{ then } \\
& " : (\text{\textasciicircum \textcircled{rl'} \textasciicircum a} (\text{chain\_of\_cj\_list rest}))" \text{ else fail } () \text{ end.}
\end{align*}
\]

Now it is easy to define

\[
\text{fun } \text{CJC\_CHAIN\_CONV } cj = \text{SYM (CHAIN\_CJC\_CONV (chain\_of\_cj\_list cj))}.
\]

We can immediately write down an equivalent of \( \text{CJC\_APPEND\_CONV} \) for (the conjunction of two) chains:

\[
\text{val } \text{CHAIN\_APPEND\_CONV } = \text{BINOP\_CONV CHAIN\_CJC\_CONV THENC CJ\_APPEND\_CONV THENC CJ\_CHAIN\_CONV}.
\]

Evaluating, for example, \( \text{CHAIN\_APPEND\_CONV} ("\text{"t}_1 \wedge | : d \leq: e : | "") \) yields the theorem

\[
\vdash | : a \leq: b <: c \leq: d : | \wedge | : d \leq: e : | = | : a \leq: b <: c \leq: d \leq: e : | .
\]

The usual application of \( \text{CHAIN\_APPEND\_CONV} \) might be in the following two-argument rule for inferring an appended chain from its two pieces:

\[
\text{fun } \text{CHAIN\_APPEND\_RULE } ch_1 \ ch_2 = \\
\text{CONV\_RULE CHAIN\_APPEND\_CONV (CONJ ch_1 \ ch_2)}.
\]

Reversing chains requires \( < \text{ and } > \) to be interchanged:

\[
\text{val } \text{less\_great\_lem } = \text{prove (}
\text{"(\forall a \ b . a \leq: b = b \geq: a) \wedge (\forall a \ b . a <: b = b >: a) \wedge (\forall a \ b . a >: b = b <: a)" , numLib.\text{ARITH\_TAC});}
\]

(HOL’s \text{ARITH\_TAC} embodies a partial decision procedure for Presburger arithmetic—a sledgehammer to crack this peanut.)

\[
\text{val } \text{LESS\_GREAT\_CONV } = \text{PURE\_ONCE\_REWRITE\_CONV [less\_great\_lem];}
\]

\[
\text{val } \text{CJC\_INVERT\_CONV } = \text{CJC\_REVERSE\_CONV THENC LESS\_GREAT\_CONV;}
\]

using which, we have simply:

\[
\text{val } \text{CHAIN\_REVERSE\_CONV } = \text{CHAIN\_CJC\_CONV THENC CJ\_INVERT\_CONV THENC CJ\_CHAIN\_CONV}.
\]
The variants of rewriting containing “ONCE” do not rework any replaced subterm, a precaution necessary here to avoid looping.) Evaluating, for example, CHAIN\_REVERSE\_CONV \( t_1 \) yields

\[ \vdash [a \leq b < c \leq d :] = [d \geq c > b \geq a :] . \]

6. An application of chains

It may be worthwhile to begin to indicate how a “notation” like inequality chains as developed here might serve some useful purpose in a larger context of machine-assisted reasoning. Let us restrict attention to ascending chains, those containing \( \leq : \) and \( < : \) only; these (and their dual, descending chains) are the kind that appear in mathematical practice, because such an \( n \)-term inequality makes immediately evident \( n(n - 1)/2 \) pairwise inequalities.

It is very straightforward to provide a function which will extract, as a theorem, any pairwise inequality entailed by an ascending chain. To be precise, and to conform to what seems to be the HOL style of using conversions wherever feasible, we may develop an ML function \( \text{ASC\_CHAIN\_CONV} \) of type \( \text{thm} \rightarrow \text{conv} \) such that

\[
\text{ASC\_CHAIN\_CONV} \left( \vdash [a_1 \cdots \leq \cdots < \cdots a_n :] \right) t = \vdash t = T,
\]

where \( t \) is any \( a_i \leq a_j \) or \( a_i < a_j \), where \( i < j \) and, in case the desired inequality is strict, there occurs at least one \( <: \) between \( a_i \) and \( a_j \) in the chain.

By making two applications of \( \text{CJ\_INVERT\_CONV} \), so that a \( \text{cj-list} \) undergoes a temporary left–right reflection, we may easily now discard both its unwanted end sections:

\[
\text{fun} \ \text{DISCARD\_ENDS} \ a \ b \ cj =
\text{let val} \ cj' = \text{CONV\_RULE} \ \text{CJ\_INVERT\_CONV} \ (\text{DISCARD\_TO\_NUM} \ a \ cj) \ \text{in} \\
\quad \text{CONV\_RULE} \ \text{CJ\_INVERT\_CONV} \ (\text{DISCARD\_TO\_NUM} \ b \ cj').
\]

The following rule, \( \text{GEN\_LE\_TRANS} \), will make an inference using any one of the four HOL-supplied transitivity theorems for \( \leq \) and \( < \). It uses the abbreviation \( \text{is\_bapp} \) for recognizing a “binary application,” that is, the curried application of a given constant to two arguments.

\[
\text{fun} \ \text{is\_bapp} \ c \ t = \text{is\_comb} \ t \ \text{andalso} \ \text{is\_comb} \ (\text{rator} \ t) \ \text{andalso} \ \text{rator} \ (\text{rator} \ (\text{rator} \ (\text{concl} \ mn))) = c; \\
\text{fun} \ \text{GEN\_LE\_TRANS} \ mn \ np = \text{MATCH\_MP} \\
\quad (\text{if} \ \text{is\_bapp} \ "$\leq" \ (\text{concl} \ mn) \ \text{then} \\
\quad \quad (\text{if} \ \text{is\_bapp} \ "$\leq" \ (\text{concl} \ np) \ \text{then} \ \text{LESS\_EQ\_TRANS} \\
\quad \quad \quad \text{else if} \ \text{is\_bapp} \ "$<" \ (\text{concl} \ np) \ \text{then} \ \text{LESS\_EQ\_LESS\_TRANS} \ \text{else} \ \text{fail} () \\
\quad \quad \quad \text{else if} \ \text{is\_bapp} \ "$<" \ (\text{concl} \ mn) \ \text{then} \\
\quad \quad \quad \quad (\text{if} \ \text{is\_bapp} \ "$\leq" \ (\text{concl} \ np) \ \text{then} \ \text{LESS\_LESS\_EQ\_TRANS} \\
\quad \quad \quad \quad \text{else if} \ \text{is\_bapp} \ "$<" \ (\text{concl} \ np) \ \text{then} \ \text{LESS\_TRANS} \ \text{else} \ \text{fail} () \\
\quad \quad \quad \quad \text{else} \ \text{fail} () \\
\quad \quad \text{else} \ \text{fail} () \\
\quad \text{\text{(CONJ mn np)}).}
\]
Reduce a chain, in cj-list form, to one inequality between its ends:

```ml
fun ASC_CJ_CONDENSE cj = if concl cj = "NOF" then fail ()
  else if rand (concl cj) = "NOF" then CONJUNCT1 cj
  else GEN_LE_TRANS (CONJUNCT1 cj) (ASC_CJ_CONDENSE (CONJUNCT2 cj)).
```

Weaken $<$ to $\leq$ if that is what is wanted, using the HOL-supplied theorem $\vdash \forall m \ n. m < n \supset m \leq n$, known as LESS_IMP_LESS_OR_EQ:

```ml
fun WEAKEN_IF_NEQ t th =
  if t = concl th then th
  else if is_bapp "$\leq$" t andalso is_bapp "$<" (concl th)
    then WEAKEN_IF_NEQ t (MATCH_MP LESS_IMP_LESS_OR_EQ th)
  else fail ()
```

Introduce a theorem for use in inferring an equivalence, such as a Boolean conversion should deliver, from the computed pairwise inequality:

```ml
val IMP_EQ_T = prove ("\forall t : bool . t \supset (t = T) " , REWRITE_TAC []).
```

All these pieces fit together to yield ASC_CHAIN_CONV:

```ml
fun ASC_CHAIN_CONV t ch =
  if is_bapp "$\geq$" t orelse is_bapp "$\geq$" t
    then CONV_RULE LESS_GREAT_CONV
        (ASC_CHAIN_CONV (rand (concl (LESS_GREAT_CONV t)))) (ch)
  else MATCH_MP IMP_EQ_T (WEAKEN_IF_NEQ t (ASC_CJ_CONDENSE
        (DISCARD_ENDS (rand (rator t)) (rand t))
        (CONV_RULE CHAIN_CJ_CONV ch))))
```

Example: HOL's axiom scheme ASSUME yields from any Boolean term $t$ the theorem $\{ t \} \vdash t$; using this as a cheap way to get a chain theorem, we may evaluate ASC_CHAIN_CONV ("b \leq d") (ASSUME $t_1$) to obtain $\{ : a \leq : b \leq : c : d : \} \vdash b \leq d = T$.

Better than writing out a closely similar definition of DESC_CHAIN_CONV, which should draw pairwise inferences from chains containing only $\geq$: and $>;$, is to use the dualizing conversion CHAIN_REVERSE_CONV, giving much more briefly:

```ml
fun DESC_CHAIN_CONV t ch =
    ASC_CHAIN_CONV t (CONV_RULE CHAIN_REVERSE_CONV ch).
```

7. Speculation and conclusions

The theorem-processing examples shown here have been analogous to the tamest first-order sort of list-processing routines—nothing to make a functional programmer breathe any quicker. As a beginning guess at what more exciting possibilities there might be, the inference rule MATCH_MP discussed in Section 2 above on conversions (which it is not) would seem to offer a close analogy in theorem processing to function application. If so, there should be an analogue of function composition, call it CMP : thm $\rightarrow$ thm $\rightarrow$ thm, satisfying, for pairs of theorems

$\theta_1 = \forall x . A_1 \supset B_1$,
$\theta_2 = \forall y . A_2 \supset B_2$
such that instances of $B_1$ under instantiation of the variables in $\bar{x}$ are instances of $A_2$ under instantiation of the variables in $\bar{y}$, and for $\tau$ an instance of $A_1$,

$$\text{MATCH}_\text{MP} (\theta_2 \text{CMP} \theta_1) \ tau = \text{MATCH}_\text{MP} (\theta_2 (\text{MATCH}_\text{MP} (\theta_1 \ tau))).$$

This rule, which might perhaps more descriptively be called “syllogism,” seems easy enough to construct, but to date I have encountered no particularly compelling application for it.

The examples which have been given here do not, it must be confessed, go very far towards substantiating the notion that theorem processing may have the potential to develop into a difficult but rewarding offspring of functional programming for computations in general. They may, however, give encouragement to those who use LCF-descended proof assistants for substantial applications (and who, it is to be hoped, have already discovered the advantages of defining specialized tactics, by combining and parametrizing those which the system supplies, to facilitate the idiosyncratic patterns of proof tasks which they encounter) by suggesting that theorem processing may be of use for building application-specific tools less closely tied to the transactions of interactive proving than are most tactics.

References