HOL Zero's Solutions for Pollack-Inconsistency

Mark Adams[1,2]

[1] Proof Technologies Ltd, Worcester, UK
[2] Radboud University, Nijmegen, The Netherlands

Abstract. HOL Zero is a basic theorem prover that aims to achieve the highest levels of reliability and trustworthiness through careful design and implementation of its core components. In this paper, we concentrate on its treatment of concrete syntax, explaining how it manages to avoid problems suffered in other HOL systems related to the parsing and pretty printing of HOL types, terms and theorems, with the goal of achieving well-behaved parsing/printing and Pollack-consistency. Included are an explanation of how Hindley-Milner type inference is adapted to cater for variable-variable overloading, and how terms are minimally annotated with types for unambiguous printing.

1 Introduction

1.1 Overview

HOL Zero [16] is a basic theorem prover for the HOL logic [8]. It differs from other systems in the HOL family [7] primarily due to its emphasis on reliability and trustworthiness. One innovative area of its design is its concrete syntax for HOL types, term and theorems, and the associated parsers and pretty printers. These are not only important for usability, to enable the user to input and read expressions without fuss or confusion during interactive proof, but also for high assurance work, when it is important to know that what has been proved is what is supposed to have been proved.

In this paper we cover aspects of HOL Zero's treatment of concrete syntax, explaining how it manages to avoid classic pitfalls suffered in other HOL systems and achieve, or so we claim, two desirable qualities outlined by Wiedijk [15], namely well-behaved parsing/printing and Pollack-consistency.

In Section 2, we provide motivation for a thorough treatment of concrete syntax. In Section 3, we cover the concrete syntax problems suffered in other HOL systems. In Section 4, we explain the solutions provided in HOL Zero by its lexical syntax. In Section 5, we describe HOL Zero's solution for interpreting terms involving variable-variable overloading, through its variant of the Hindley-Milner type inference algorithm. In Section 6, we describe HOL Zero's solution for printing terms and theorems that would be ambiguous without type annotation, through its algorithm for minimal type annotation. In Section 7, we present our conclusions.

1.2 Concepts, Terminology and Notation

In [15], a theorem prover's term parser and printer are *well-behaved* if the result of parsing the output from printing any well-formed term is always the same as the original term. This can be thought of as ∀ `tm` • `parse(print(tm)) = tm`. A term parser is *input-complete* if every well-formed term can be parsed from concrete syntax. A term or theorem printer is *Pollack-consistent* if provably different terms or theorems can never be printed the same. For any printer, by *unambiguous printing* we mean that different internal representation can never be printed the same. Note that well-behaved parsing/printing implies input-completeness, unambiguous printing and Pollack-consistency for terms. Also note that none of these notions fully address the issue of *faithfulness*, where internal representation and concrete syntax correctly correspond. A printer that printed `false` as `true` and `true` as `false` might be Pollack-consistent but would not be faithful.

By *entity* we mean a HOL constant, variable, type constant or type variable. Note that the same entity can occur more than once in a given type or term. Two different entities are *overloaded* if they occur in the same scope and have the same name. A *syntax function* is an ML function dedicated to a particular syntactic category of HOL type or term, for constructing a type or term from components, destructing into components, or testing for the syntactic category. A *quotation* is concrete syntax for a HOL type or term that can be parser input or printer output. An *antiquotation* is an embedding of ML code within HOL concrete syntax (this is only supported by some HOL systems). A *symbolic* name is a non-empty string of symbol characters such as `!`, `+`, `|`, `#`, etc. An *alphanumeric* name is a non-empty string of letter, numeric, underscore or single-quote characters. An *irregular* name is a name that is neither symbolic nor alphanumeric. Note that the precise definitions of *symbolic* and *alphanumeric* vary between systems.

In Table 1 we summarise the concrete syntax constructs used in this paper.

Construct	HOL4, HOL Light and HOL Zero	Isabelle/HOL	ProofPower HOL
logical entailment	`P |- Q`	`Q [P]`	`P` ⊢ `Q`
universal quantification	`!v. P`	`ALL v. P`	∀ `v•` `P`
existential quantification	`?v. P`	`EX v. P`	∃ `v•` `P`
lambda abstraction	`\v. E`	`%v. P`	λ `v•` `E`
implication	`P ==> Q`	`P --> Q`	`P` ⇒ `Q`
disjunction	`P \/ Q`	`P | Q`	`P` ∨ `Q`
conjunction	`P /\ Q`	`P & Q`	`P` ∧ `Q`
equality	`x = y`	`x = y`	`x = y`
non-equality	`x <> y`	`x ~= y`	`x` ≠ `y`
less than or equals	`x <= y`	`x <= y`	`x` ⩽ `y`
addition	`x + y`	`x + y`	`x + y`

Table 1. Syntactic constructs used in our examples. Constructs are listed in increasing order of binding power, so for example `x + y <= z` is the same as `(x + y) <= z`, but note that equality binds less tightly than implication in HOL4.

2 The Need for Good Parsers and Printers

2.1 Proof Auditing

In recent years, HOL systems have been employed in large-scale high-assurance projects such as the verification of the seL4 operating system kernel [10], the verification of safety-critical avionics in the EuroFighter Typhoon [2], and the Flyspeck project formalising the Kepler Conjecture proof [6], a major result in mathematics. Projects of such importance should be independently audited to reduce the risk that they contain fundamental errors. As argued in [13], not only is it vital that the inference steps performed in a formal proof are correct, but also that the formal proof is proving what is intended to be proved. In a software verification project, there may be a large specification of required program behaviour and/or various formal statements of properties that the program must satisfy, and these should be reviewed. In a mathematics formalisation, the statement of the ultimate theorem being proved should be reviewed. The definitions of the constants used in any of these statements also need to be reviewed.

To be reviewed, these expressions need to be written in human-readable form. But how can the auditor be sure that the expressions seen in concrete syntax correctly correspond to their internal representation that gets manipulated in the formal proof? They must rely on the parsers and/or pretty printers to faithfully and unambiguously convert between the two representations. However, as with most theorem provers that support concrete syntax, the parsers and printers of HOL systems (other than HOL Zero) are known to suffer from problems such as input-incompleteness, ambiguous printing and Pollack-inconsistency, as pointed out in [15] and detailed further in Section 3.

As discussed in [1], the auditor can avoid the need to trust the system's parsers, as well as the need to review the project's proof scripts, by examining the system's state after the formal proof has been processed. This would normally require the full concrete syntax printers to be trusted.

Arguably, the auditor could also avoid the need to trust the full concrete syntax printers, either by using relatively-simple primitive syntax printers, or by using syntax functions to decompose expressions, or by directly viewing internal representation in ML. However, any of these measures would greatly reduce readability (for example, see how the small expression in Figure 1 balloons up), and for non-trivial content, the process of review would itself become error-prone. It would be far better if the auditor could trust the correctness of the printers for full concrete syntax, because this would greatly simplify the review process.

Many of the known problems with printing are obscure cases. However, we do not consider it satisfactory to dismiss them as too unlikely to occur in practice. For two reasons, they are more likely than might be imagined. Firstly, because industrial-sized formal proof projects inevitably involve extending the theorem prover with bespoke automated proof routines, problematic obscure syntax that would not be used in interactive proof could quite conceivably be generated by poorly written code, for example in generating a variable name. Secondly, because formal proofs can be outsourced, as was done in Flyspeck, we cannot

```
‘!x y. x > 1 /\ y > 1 ==> x * y > 1‘

‘((!):(num->bool)->bool)
   (\(x:num).
     ((!):(num->bool)->bool)
       (\(y:num).
         ((==>):bool->bool->bool)
           (((/\):bool->bool->bool)
             ((>):num->num->bool) (x:num) (NUMERAL (BIT1 _0)))
             ((>):num->num->bool) (y:num) (NUMERAL (BIT1 _0))))
           ((>):num->num->bool)
             ((*):num->num->num) (x:num) (y:num))
             (NUMERAL (BIT1 _0)))))‘
```

Fig. 1. Concrete and primitive syntax for the same expression in HOL Light.

discount the possibility of users maliciously exploiting flaws in a pretty printer in order to cheat the system and get paid for theorems they did not really prove.

Neither do we consider it realistic that the auditor can properly rule out the possibility of subtle printer exploits by reviewing the project's proof scripts. These can run to tens or even hundreds of thousands of lines of code, and unless they adhere to a language subset that rules out exploits, they are virtually impossible to review properly in reasonable time (e.g. see [1]).

Finally, the auditor not only needs to trust the pretty printers, but also needs to know *why* they can be trusted. Thus it is preferable if their implementation is simple or has a simple architectural argument for why it is fail-safe, or ideally a formal proof of correctness. Complex code implementing a complex architecture of interaction does not help in this regard.

2.2 Usability

Another concern, quite separate from the need to review results, is usability during interactive proof. Users naturally expect not to be distracted with using convoluted mechanisms to ensure their input gets parsed to the intended internal representation, or with worries about whether a printed expression gets wrongly printed, or whether it gets ambiguously printed and wrongly interpreted by the user. In addition, users reasonably expect to always be able to parse back in printed terms to result in the same internal term. However, there is no HOL system (other than HOL Zero) that meets these expectations in basic usage.

3 Classic Problems with HOL Parsers and Printers

In this section, we attempt to enumerate all the common problems that exist in the treatment of concrete syntax, and comment on how well they are addressed in each of the four main HOL systems: HOL4 [14], Isabelle/HOL [12], HOL

Light [9] and ProofPower HOL [3]. The problems relate to basic interaction with the system, in monochrome and using the system's standard character set (UTF-8 for HOL4, ASCII for Isabelle/HOL and HOL Light, extended ASCII for ProofPower HOL). We illustrate some problems with examples, providing code for reproducing simple but seemingly absurd theorem results (thus showing Pollack-inconsistency). The reader need not understand the code, but just the concrete syntax (see Table 1) of the resulting theorem, printed at the bottom.

Some systems have facilities that can be employed to help. Both HOL4 and Isabelle/HOL support coloured syntax highlighting in printed output when used with certain GUIs, distinguishing between free variables, bound variables, constants and keywords. However, this restricts choice of GUI, complicates the trusted code base and does not help the parser parse the output. HOL4, Isabelle/HOL and ProofPower HOL support antiquotation in parsed type and term quotations, allowing entities to be constructed outside of concrete syntax. However, such expressions are quite difficult to read, and the user must know how to use syntax functions for constructing entities. Also, HOL4 and Isabelle/HOL do not print using antiquotation, and so this facility does not help the auditor.

HOL4, Isabelle/HOL and ProofPower HOL each have a type annotation printing mode for their term and theorem printers, which can provide solutions but are unset by default. HOL4's (set by the `show_types` flag) annotates one occurrence of each variable entity, and each instance of a polymorphic constant unless it is a function with at least one argument supplied. Isabelle/HOL's (set by the `show_types` flag) annotates one occurrence of each variable entity but no constants. ProofPower HOL's (set by the `pp_show_HOL_types` flag) annotates every occurrence of each variable but no constants.

3.1 Entities with Irregular Names

In HOL abstract syntax, there are no restrictions on the form of the name that can be given to an entity.[3] In HOL4 and Isabelle/HOL, irregular names can only be parsed by using antiquotation, and are printed incorrectly and without delimitation. In HOL Light, irregular names cannot be parsed, and are printed incorrectly without delimitation. ProofPower HOL does cater for irregular entity names, by allowing names to be enclosed within `$""` delimiters, except that type variable names that do not begin with the single-quote character cannot be parsed, and entity names that begin with a double-quote character can only be parsed using antiquotation and are printed with antiquotation.

Example 1. The following theorem about natural numbers gets misleadingly printed in HOL Light due to the irregular variable name "`!y. x`".

```
# let x = mk_var ("x",‘:num‘) and y = mk_var ("y",‘:num‘) in
  let v = mk_var ("!y. x",‘:num‘) in
  let tm1 = list_mk_comb (‘(+)‘,[v;y]) in
  EXISTS (mk_exists (x, mk_eq (tm1,x)), tm1) (REFL tm1);;
val it : thm = |- ?x. !y. x + y = x
```

[3] Although in hol90, type variable names must start with a single-quote character.

3.2 Entities with Keyword Names

In HOL abstract syntax, there is nothing preventing an entity from having the same name as a keyword from a given system's concrete syntax. In HOL4, such entities are prefixed with `$` in their identifier for both parsing and printing, and can also be distinguished from the keyword by syntax highlighting. In Isabelle/HOL, such entities can only be parsed by using antiquotation, and in printed output can only be distinguished from keywords by using coloured syntax highlighting. In HOL Light, such entities cannot be parsed, and are printed incorrectly as though they were keywords. ProofPower HOL uses `$""` delimiters for entities overloaded with keywords, and does not suffer from any problems like those for irregular names because no keywords involve the single or double quote characters.

3.3 Variable-Constant Overloading

In HOL abstract syntax, constants cannot be overloaded with other constants, and type constants cannot be overloaded with other type constants.[4] However, variables may be overloaded with constants, and type variables overloaded with type constants. In HOL4 and Isabelle/HOL, such variables and type variables can only be parsed by using antiquotation, and in printed output can only be distinguished from constants and type constants by using coloured syntax highlighting. In HOL Light, such variables and type variables cannot be parsed, and are printed incorrectly as though they are the corresponding constants or type constants respectively. In ProofPower HOL, such variables and type variables can only be parsed by using antiquotation, and are printed using antiquotation.

3.4 Variable-Variable Overloading

HOL abstract syntax allows different variables with the same name but different types to occur in the same scope. This causes problems for the term quotation parser in any system that uses the basic algorithm for Hindley-Milner type inference, because this algorithm does not cater for variable-variable overloading (as explained in Section 5.1). HOL4, Isabelle/HOL, HOL Light and ProofPower HOL all use the Hindley-Milner algorithm, and so cannot parse terms with overloaded variables, even if antiquotation is used. Neither can these systems show distinction between overloaded variables in default printed output, although the type annotation printing modes of HOL4, Isabelle/HOL and ProofPower HOL do.

Example 2. The bound variable for the inner universal quantification in the following theorem proved in ProofPower HOL is a boolean `x`, rather than the natural number `x` used elsewhere in the theorem.

---

[4] We refer here to the "vanilla" version of the HOL language, implemented by all HOL systems except Isabelle/HOL and HOL Omega.

```
:) let val x1 = mk_var ("x",⌜:ℕ⌝) and x2 = mk_var ("x",⌜:BOOL⌝)
       and v = mk_var ("a",⌜:ℕ⌝)
       val tm = mk_eq (x1,v)
   in
     ∀_intro x1
       (∃_intro (mk_∃ (v, mk_⇒ (tm, mk_∀ (x2,tm))))
         (⇒_intro tm (∀_intro x2 (asm_rule tm))))
   end;
val it = ⊢ ∀ x• ∃ a• x = a ⇒ (∀ x• x = a): THM
```

3.5 Type Ambiguity in Printed Terms

Typically, the default mode for printing in HOL systems, including in HOL4, Isabelle/HOL, HOL Light and ProofPower HOL, is to not provide any type annotation when printing terms and theorems. This is problematic because an expression involving variables or polymorphic constants then gets printed ambiguously when there is more than one type-correct way of assigning types to the atoms in the expression. However, HOL4's type annotation printing mode will perform sufficient annotation to remove ambiguity, except potentially in the obscure circumstance of a polymorphic function constant application with types not fully resolved by the arguments supplied. Isabelle/HOL's and ProofPower's will also suffice, so long as no constant annotation is required.

Example 3. The following theorem in HOL4 is proved for the one-valued type `unit`, but by default is not printed with type annotation and so looks like it has been proved for any type.

```
> let val x = ‘‘x:unit‘‘ and y = ‘‘y:unit‘‘
  in
    GENL [x,y]
      (DISCH ‘‘(x:unit)<>y‘‘
        (TRANS (SPEC x oneTheory.one)
               (SYM (SPEC y oneTheory.one)) ))
  end;
# val it = |- !x y. x <> y ==> (x = y): thm
```

3.6 Juxtaposition of Lexical Tokens in Printed Expressions

Concrete syntax is generally printed with spacing characters separating lexical tokens, to ensure that the boundaries between tokens are obvious. In some circumstances, however, it tends to be more readable to be more concise and print without spacing, for example between parentheses and the expression they enclose. This is safe providing that boundaries between tokens are still clear, so that juxtaposed tokens without intervening space cannot be confused for a single token, or vice versa. HOL Light, however, does not ensure boundaries still exist when it prints without spacing in the printing of bindings, pairs and lists. We do not know of any problem cases in HOL4, Isabelle/HOL or ProofPower HOL.

Example 4. In the following theorem in HOL Light, no space is printed in the second conjunct between the binder `!`, binding variable `#` and "`.`" keyword, making it indistinguishable from the variable "`!#.`" in the first conjunct.

```
# let x = mk_var ("x",‘:num‘) and v1 = mk_var ("#",‘:num‘)
  and v2 = mk_var ("!#.",‘:num->num‘) in
  let tm1 = mk_comb (v2,v1) in
  let tm2 = mk_exists (x, list_mk_comb (‘(<=)‘,[tm1;x])) in
  CONJ (EXISTS (tm2,tm1) (SPEC tm1 LE_REFL))
       (MESON [ARITH_RULE ‘!x. ~ (SUC x <= x)‘]
              ‘~ ?x. ! # . # <= x‘);;
val it : thm = |- (?x. !#. # <= x) /\ ~(?x. !#. # <= x)
```

4 Lexical Solutions

Some of the problems highlighted in Section 3 are tackled in HOL Zero purely through lexical considerations. This section covers those solutions.

4.1 Identifier Delimiters

For the problems of irregular entity names (see 3.1) and entities with the same name as keywords (see 3.2), HOL Zero allows an entity's identifier to wrap double-quote delimiters (`""`) around the entity name. Any entity name can be wrapped with these delimiters, but an irregular or keyword name must be wrapped in order to get properly parsed. An entity identifier gets printed with the delimiters if and only if the entity name is irregular or a keyword.

This is essentially the same solution as used in ProofPower HOL, but there are no corner cases that cause problems. The backslash character (\) functions as an escape in the identifier, where double-quote and backslash are preceded by a backslash, and unprintable ASCII characters (such as tab and line feed) and back-quote (used in HOL Zero to delimit HOL type and term quotations) are denoted by their 3-digit decimal ASCII code preceded by a backslash.

Example 5. The irregular variable name in Example 1 is parsed and printed in HOL Zero using double-quote delimiters.

```
# ‘?x. "!y. x" + y = x‘;;
- : term = ‘?x. "!y. x" + y = x‘
```

4.2 Variable Marking

For the problem of variables being overloaded with constants (see 3.3), HOL Zero allows a variable identifier to indicate that it denotes a variable, by preceding the variable's name with a percent character (`%`). This marking must immediately precede the variable name, without intervening space. Similarly, a type variable identifier can be marked as such by preceding the name with a single-quote

character (’). If a variable is overloaded with a constant, or a type variable with a type constant, then it necessarily gets marked as such in printed output.

These marking characters must occur outside any double-quote delimiters used in the entity identifier (see 4.1). Note that a single-quote mark at the start of a type variable identifier cannot get confused with a single-quote at the start of a type variable name, because alphanumeric names in HOL Zero cannot start with a single-quote and so the name would be classed as irregular and get enclosed within double-quote delimiters. Similarly, a percent mark at the start of a variable identifier cannot get confused with a percent at the start of a variable name, because in HOL Zero percent is not classed as a symbolic character.

Example 6. The variable `true` is overloaded with the constant, and so needs a variable mark when parsed and printed.

```
# ‘%true + 1 < 5‘;;
- : term = ‘%true + 1 < 5‘
```

Example 7. The type variable `nat` is overloaded with the type constant for natural numbers, and so needs a type-variable mark when parsed and printed. Note that, by default, all type variables are printed with the type variable mark.

```
# ‘!(x:’nat) (y:A). ?z. z = (x,y)‘;;
- : term = ‘!(x:’nat) (y:’A). ?z. z = (x,y)‘
```

4.3 Spacing between Tokens

For the problem of juxtaposed lexical tokens (see 3.6), HOL Zero ensures that in any situation where tokens get printed without any intervening space, a check is performed to make sure that the two tokens are compatible, i.e. that there is a lexical boundary between them when printed together, and if this check fails then spacing is inserted.

Example 8. In HOL Zero, the juxtaposed symbolic tokens in the head of the universal quantification in the second conjunct in Example 4 are printed with spacing to make clear they are separate tokens.

```
# ‘(?x. !#. # <= x) /\ ~ (?x. ! # . # <= x)‘;;
- : term = ‘(?x. !#. # <= x) /\ ~ (?x. ! # . # <= x)‘
```

5 Type Inference for Variable-Variable Overloading

In order to cater for variable-variable overloading in the interpretation of HOL concrete syntax (see 3.4), it is not sufficient to use the classic Hindley-Milner type inference algorithm [11] that is used in other HOL systems. This algorithm was originally designed for parsing the programming language ML, which supports parametric polymorphism, needed for HOL’s polymorphic constants, but not ad-hoc polymorphism, needed for HOL’s variable-variable overloading. In this

section, we explain how the Hindley-Milner algorithm is adapted in HOL Zero to cater for both.

Note that the discussion takes place at the level of HOL primitive syntax, because this is the level of representation used in the abstract syntax tree (AST) upon which type inference is carried out, and thus *binding* refers to lambda abstraction.

5.1 Hindley-Milner Type Inference

The Hindley-Milner type inference algorithm first involves assigning provisional types to all constant and variable atoms in the AST of the expression being parsed. Each instance of a non-polymorphic constant is simply assigned the constant's type, each instance of a polymorphic constant is assigned the constant's generic type but with unique meta type variables replacing any placeholder types, and each variable is given a unique meta type variable which is assigned to each occurrence of the variable (according to the scoping rules explained below). These meta type variables are then resolved bottom-up when function applications are encountered in ascending the AST, where the domain type of a function is unified with the type of its argument.

The notion of a variable identifier's syntactic scope in Hindley-Milner is straightforward. All variable atoms with a given name refer to the same entity throughout the AST of the expression up to the first common binding for a variable with that name (if such a binding exists), or otherwise the top level of the expression (when the variable is free). The types of all these variable atoms must unify, or otherwise the expression is ill-typed. Above the binding in the AST, the variable ceases to be in scope and any variable atoms with the given name refer to a different entity. This notion of scope does not allow for variable-variable overloading.

Example 9. Consider the following expression in HOL Zero concrete syntax:

```
x \/ (!(x:nat). x = 5 \/ true = x)
```

Using purely primitive syntax, this is written as follows (where `$` is used to strip an operator of its fixity):[5]

```
$\/ x ($! (\(x:nat). $\/ ($= x 5) ($= true x)))
```

We uniquely number variables that are different according to the Hindley-Milner scoping rules, as well as each instance of the same polymorphic constant, so that they can be treated distinctly.

```
$\/ x1 ($! (\(x2:nat). $\/ ($=1 x2 5) ($=2 true x2)))
```

[5] Note that, in HOL Zero, numerals are not atoms of primitive syntax, but for illustrative purposes it suffices to treat `5` as an atom in this example.

Provisional types get assigned as follows, with τ_n denoting the nth generated meta type variable:

`\/`	$\texttt{bool} \to \texttt{bool} \to \texttt{bool}$	$\texttt{x}_1$	τ_4
`!`	$(\tau_1 \to \texttt{bool}) \to \texttt{bool}$	$\texttt{x}_2$	τ_5
$=_1$	$\tau_2 \to \tau_2 \to \texttt{bool}$		
$=_2$	$\tau_3 \to \tau_3 \to \texttt{bool}$		
`5`	`nat`		
`true`	`bool`		

Type inference on the AST then resolves meta type variables bottom-up, starting as follows on the branches of the inner `\/`:

$\tau_2 = \tau_5 = \texttt{nat}$
$\tau_3 = \texttt{bool} = \tau_5$

At this point, there is a conflict: τ_5 cannot be both `nat` and `bool`. Thus the expression is ill-typed according to traditional Hindley-Milner rules.

5.2 A New Notion of Syntactic Scope for Variables

Clearly we need a new notion of a variable's syntactic scope in order to cater for variable-variable overloading. As in Hindley-Milner type inference, we want this notion to be simple and intuitive for users, as well as relatively light in the amount of type annotation required. We do not want this notion, for example, to require every occurrence of every variable to be type-annotated, because this would be too inconvenient for the user.

In HOL Zero, the scoping rules are exactly the same as for Hindley-Milner for the basic case when there is no variable overloading – all variable atoms with a given name denote the same entity up to the binding for the variable name, or the top level of the expression if there is no such binding, so long as the types of these variable atoms all unify. The difference lies in the case when the variable atoms for a given variable name don't all unify. Unlike Hindley-Milner, HOL Zero allows for an extra case, for when the variable atoms each have fully-resolved types (i.e. not involving meta type variables) purely from local type resolution that can take place within the binding alone, i.e. not considering contextual type information from outside the binding in the AST. In this special case, according to the HOL Zero scoping rules there is a variable in scope for each of the fully resolved types, and only the variable corresponding to the binding variable ceases to be in scope above the binding in the AST.

Thus, in a nutshell, the HOL Zero scoping rules say that at a binding, all variables with the same name as the binding variable must unify or otherwise have types that can be fully resolved locally. Although more complicated than in Hindley-Milner, this new notion of variable scope is still relatively straightforward to understand and to use. All expressions that parse under Hindley-Milner type inference will still parse under HOL Zero type inference, to result in the

same internal term. However, any expression involving variable-variable overloading can also be parsed, so long as there is sufficient type-annotation. Note that one consequence of the rules is that the scope of a variable can only be determined at the type inference stage of parsing, rather than earlier, but we do not see this as a problem.

5.3 Adjustments to the Hindley-Milner Algorithm

To support the HOL Zero notion of variable scope, the Hindley-Milner algorithm needs a few adjustments. The first is that, in the initial assignment of provisional types throughout the AST, each variable atom, rather than variable entity, is given its own unique meta type variable. This is because the variable referred to by a given variable atom is only determined as type inference unfolds, and at the start of type inference each variable atom could potentially refer to a different variable.

The second adjustment is to the variable environment that is inevitably passed around in the algorithm's implementation, that provides the partially resolved type for a given variable name. This environment needs to be adjusted to carry a list of types, rather than a single type, with one type for each potentially distinct variable.

The third adjustment is another restriction on how the algorithm is implemented. In the Hindley-Milner algorithm, the variable environment for the components of a compound expression may build on the variable environment for a sibling component. However, this is not allowed in general in the HOL Zero type inference implementation, because the scoping rules must consider whether variables' types are fully resolved purely locally.

The fourth adjustment is to type inference at a binding, to cater for the extra case, of having variables overloaded with the binding variable, instead of simply raising an error.

Example 10. In the expression from Example 9, we uniquely number each variable atom, because at this stage they are potentially all different according to the HOL Zero scoping rules.

$$\texttt{\$\textbackslash/ x}_1\ \texttt{(\$! (\textbackslash(x}_2\texttt{:nat). \$\textbackslash/ (\$=}_1\ \texttt{x}_3\ \texttt{5) (\$=}_2\ \texttt{true x}_4\texttt{)))}$$

Provisional types get assigned as follows:

`\/`	$\texttt{bool} \to \texttt{bool} \to \texttt{bool}$	$\texttt{x}_1$	τ_4
`!`	$(\tau_1 \to \texttt{bool}) \to \texttt{bool}$	$\texttt{x}_2$	τ_5
$\texttt{=}_1$	$\tau_2 \to \tau_2 \to \texttt{bool}$	$\texttt{x}_3$	τ_6
$\texttt{=}_2$	$\tau_3 \to \tau_3 \to \texttt{bool}$	$\texttt{x}_4$	τ_7
`5`	`nat`		
`true`	`bool`		

Meta type variables are resolved bottom-up. On reaching the binding, the following type information has been inferred:

$\tau_2 = \tau_6 = \texttt{nat}$
$\tau_3 = \texttt{bool} = \tau_7$
$\tau_5 = \texttt{nat}$

This means that types for the variables local to the binding are thus:

$\texttt{x}_2$	nat
$\texttt{x}_3$	nat
$\texttt{x}_4$	bool

According to HOL Zero scoping rules, given that `x` is the name of the binding variable and the types of the different `x` atoms do not all unify, all of the local `x` atoms at this point must have fully resolved types, which they do. The atom for the binding variable is $\texttt{x}_2$ and has the same resolved type as $\texttt{x}_3$, and so these refer to the same variable, and are both removed from scope, leaving just $\texttt{x}_4$ in the local variable environment. Type inference can now proceed upwards from the binding.

$\tau_1 = \tau_5$
$\tau_4 = \texttt{bool}$

Having reached the top level of the expression, there are no conflicts, so the expression has passed type inference. The atoms are thus resolved as follows:

\/	bool → bool → bool	$\texttt{x}_1$	bool
!	(nat → bool) → bool	$\texttt{x}_2$	nat
$=_1$	nat → nat → bool	$\texttt{x}_3$	nat
$=_2$	bool → bool → bool	$\texttt{x}_4$	bool
5	nat		
true	bool		

Thus there are two variables with name `x`: one with type `nat`, which is a bound variable, and one with type `bool`, which is free.

6 Minimal Unambiguous Type Annotation

Unless the types of its variables and polymorphic constants can be fully resolved from context, a printed term or theorem needs to be type-annotated in order to avoid type ambiguity (see 3.5). However, type annotating every variable and constant atom significantly reduces readability of output. HOL4's type annotation printing mode offers a better solution by annotating each variable entity only once in a given term, and only annotating instances of polymorphic constants. However, this level of annotation can still be excessive for large expressions.

In this section, we describe HOL Zero's algorithm for minimal type annotation when printing, so that there is just enough to avoid ambiguity and the output stays readable. This algorithm works for HOL Zero's notion of variable scoping that caters for variable-variable overloading (see 5.2).

Example 11. No type annotation is necessary when types can be inferred, as in the example in Figure 1.

```
‘!x y. x > 1 /\ y > 1 ==> x * y > 1‘
```

Example 12. It is not necessary to type-annotate variable `a` from Example 2, because its type can be inferred:

```
‘!x. ?a. x = a ==> (!(x:bool). (x:nat) = a)‘
```

The basic outline of the algorithm is first to make a copy of the term to be printed but with each atom replaced with a provisional type, in exactly the same way as is done for HOL Zero type inference, with unique meta type variables for each variable atom and for each placeholder type in a polymorphic constant's generic type. The mapping from meta type variables to actual types is then worked out by matching this term copy with the original term. A subset of the atoms in the term copy is then picked for annotation, in such a way as to be minimal but sufficient to remove ambiguity from the printed term. The chosen atoms are then annotated with their types according to the meta type variable mapping to actual types. Before being printed, the resulting annotated term is passed to the type analyser to check that type resolution results in an internal term that is identical to the original term. This final check means that the relatively-complicated minimal type annotation algorithm does not need to be trusted, at the expense of having to trust the relatively-simple type analyser.

The only part of this that needs any further description is the process for picking atoms for annotation from the term copy. This first involves calculating four lists of atoms from the term copy, corresponding to the overloaded variables, the free variables, the bound variables and the polymorphic constants of the expression. The atoms in these four lists are the candidates for type annotation. Type inference is then performed on the term copy (with its meta type variables in place of actual types), to result in a partial instantiation list for the meta type variables based purely on what can be deduced from the term copy without any type annotations added.

This partial instantiation list is then used as the basis for removing atoms from the four atom lists, to remove those atoms that have types that can be fully resolved without any type annotation. The remaining atoms are selected down into a list to be type annotated. This is done by first ordering the atoms according to priority, with overloaded variables having highest priority, then bound variables, then free variables and polymorphic constants having lowest priority. The list is then scanned to remove atoms coming later in the list that will not need annotating because their types are inferred by annotating atoms earlier on in the list. The remaining atoms are then returned for annotation.

7 Conclusions

In this paper, we have enumerated the classic problems that have plagued default usage of HOL system parsers and pretty printers over the years. Some HOL systems fare better than others, and offer a patchwork of solutions such as antiquotation and coloured syntax highlighting which can be employed to solve some problems. However, these complicate the trusted code base, complicate output and restrict how the auditor can work, and in any case do not completely solve all problems. Our view is that simpler and more comprehensive solutions are possible, that work in monochrome ASCII and are easier to trust. We find it surprising that such solutions are not already prevalent for a logic that is so established as HOL.

We have shown how HOL Zero manages to overcome all the classic problems we list and fit our criteria for simplicity, by employing three main solutions. The first is a better lexical syntax regime that supports delimiting and marking for otherwise problematic entity names, and that ensures sufficient spacing between lexical tokens. The second is an adapted form of Hindley-Milner type inference that enables term quotations with variable-variable overloading to be represented in concrete syntax. The third is an algorithm for performing minimal type annotation on the atoms of a term, so that it can be printed in a way that is both unambiguous and about as readable as can be expected.

Together, these solutions enable parsing and printing of concrete syntax to be more complete and less ambiguous than in the other HOL systems. This boosts HOL Zero's credentials as a trustworthy system for auditing formal proofs. We believe HOL Zero does not suffer from any incompleteness or ambiguity in its parsers or printers, and printed output can always be parsed back in to give the same internal representation. This would make HOL Zero's parsers and printers well-behaved and Pollack-consistent. As far as we know, this would be a first amongst not only HOL systems, but also various other theorem proving systems that support concrete syntax, such as Coq and Mizar.

It would be interesting to establish for certain whether our claims of correctness in HOL Zero's parsing and printing are true. HOL Zero has a bounty [16] that, as well as for logical unsoundness, gets paid out for "printer unsoundness", i.e. if the pretty printer can be made to produce output that is ambiguous or not faithful to internal representation. Six printer-related problems have been reported, but none since August 2011. Trustworthiness could be further bolstered by adding a check for well-behaved parsing/printing to the printers.

Ideally there would be a formal proof about the correctness of HOL Zero's parsers and printers, which would cover questions of faithfulness as well as well-behaved parsing/printing and Pollack-consistency. The challenges in achieving this would presumably be quite different from those in formally verifying [5] the parser and printer of the Milawa theorem prover [4], which has no concrete syntax as such and so effectively boiled down to verifying the underlying Lisp implementation's parser and pretty printer for s-expressions.

Our solutions could conceivably be implemented in other HOL systems, to improve their usability as well as trustworthiness. It should be possible to in-

corporate type inference for overloaded variables and minimal type annotation without any major backwards compatibility issues, because the former just expands the set of term quotations that can be parsed, and the latter could be an optional printing mode. Incorporating delimiters for problematic entity names and marking for variables would require adjustment to the lexical syntax as well as the parser, and may have knock-on effects, but these are unlikely to be severe.

References

1. M. Adams. *Proof Auditing Formalised Mathematics.* In Volume 9(1) of Journal of Formalized Reasoning, pages 3-32. 2016.
2. M. Adams & P. Clayton. *ClawZ: Cost-Effective Formal Verification for Control Systems.* In Proceedings of the 7th International Conference on Formal Methods and Software Engineering, Volume 3785 of Lecture Notes in Computer Science, pages 465-479. Springer, 2005.
3. R. Arthan & R. Jones. *Z in HOL in ProofPower.* In Issue 2005-1 of the British Computer Society Specialist Group Newsletter on Formal Aspects of Computing Science, pages 39-54. 2005.
4. J. Davis. *A Self-Verifying Theorem Prover.* PhD Thesis, University of Texas at Austin, 2009.
5. J. Davis & M. Myreen. *The Reflective Milawa Theorem Prover Is Sound.* Volume 55(2) of Journal of Automated Reasoning, pages 117-183. Springer, 2015.
6. T. Hales et al. *A Formal Proof of the Kepler Conjecture.* arXiv:1501.02155v1 [math.MG]. arxiv.org, 2015.
7. M. Gordon. *From LCF to HOL: A Short History.* In Proof, Language and Interaction, pages 169-186. MIT Press, 2000.
8. M. Gordon & T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
9. J. Harrison. *HOL Light: An Overview.* In Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, Volume 5674 of Lecture Notes in Computer Science, pages 60-66. Springer, 2009.
10. G. Klein et al. *seL4: Formal Verification of an OS Kernel.* In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, pages 207-220. ACM, 2009.
11. R. Milner. *A Theory of Type Polymorphism in Programming.* In Volume 17 of Journal of Computer and System Sciences, pages 348-375. Elsevier, 1978.
12. T. Nipkow, L. Paulson & M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Volume 2283 of Lecture Notes in Computer Science. Springer, 2002.
13. R. Pollack. *How to Believe a Machine-Checked Proof.* In Twenty-Five Years of Constructive Type Theory, chapter 11. Oxford University Press, 1998.
14. K. Slind & M. Norrish. *A Brief Overview of HOL4.* In Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics, Volume 5170 of Lecture Notes in Computer Science, pages 28-32. Springer, 2008.
15. F. Wiedijk. *Pollack-Inconsistency.* In Volume 285 of Electronic Notes in Theoretical Computer Science, pages 85-100. Elsevier Science, 2012.
16. HOL Zero homepage: `http://www.proof-technologies.com/holzero/`.