

# PHOLI: Partial Higher-Order Logic with Interfaces

Anonymous author

Anonymous affiliation

## Abstract

---

We give an overview of a new logic for interactive proof checking, which we call PHOLI.

The intended application is the formalization of mathematical proofs that occur in theoretical computer science. PHOLI is based on a 3-valued logic with strict semantics. Types and preconditions of partial functions can be expressed in this logic. The first-order variant does not need an additional type system, but PHOLI still has a minimal structural type system to avoid logical paradoxes caused by higher-order. Higher-order logic is unavoidable because many notions in computer science are inductive in nature. PHOLI uses overloading techniques from modern programming languages to resolve ambiguous names. We will present terms, the structural type system, the type resolving algorithm, a simple natural deduction calculus which specifies PHOLI completely, and a few formalizations. PHOLI has no global data structures, which implies that every defined data structure is automatically polymorphic.

2012 ACM Subject Classification Author: Please fill in 1 or more \ccsdesc macro

Keywords and phrases interactive proving, 3-valued logic

Digital Object Identifier 10.4230/LIPIcs...

## 1 Introduction

We give an overview of the PHOLI calculus. We start by formulating the goals of PHOLI. Some of these goals may be very long term, but it is still good to keep in mind what the goals are. Firstly, verification should obviously ensure correctness. Secondly, formalizing a group of proofs should improve the understanding of these proofs, in the same way that implementation in a high-level language improves understanding of an algorithm. Most of this understanding will come from observing the dependencies and observing that many subparts of the proofs do not depend on everything that was proven or assumed around them. This makes it possible to make these subproofs more generic, and reuse them later. A good proof checker will invite to see proofs in unexpected ways, in the same way that a good programming language invites to look at algorithms in surprising ways. Ideally, a system based on PHOLI should be usable in teaching. In our experience with teaching formal language theory, in particular pumping lemmas ([10]), students have problems understanding counterfactuals, and understanding the difference between existential and universal quantification (instances can be chosen or must be generic), in combination with the De Morgan laws.

The calculus must be well-defined, not too complicated, and its implementation should also not be too complicated, or at least have a reasonably small trusted core.

A future implementation should support automatic proof search, at least for type checking. Because in PHOLI, types are expressed inside the logic, there is no border on how complex they can be. We nevertheless think that most type conditions can be expressed by Horn clauses that derive the types of a term from the types of its subterms, so that most will be automatically checkable.

PHOLI takes the 3-valued logic of [3] as starting point. This logic is best understood in terms of strong unsatisfiability. A sequent  $A_1, \dots, A_n \vdash$  is *strongly unsatisfiable* if it is unsatisfiable, and in addition, the first formula whose interpretation is not **t**, is always interpreted as **f**. Strong unsatisfiability is able to capture the requirement that formulas and functions must be declared before they are used. If one assumes that  $\# A$  means



© Anonymous author(s);

licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

46  $I(A) \in \{\mathbf{f}, \mathbf{t}\}$ , then  $\#A, \neg(A \rightarrow (A \vee B)) \vdash$  is not strongly unsatisfiable, because one can  
 47 interpret  $I(A) = \mathbf{t}$ ,  $I(B) = \mathbf{e}$ . On the other hand  $\#A, \#B, \neg(A \rightarrow (A \vee B)) \vdash$  is strongly  
 48 unsatisfiable. We give the basics of strict 3-valued logic in subsection 2.1.

49 In order to make this logic usable, it has to be generalized to higher-order. This is  
 50 in principle straightforward, but one needs to add a type system. In order to separate  
 51 functions from simple objects, (or in general, keep values of different logical categories apart),  
 52 a rudimentary type system is needed. We call the resulting types *structural types*. Once  
 53 we have structural types, it is convenient to allow the user to introduce more types by  
 54 defining **structs**. This makes it possible to define things like finite automata, which can  
 55 be represented by a tuple  $(Q, \Sigma, \delta, Q_s, Q_a)$ . Although technically, one could formalize tuples  
 56 inside 3-valued logic by adding proper axioms (for example set theory), in practice, one  
 57 does not want to do that. In practice, there never is any need to mix automata with other  
 58 mathematical objects, like e.g. numbers. Once we have the possibility to define **structs**, it is  
 59 also convenient to use them to define signatures. We do this in Section 4 when we introduce  
 60 natural numbers. An additional advantage of adding a structural type system is that names  
 61 can be overloaded for different types. This is used in Sections 3.1 and 3.2, where strictness  
 62 and functionality predicates are being defined for predicates of different arities. In addition,  
 63 one can reuse the same predicate name for different **structs**, for example calling a predicate  
 64 'well-formed'. In our view, this addition greatly improves user friendliness.

## 65 2 Overview of PHOLI

66 We start by giving an informal introduction of PHOLI. Our goal is to find a logic in which  
 67 it is convenient to express mathematical concepts, as they occur in theoretical computer  
 68 science. The problem with a project like this is that it is difficult to distinguish between  
 69 problems arising from lack of familiarity with the logic, problems arising from lack of tools,  
 70 and problems that are intrinsic to the logic itself. This is necessarily a slow process.

### 71 2.1 Strict Three-Valued Logic

72 PHOLI is based on a variant of three-valued logic developed in [3], which was called *partial*  
 73 *classical logic* (PCL). The main goal of PCL was to be able to transfer type information into  
 74 the formulas, so that one can write  $\forall x \text{Nat}(x) \rightarrow x \geq x$  instead of  $\forall x:\text{Nat} x \geq x$ .

75 The advantage of this approach is that it becomes possible to reason about types inside  
 76 the logic, which makes it possible to add preconditions to functions, add subtypes, add a  
 77 single element to a type, etc. The obvious disadvantage is that type checking is not automatic  
 78 any more.

79 Since types now have become a part of the logic, instead of something that is checked  
 80 before proving starts, one needs a way of expressing ill-typedness in the semantics. For this  
 81 purpose, a third truth value  $\mathbf{e}$  was introduced, in addition to the standard truth values  $\mathbf{f}$ ,  $\mathbf{t}$ .  
 82 The meaning of  $\mathbf{e}$  is *error*. This is different from Kleene's logic (see [6], or [4]), where the  
 83 third truth-value is interpreted as *unknown*. The main problem of interpreting ill-typed  
 84 formulas as unknown, is that well-typedness cannot be used as precondition. If one assumes  
 85 that an ill-typed formula is true, one implicitly assumes that it is well-typed, hence it acts  
 86 as a type declaration. This invalidates the main goal of a type system, which is to reject  
 87 ill-typed formulas. In order to express well-typedness, PCL uses a logical operator  $\#A$ , with

88 truth table

89  $\# :$

t
f
t

90 The interpretation of  $\#A$  equals **t** if the interpretation of  $A$  is a proper boolean. Using  
 91  $\#$ , one can express that predicate  $\leq$  is well-defined, whenever its arguments are natural  
 92 numbers:  $\forall x, y \text{ Nat}(x) \wedge \text{Nat}(y) \rightarrow \# x \leq y$ . Implication  $\rightarrow$  is separated into two implication  
 93 operators: Lazy implication is written as  $[A] B$ , and strict implication is written as  $A \rightarrow B$ .  
 94 In lazy implication,  $A$  acts as a declaration whose scope extends into  $B$ . In strict implication,  
 95 both  $A$  and  $B$  must be independently well-typed. The formula  $[\#A] A \vee \neg A$  is valid in PCL,  
 96 while  $\#A \rightarrow (A \vee \neg A)$  is ill-typed. The truth tables are as follows (first argument down,  
 97 second argument to the right):

98  $[ ] :$

t	t	t
e	e	e
f	e	t

$\rightarrow :$

t	e	t
e	e	e
f	e	t

99 The difference is that, in case of  $[A] B$ , whenever  $I(A) = \mathbf{f}$ , the value of  $I([A] B)$  will always  
 100 be **t**, while in case of  $A \rightarrow B$ , the value of  $I(B)$  still needs to be a proper truth value. This  
 101 guarantees that  $I(\forall x [\text{Nat}(x)] x \geq x)$  is always interpreted inside  $\{\mathbf{f}, \mathbf{t}\}$ , while at the same  
 102 time,  $I(\forall x \text{Nat}(x) \rightarrow x \geq x)$  can be **e**.

103 In the similar way as  $\rightarrow$  was separated into two operators, we separate  $\wedge$  into two  
 104 operators  $\langle A \rangle B$  and  $A \wedge B$ . The intuitive meanings are the same. In  $\langle A \rangle B$ , the left hand  
 105 side  $A$  can act as a declaration whose scope extends into  $B$ , while in  $A \wedge B$ , both  $A$  and  $B$   
 106 must be well-typed independently. The respective truth tables are:

107  $\langle \rangle :$

f	f	f
e	e	e
f	e	t

$\wedge :$

f	e	f
e	e	e
f	e	t

108 In the same way as with  $\rightarrow$  and  $\forall$ , the interpretation  $I(\exists x \langle \text{Nat}(x) \rangle x \geq x)$  is always in  
 109  $\{\mathbf{f}, \mathbf{t}\}$ , while  $I(\exists x \text{nat}(x) \wedge x \geq x)$  can be **e**, when  $I(\text{Nat}(x)) = \mathbf{f}$ .

110 For completeness, we list the remaining propositional operators:

111  $\vee :$

f	e	t
e	e	e
t	e	t

$\rightarrow :$

t	e	t
e	e	e
f	e	t

$\leftrightarrow :$

t	e	f
e	e	e
f	e	t

## 112 2.2 Structural Types

113 In PCL, types were expressed as predicates inside the logic. This is unproblematic as long as  
 114 one uses only first-order logic, but it is impossible in higher-order logic, because it will not  
 115 prevent the paradoxes. As a consequence, some kind of built-in type system is needed, which  
 116 is able to separate logical levels.

117 In addition to separating logical levels, it is useful to allow the user to define types.  
 118 Defined types are useful for representing mathematical objects that are naturally represented  
 119 as tuples, like partially ordered sets, rewrite systems, automata, or grammars. Although one  
 120 could in principle do this inside the logic by axiomatizing tuples, in practice, one never mixes  
 121 tuples representing different mathematical structures. Expressing types by predicates is only

122 useful when a single value can have more than one type at the same time. This also applies  
 123 to preconditions of functions because preconditions can be viewed as subtypes. A value that  
 124 satisfies a precondition can be viewed as having two types: Its base type, and the implicit  
 125 type attached to the precondition.

126 ► **Definition 1.** Structural types are recursively defined as follows:

- 127 ■  $\mathbf{O}$  is a structural type, representing the type of objects.
- 128 ■  $\mathbf{T}$  is a structural type, representing the type of truth-values (including  $\mathbf{e}$ .)
- 129 ■ If  $v$  is an identifier, then  $v$  is a structural type.
- 130 ■ If  $\lambda$  is an index, then  $\lambda$  is a structural type.
- 131 ■ If  $U_1, \dots, U_n$  are structural types, and  $V$  is a structural type, then  $V(U_1, \dots, U_n)$  is a  
 132 structural type.

133 PHOLI distinguishes between  $V(U_1, \dots, U_n, W_1, \dots, W_m)$  and  $V(W_1, \dots, W_m)(U_1, \dots, U_n)$ .  
 134 In the first case, the function requires  $n+m$  arguments of structural types  $W_1, \dots, W_m, U_1, \dots, U_n$ .  
 135 In the second case, the function additionally can be called with arguments  $U_1, \dots, U_n$ . PHOLI  
 136 uses *controlled Currying*. Currying means that there is a single, binary application operator,  
 137 and that functions are applied one argument at a time. This requires that all incomplete  
 138 applications, including the function itself, must be independently typeable. We want to  
 139 allow inexact names that can be overloaded for different argument types, like all modern  
 140 programming languages as Java and  $C^{++}$  have them. This is only possible if there are no  
 141 incomplete applications. Since it is sometimes convenient to use an incomplete application,  
 142 we allow incomplete applications at designated points that the user can choose.

143 PHOLI allows inexact names, which during type checking are replaced by exact overloads.  
 144 Inexact names are represented by identifiers, while exact overloads are represented either by  
 145 a De Bruijn index (if local), or by an index into the environment (if global). Inexact names  
 146 are also allowed in structural types. During type checking they are replaced by the index of  
 147 a struct definition in the environment. A struct definition has the following form:

148 ► **Definition 2.** If  $v_1, \dots, v_n$  ( $n \geq 0$ ) is a sequence of mutually distinct identifiers,  $T_1, \dots, T_n$   
 149 is a sequence of structural types, then  $v := \mathbf{struct}(v_1:T_1, \dots, v_n:T_n)$  is a **struct** definition.

150 Struct definitions must be not circular. This is different from programming languages like  
 151  $C/C^{++}$ , where one can write `struct list { val double; list* next; }`; We assume  
 152 that type definitions are intentional. This means that two declared **struct** types with  
 153 identical fields having the same names will be treated as distinct.

154 ► **Example 3.** In order to define natural numbers, one needs a zero, and a successor function.  
 155 They can be defined together as `struct(0:O, succ:O(O))`. We could call this struct 'Nat'.  
 156 We could define when a predicate is under  $s.0$  and  $s.succ$  as follows:

157 
$$\text{isclosed} := \lambda n:\text{Nat} \lambda P:\mathbf{T}(\mathbf{O}) P(s.0) \wedge \forall x:\mathbf{O} P(x) \rightarrow P(s.succ(x)).$$

158 The type is  $\mathbf{T}(\mathbf{T}(\mathbf{O}))(\text{Nat})$ . Restriction of Currying makes it possible to define `isclosed`  
 159 predicates for different structs without risk of confusion. If we would merge the two  $\lambda$ -s,  
 160 the type would become  $\mathbf{T}(\text{Nat}, \mathbf{T}(\mathbf{O}))$ . In that case, one could not use `isclosed(s, P)` or  
 161 `s.isclosed(P)` without an additional argument. For example, an object is *generated by*  $s$  (of  
 162 type  $\text{Nat}$ ) if all closed predicates are true on it:

163 
$$\text{gen} := \lambda s:\text{Nat} \lambda x:\mathbf{O} \forall P:\mathbf{T}(\mathbf{O}) [\text{strict}(P)] s.\text{isclosed}(P) \rightarrow P(x).$$

164 The type is  $\mathbf{T}(\mathbf{O})(\text{Nat})$ . In the definition, `isclosed` is applied on  $P$  incompletely. `strict` is  
 165 defined as `strict := \lambda P:\mathbf{T}(\mathbf{O}) \forall x:\mathbf{O} \#P(x)`, which ensures that that applying  $P$  never results  
 166 in  $\mathbf{e}$ , and the meaning of `[]` was defined in Section 2.1. It is a kind of implication that requires  
 167 `s.isclosed  $\rightarrow$  P(x)` to be well-typed only when `strict(P)` is true.

168 **2.3 Terms**

169 ► **Definition 4.** *The set of terms is recursively defined as follows:*

- 170 ■ *If  $\mu$  is an index, then  $\mu$  is a term.*
- 171 ■ *If  $v$  is a local variable, then  $v$  is a term.*
- 172 ■ *If  $v$  is an identifier, then  $v$  is a term.*
- 173 ■  *$\perp$  and  $\top$  are terms.*
- 174 ■ *If  $A$  is a term, then  $\neg A$ , and  $\# A$  are also terms.*
- 175 ■ *If  $A$  and  $B$  are terms, then  $A \wedge B$ ,  $A \vee B$ ,  $A \rightarrow B$ ,  $A \leftrightarrow B$ ,  $[A]B$ , and  $\langle A \rangle B$  are also terms.*
- 176 ■ *If  $t$  and  $u$  are terms, then  $t \approx u$  is also a term.*
- 177 ■ *If  $A$  is a term and  $v_1, \dots, v_n$  are local variables,  $T_1, \dots, T_n$  are structural types,  $\forall v_1:T_1 \dots v_p:T_p A$  and  $\exists v_1:T_1 \dots v_p:T_p A$  are terms.*
- 178 ■ *If  $f$  and  $t_1, \dots, t_n$  ( $n > 0$ ) are terms, then  $f \cdot (t_1, \dots, t_n)$  is also a term. The intended meaning is the application of  $f$  on  $t_1, \dots, t_n$ . If  $f$  is an identifier, local variable, or index, we write  $f(t_1, \dots, t_n)$  instead of  $f \cdot (t_1, \dots, t_n)$ .*
- 179 ■ *If  $v$  is a local variable,  $t$  is term,  $T$  is a structural type, and  $u$  is a single term, then **let**  $v := t:T$  **in**  $u$  is also a term.*
- 180 ■ *If  $t$  is a term,  $v_1, \dots, v_p$  with  $p > 0$  is a sequence of variables with structural types  $T_1, \dots, T_p$ , then  $\lambda v_1:T_1 \dots v_p:T_p t$  is a term.*

187 Instead of standard function notation  $f(t_1, t_2, \dots, t_n)$ , we also allow member notation  
 188  $t_1.f(t_2, \dots, t_n)$ . This is sometimes more natural, especially when  $t_1$  has a struct type.  
 189 Excepts for syntax, there is no difference.

190 We call all declarations, definitions, assumptions and theorems together the *environment*.

191 ► **Definition 5.** *The environment is a finite sequence of items of one of the six forms below:*

- 192 ■ *A struct definition of form  $v := \mathbf{struct}(\dots)$ , see Definition 2.*
- 193 ■ *A declaration of form  $v:T$ .*
- 194 ■ *A definition of form  $v := t:T$ .*
- 195 ■ *An assumption or theorem of form  $v(T_1, \dots, T_n) := \mathbf{asm}(F, \pi)$ , or  $v(T_1, \dots, T_n) := \mathbf{thm}(F, \pi)$ .  $T_1, \dots, T_n$  is a sequence of structural types, and  $F$  must be a term (formula).*
- 196 ■ *A field definition of form  $v := \mathbf{fld}(\mu, i)$ , where  $\mu$  is an index and  $i$  is the position of the field in its struct.*
- 197 ■ *A constructor definition of form  $v := \mathbf{constr}(\mu)$ , where  $\mu$  is an index.*

200 *All cases introduce an identifier. It is possible that the same identifier is introduced more than once.*

202 During type checking, identifiers are replaced by indices into the environment, which makes  
 203 them completely unambiguous.

204 Names of theorems and assumptions occur only at designated places in proofs. Hence  
 205 their names cannot conflict with the other names. In case of  $\mathbf{asm}(F, \pi)$ ,  $\pi$  must be a proof  
 206 of  $\#F$ . In case of  $\mathbf{thm}(F, \pi)$ ,  $\pi$  must be a proof of  $F$ . The  $T_1, \dots, T_n$  are structural types  
 207 that further identify the theorem, to support overload resolution. They are automatically  
 208 extracted from  $F$ , if  $F$  starts with a universal quantifier.

209 Field and constructor definitions are automatically obtained from the struct definition. If  
 210 the struct definition has form  $v := \mathbf{struct}(v_1:T_1, \dots, v_n:T_n)$  and its index in the environment  
 211 is  $\mu$ , then the field definitions have form  $v := \mathbf{fld}(\mu, i)$  for  $1 \leq i \leq n$ , and the constructor  
 212 definition has form  $v := \mathbf{constr}(\mu)$ . No types are stored, since type checking algorithm will  
 213 infer their types from the struct declaration.

214 In a declaration or definition, where  $T$  is a functional type, we will allow to write the  
 215 arguments of the type as arguments of the defined variable. This results in more readable  
 216 definitions/declarations. So, instead of

$$217 \quad v := (\lambda \bar{v}_1:\bar{T}_1 \cdots \lambda \bar{v}_n:\bar{T}_n t):T(\bar{T}_n) \cdots (\bar{T}_1)$$

218 the user can write

$$219 \quad v(\bar{v}_1:\bar{T}_1) \cdots (\bar{v}_n:\bar{T}_n) := t:T,$$

220 which is a lot more readable.

221 ► **Example 6.** Looking back at Example 3, the definition of `isclosed` can be written as

$$222 \quad \text{isclosed}(n:\text{Nat})(P:\mathbf{T}(\mathbf{O})) := P(s.0) \wedge \forall x:\mathbf{O} P(x) \rightarrow P(s.\text{succ}(x)),$$

223 and the definition of `gen` can be written as

$$224 \quad \text{gen}(s:\text{Nat})(P:\mathbf{T}(\mathbf{O})) := [\text{strict}(P)] s.\text{isclosed}(P) \rightarrow P(x).$$

225 For a given identifier  $v$  and index  $i$ , there can be at most one assumption or definition. If an  
 226 identifier  $v$  has distinct definitions or declarations  $v^i, v^j$  with types  $T$  and  $T'$ , it must be the  
 227 case that  $\Pi(T) \neq \Pi(T')$ . Otherwise, there is no situation in which it would be possible to  
 228 choose between  $v^i$  and  $v^j$ .

## 229 2.4 Structural Type Checking

230 Now that we defined the belief state, we can define type checking for structural types. The  
 231 algorithm is standard, with the only change that PHOLI supports overloading, so that  
 232 functions with the same name can be defined for different application patterns. It is not  
 233 difficult to obtain, and it increases user friendliness. For the types that the user defines inside  
 234 the logic, one should not try to allow overloading. There is no way to automatically determine  
 235 such types, and it would be risky, because in many cases it will be hard to determine which  
 236 overload has been selected, which would make it hard to determine the exact meaning of a  
 237 term.

238 ► **Definition 7.** A context  $\Gamma$  is a finite sequence of assumptions of form  $v_1:T_1, \dots, v_n:T_n$   
 239 with  $n \geq 0$ . The variables  $v_i$  must be pairwise distinct.

240 We think that there should be no overload resolution for local variables, so that variables in  
 241 contexts do not have indices. In reality, the  $v_i$  are represented by De Bruijn indices, so that  
 242 there is no need to store them in the context.

243 ► **Definition 8.** Function `apply`( $T, i, (T_1, \dots, T_n)$ ) tries to apply structural type  $T$  on  
 244 structural types  $T_1, \dots, T_n$ . If it succeeds, it returns the resulting type.

245 ■ If  $i = n + 1$ , then `apply` succeeds and the result is  $T$ .

246 ■ Otherwise, if  $T$  has form  $U(U_1, \dots, U_m)$  and  $i + m > n + 1$ , then `apply` fails.

247 ■ If  $T$  has form  $U(U_1, \dots, U_m)$ ,  $i + m \leq n + 1$ , and for  $1 \leq j \leq m$ , we have  $U_j = T_{i+j-1}$ ,  
 248 then `apply`( $T, i, (T_1, \dots, T_n)$ ) = `apply`( $U, i + m, T_1, \dots, T_n$ ).

249 Function `apply`( $E_\mu, (T_1, \dots, T_n)$ ) tries to apply  $E_\mu$  on structural types  $T_1, \dots, T_n$ . It first  
 250 decides what type  $E_\mu$  requires, and after that, it calls `apply` with the required type. If it  
 251 succeeds, it returns the resulting type.

- 252 ■ If  $E_\mu$  is a declaration  $v:T$  or a definition  $v := t:T$ , then  $\mathbf{apply}(E_\mu, (T_1, \dots, T_n)) =$   
 253  $\mathbf{apply}(T, 1, (T_1, \dots, T_n))$ .
- 254 ■ If  $E_\mu$  is a field definition of form  $\mathbf{fld}(\mu', i)$ , then let  $\mathbf{struct}(v_1:T_1, \dots, v_n:T_n) = E_{\mu'}$  be the  
 255 definition of its struct. We set  $\mathbf{apply}(E_\mu, (T_1, \dots, T_n)) = \mathbf{apply}(V_i(\mu'), 1, (T_1, \dots, T_n))$ .  
 256 Here we are using the index  $\mu'$  as a structural type. We treat a field as a function from  
 257 the struct to the type of the field.
- 258 ■ If  $E_\mu$  has form  $v := \mathbf{constr}(\mu')$ , then let  $\mathbf{struct}(v_1:T_1, \dots, v_n:T_n) = E_{\mu'}$  be the definition  
 259 of its struct.  $\mathbf{apply}(E_\mu, (T_1, \dots, T_n))$  equals  $\mathbf{apply}(\mu'(V_1, \dots, V_n), 1, (T_1, \dots, T_n))$ . We  
 260 are using  $\mu'$  as a structural type again. The constructor is a function from the types of  
 261 the fields to the struct.
- 262 In the remaining cases,  $\mathbf{apply}(E_\mu, (T_1, \dots, T_n))$  fails.

263 ► **Definition 9.** Let  $t$  be a term, let  $\Gamma$  be a local context, let  $E$  be the environment. We  
 264 define partial function  $\mathbf{stype}(\Gamma, t)$  that returns a pair  $(t', Z)$ , where  $t'$  is a term, and  $Z$  is  
 265 the structural type of  $t'$ . Term  $t'$  is obtained from  $t$  by replacing inexact identifier names by  
 266 local variables or by indices into the environment. We are using letter  $Z$  for the structural  
 267 type instead of  $T$ , because  $T$  is too similar to  $\mathbf{T}$ .

268 We omit  $E$  from the parameter list, because it never changes during typechecking. The result  
 269 of  $\mathbf{stype}(\Gamma, t)$  is recursively defined following the structure of  $t$  :

- 270 ■ If  $t$  is an identifier, then  $\mathbf{stype}(\Gamma, t) = \mathbf{stype}(\Gamma, t \cdot ())$ .
- 271 ■ If  $t$  equals  $\top$  or  $\perp$ , then  $\mathbf{stype}(\Gamma, t)$  equals  $(t, \mathbf{T})$ .
- 272 ■ If  $t$  has form  $\neg t_1$  or  $\#t_1$ , then let  $(t'_1, Z_1) = \mathbf{stype}(\Gamma, t_1)$ . If  $Z_1 = \mathbf{T}$ , then  $\mathbf{stype}(\Gamma, t)$   
 273 equals  $(\neg t'_1, \mathbf{T})$ , (or  $(\#t'_1, \mathbf{T})$ ). Otherwise it fails.
- 274 ■ If  $t$  has form  $t_1 \star t_2$ , where  $\star \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ , then let  $(t'_1, Z_1) = \mathbf{stype}(\Gamma, t_1)$ , and let  
 275  $(t'_2, Z_2) = \mathbf{stype}(\Gamma, t_2)$ . If  $Z_1 = Z_2 = \mathbf{T}$ , then  $\mathbf{stype}(\Gamma, t)$  equals  $(t'_1 \star t'_2, \mathbf{T})$ . Otherwise,  
 276 it fails.
- 277 ■ If  $t$  has form  $[t_1]t_2$ , then let  $(t'_1, Z_1) = \mathbf{stype}(\Gamma, t_1)$ , and let  $(t'_2, Z_2) = \mathbf{stype}(\Gamma, t_2)$ . If  
 278  $Z_1 = Z_2 = \mathbf{T}$ , then  $\mathbf{stype}(\Gamma, t)$  equals  $([t'_1]t'_2, \mathbf{T})$ . Otherwise, it fails. The case for  $\langle t_1 \rangle t_2$   
 279 is analogous.
- 280 ■ If  $t$  has form  $t_1 \approx t_2$ , let  $(t'_1, Z_1) = \mathbf{stype}(\Gamma, t_1)$  and let  $(t'_2, Z_2) = \mathbf{stype}(\Gamma, t_2)$ . If  
 281  $Z_1 = Z_2 = \mathbf{O}$ , then  $\mathbf{stype}(\Gamma, t)$  returns  $(t'_1 \approx t'_2, \mathbf{O})$ . Otherwise, it fails.
- 282 ■ If  $t$  has form  $Qv_1:V_1 \cdots v_p:V_p t_1$ , then let  $(t'_1, Z_1) = \mathbf{stype}(\Gamma \circ (v_1:V_1 \cdots v_p:V_p), t_1)$ . If  
 283  $Z_1 = \mathbf{T}$ , then  $\mathbf{stype}(\Gamma, t) = (Qv_1:V_1 \cdots v_p:V_p t'_1, \mathbf{T})$ . Otherwise, it fails.
- 284 ■ If  $t$  has form  $\lambda v_1:V_1 \cdots v_p:V_p t_1$ , then let  $(t'_1, Z_1) = \mathbf{stype}(\Gamma \circ (v_1:V_1 \cdots v_p:V_p), t_1)$ . After  
 285 that,  $\mathbf{stype}(\Gamma, t) = (\lambda v_1:V_1 \cdots v_p:V_p t'_1, Z_1(V_1, \dots, V_p))$ .
- 286 ■ If  $t$  has form  $v \cdot (t_1, \dots, t_n)$  with  $v$  an identifier, then first get  $(t'_i, Z_i) = \mathbf{stype}(\Gamma, t_i)$ , for  
 287  $1 \leq i \leq n$ . If  $v$  occurs as  $v:Z$  in  $\Gamma$ , then  $\mathbf{stype}(\Gamma, t)$  equals  $\mathbf{apply}(Z, 1, (Z_1, \dots, Z_n))$ .  
 288 Otherwise, let  $M$  be the set of indices in  $E$  that introduce something called  $v$ . Let  
 289  $M' = \{(\mu, Z) \mid \mu \in M \text{ and } \mathbf{apply}(E_\mu, (Z_1, \dots, Z_n)) \text{ succeeds and returns } Z\}$ . If  $M'$   
 290 contains zero or more than one element, then  $\mathbf{stype}(\Gamma, t)$  has failed. Otherwise let  $(\mu, Z)$   
 291 be the unique element of  $M'$ . Then  $\mathbf{stype}(\Gamma, t)$  equals  $(\mu \cdot (t'_1, \dots, t'_n), Z)$ .
- 292 ■ If  $t$  has form  $t_0 \cdot (t_1, \dots, t_n)$  with  $t_0$  not an identifier, then let  $(t'_0, Z_0) = \mathbf{stype}(\Gamma, t_0)$ ,  
 293 and let  $(t'_i, Z_i) = \mathbf{stype}(\Gamma, t_i)$ , for  $1 \leq i \leq n$ . Let  $Z = \mathbf{apply}(Z_0, 1, (Z_1, \dots, Z_n))$ . Then  
 294  $\mathbf{stype}(\Gamma, t)$  equals  $(t'_0 \cdot (t'_1, \dots, t'_n), Z)$ .
- 295 ■ If  $t$  has form  $\mathbf{let } v := t_1:T \mathbf{ in } t_2$ , then let  $(t'_1, Z_1) = \mathbf{stype}(\Gamma, t_1)$ . If  $Z_1$  not equal to  $T$ ,  
 296 then fail. Otherwise, let  $(t'_2, Z_2) = \mathbf{stype}(\Gamma \circ (v:T), t_2)$ . After that,  $\mathbf{stype}(\Gamma, t)$  equals  
 297  $(\mathbf{let } v := t'_1:T \mathbf{ in } t'_2, Z_2)$ .

298 Note that there is no special case for field selection  $t.f$ , since the parser replaces it by  $f(t)$ .

299 This concludes the discussion terms of PHOLI, and its type system. In order to prove  
 300 theorems, one could use the sequent calculus  $\text{Seq}_{PCL}$  in [3], but an earlier prototype has  
 301 shown that this calculus (and probably every other sequent calculus) is too user unfriendly.  
 302 We refrain in this paper from defining a concrete calculus for proof editing, but it will  
 303 probably be resolution based.

## 304 2.5 An Axiomatic Calculus

305 We present a simple axiomatic calculus that is suitable for checking PHOLI proofs. The  
 306 calculus is simple enough to serve as a trusted code base for checking proofs. We do not  
 307 expect that a simpler calculus is possible. At this moment, we are not sure if the calculus is  
 308 suitable for interactive proof checking. We call the calculus  $\text{Ax}_{rmPHL}$ . The calculus is based  
 309 on the semantics that we informally described in Section 2.1. Because PHOLI is higher-order,  
 310 we need to be able to simplify certain functional expressions that may be introduced by a  
 311 higher-order substitution.

312 ► **Definition 10.** We write  $t_1 \Rightarrow t_2$  when  $t_1$  can be reduced into  $t_2$ , possibly with a subscript  
 313 indicating the reduction used. The following reductions are possible:

314 **beta:** If  $m \geq n$ , then

$$315 \quad (\lambda v_1:T_1 \cdots v_n:T_n t) \cdot (t_1, \dots, t_m) \Rightarrow_{\beta} t[v_1 := t_1, \dots, t_n := v_n] \cdot (t_{n+1}, \dots, t_m).$$

316  $\beta$ -reduction takes  $n$  arguments from an application term, and leaves the remaining  
 317 arguments in the application. If  $m = n$ , the resulting empty application can be removed  
 318 by associativity-reduction.

319 **def:** If  $E_{\mu} = (v := t:T)$ , then  $\mu \Rightarrow_{\delta} t$ . If  $\mu$  is the index of a definition in the environment, it  
 320 can be expanded. We are using the fact that indices can be used as terms.

321 **projection:** If  $E_{\gamma} = \mathbf{constr}(\mu)$ , and  $E_{\varphi} = \mathbf{fld}(\mu, i)$ , then  $\gamma$  is a constructor  $\mu$ , and  $\varphi$  is  
 322 a field of the same struct  $\mu$ . In that case, one has

$$323 \quad \varphi \cdot (\gamma \cdot (t_1, \dots, t_n), u_1, \dots, u_m) \Rightarrow_{\pi} t_i \cdot (u_1, \dots, u_m).$$

324 It is possible that  $m = 0$ .

325 **assoc:** The following reductions are needed because we allow incomplete function application:

$$326 \quad \begin{array}{ll} t \cdot () & \Rightarrow_a t \\ (t \cdot (t_1, \dots, t_n)) \cdot (u_1, \dots, u_m) & \Rightarrow_a t \cdot (t_1, \dots, t_n, u_1, \dots, u_m) \end{array}$$

327 In order to understand projection, note that by associativity reduction  $\varphi \cdot (\gamma \cdot (t_1, \dots, t_n), u_1, \dots, u_m)$   
 328 is equivalent to  $(\varphi \cdot (\gamma \cdot (t_1, \dots, t_n))) \cdot (u_1, \dots, u_m)$ . In principle, **let**  $v := t:T$  **in**  $u$  can be  
 329 replaced by  $(\lambda v:T u) \cdot t$ , but that is not what **let** is intended for. **let** is used when the user  
 330 wants a gradual, controlled replacement of  $v$  by  $t$ .

331 In an axiomatic method, only formulas are derived, which implies that technically, no  
 332 further structure is needed. Nevertheless, it is still convenient to introduce sequents, because  
 333 they can be viewed as a restricted form of formulas, which separate into a passive and an  
 334 active part and restrict the possible forms of the passive part.

335 ► **Definition 11.** A sequent is an object of  $\Gamma_1, \dots, \Gamma_n \vdash A$  ( $n \geq 0$ ), where each  $\Gamma_i$  is either a  
 336 declaration of form  $v:T$ , a definition  $v := t:T$ , or a formula.

337 The intuitive meaning of a valid sequent is: If in some interpretation, all variables in  $\Gamma$  are  
 338 introduced, in such a way that the definitions are respected, and all formulas are true, then  
 339  $A$  is true in this interpretation.

340 We want to view sequents as a restricted form of formulas, but unfortunately, we lack an  
 341 operator for this. Consider for example, the sequent  $p:\mathbf{T}, p \vdash p$ . PCL has no operator that  
 342 can express this.

343 ► **Definition 12.** We define a new operator  $A \Longrightarrow B$  with the following semantics:  $A \Longrightarrow B$   
 344 is interpreted as  $\mathbf{t}$  if either  $A$  is  $\mathbf{e}$  or  $\mathbf{f}$ , or  $B$  is  $\mathbf{t}$ . In the remaining cases its interpretation is  
 345  $\mathbf{f}$ .

346 Operator  $\Longrightarrow$  can be used only in proofs, it cannot occur in assumptions or theorems in the  
 347 environment. Note that  $A \Longrightarrow B$  can be translated into  $[\#A][A]\langle\#B\rangle B$ . This is a bit long,  
 348 so we prefer to use  $\Longrightarrow$ . At this point, sequents can be translated into formulas:

349 ► **Definition 13.** Let  $\Gamma \vdash A$  be a sequent. The meaning of  $\Gamma \vdash A$  is a formula obtained as  
 350 follows: If  $\|\Gamma\| = 0$ , then the meaning of  $\Gamma \vdash A$  equals  $A$ . If  $\|\Gamma\| > 0$ , write  $\Gamma = \Gamma_1, \Gamma_2, \dots, \Gamma_n$ ,  
 351 and let  $F$  be the meaning of  $\Gamma_2, \dots, \Gamma_n \vdash A$ . If  $\Gamma_1$  has form  $v:V$ , then the meaning of  $\Gamma \vdash A$   
 352 is  $\forall v:V F$ . If  $\Gamma_1$  has form  $v := t:T$ , then the meaning of  $\Gamma \vdash A$  equals **let**  $v := t:T$  **in**  $F$ . If  
 353  $\Gamma_1$  is a formula  $A$ , then the meaning of  $\Gamma \vdash A$  is  $A \Longrightarrow F$ .

354 We only consider sequents that are well-typed, i.e. whose meaning will be accepted by  
 355 **stype**( $\cdot, \cdot$ ) returning  $\mathbf{T}$ . This implies that local variables introduced in a sequent are treated  
 356 as local variables. Calculus  $\text{Ax}_{\text{PHL}}$  is an axiomatic calculus. We now define what the axioms  
 357 are:

358 ► **Definition 14.** Let  $F$  be a closed formula, containing only **(1)** second order universal  
 359 quantification of form  $\forall v:\mathbf{T} A$ , and **(2)** the propositional operators  $\perp, \top, \neg, \#, \wedge, \vee, \rightarrow, \leftrightarrow$   
 360  $, [ \ ] , \langle \ \rangle$ , as well as  $\Longrightarrow$ . We call  $F$  an axiom if  $F$  is  $\mathbf{t}$  in every interpretation.

361 It is easy to design an algorithm that checks if a formula  $F$  is an axiom. Since all quantifications  
 362 are propositional, and there are only three truthvalues, it is straightforward to iterate through  
 363 them.

364 ► **Definition 15.** Let  $I$  be a finite mapping of local variables to truth values  $\mathbf{f}, \mathbf{e}, \mathbf{t}$ . We define  
 365 a function **eval**( $I, F$ ) that evaluates a formula (containing only propositional operators and  
 366 propositional quantification) in  $I$  as follows:

- 367 ■ If  $A = \perp$ , then **eval**( $I, A$ ) =  $\mathbf{f}$ . If  $t = \top$ , then **eval**( $I, t$ ) =  $\mathbf{t}$ .
- 368 ■ If  $A$  is a local variable  $v$ , then **eval**( $I, A$ ) =  $I(v)$ .
- 369 ■ If  $A$  has form  $\#A_1$  or  $\neg A_1$ , then let  $b = \mathbf{eval}(I, A_1)$ . After that, use the truthables in  
 370 Section 2.1 to obtain the value of **eval**( $I, A$ ).
- 371 ■ If  $A$  has form  $A_1 \wedge A_2, A_1 \vee A_2, A_1 \rightarrow A_2$ , or  $A_1 \leftrightarrow A_2$ , then let  $b_1 = \mathbf{eval}(I, A_1)$ , and  
 372 let  $b_2 = \mathbf{eval}(I, A_2)$ . Use the truthables in Section 2.1 to obtain the value of **eval**( $I, A$ ).
- 373 ■ If  $A$  has form  $A_1 \Longrightarrow A_2$ , then let  $b_1 = \mathbf{eval}(I, A_1)$ , and let  $b_2 = \mathbf{eval}(I, A_2)$ . Use  
 374 Definition 12 to obtain the value of **eval**( $I, A$ ).
- 375 ■ If  $A$  has form  $\forall v:\mathbf{T} A_1$ , then let  $B = \{ \mathbf{eval}(I \cup \{v, b\}, A_1) \mid b \in \{\mathbf{f}, \mathbf{e}, \mathbf{t}\} \}$ . If  $\mathbf{e} \in B$ , then  
 376 **eval**( $I, A_1$ ) =  $\mathbf{e}$ . Otherwise, if  $\mathbf{f} \in B$ , **eval**( $I, A_1$ ) =  $\mathbf{f}$ . In the remaining case,  $B = \{\mathbf{t}\}$ ,  
 377 and we set **eval**( $I, A_1$ ) =  $\mathbf{t}$ .
- 378  $F$  is a tautology if **eval**( $\emptyset, F$ ) =  $\mathbf{t}$ .

379 Unfortunately, function **eval**( $I, F$ ) has exponential complexity, and there is little hope for  
 380 a better algorithm since the property of being a tautology is PSPACE complete. On the  
 381 positive side, algorithm **eval**( $I, F$ ) runs in polynomial space.

382 ► **Definition 16.** Before we define  $\text{Ax}_{\text{PHL}}$ , we introduce three more reduction rules that are  
 383 specific to  $\text{Ax}_{\text{PHL}}$ :

## XX:10 PHOLI: Partial Higher-Order Logic with Interfaces

384 ■ The following reduction essentially turns  $\exists$  into a defined operator:

$$385 \quad \exists v_1:T_1 \cdots v_p:T_p A \Rightarrow_{\exists} \neg \forall v_1:T_1 \cdots v_p:T_p \neg A.$$

386 ■ The following reduction permutes  $\#$  with  $\forall$ :

$$387 \quad \# \forall v_1:T_1 \cdots v_p:T_p A \Rightarrow_{\#} \forall v_1:T_1 \cdots v_p:T_p \# A.$$

388 ■ The following reduction represents the fact that any two terms are equality comparable:

$$389 \quad \#(t_1 \approx t_2) \Rightarrow_{\approx} \top.$$

390 ► **Definition 17. axiom:** If  $A$  is an axiom, as defined in Definition 14, or  $A \in \Gamma$ , then

$$391 \quad \overline{\Gamma \vdash A}$$

392 **import:** If either  $\mathbf{thm}(A, \pi)$  or  $\mathbf{asm}(A, \pi)$  occurs in  $E$ , then

$$393 \quad \overline{\Gamma \vdash A}$$

**red:**

$$394 \quad \frac{\Gamma \vdash A}{\Gamma \vdash A'}$$

395 If  $A'$  and  $A$  can be merged by applying reductions of Definition 10 in either direction,  
396 possibly applying them on subterms. If there are definitions  $v := t:T$  in  $\Gamma$ , these can be  
397 used as reductions too.

$\implies$  **elim/intro:**

$$398 \quad \frac{\Gamma \vdash A \quad \Gamma \vdash A \implies B}{\Gamma \vdash B} \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \implies B}$$

$\forall$  **elim/intro**

$$399 \quad \frac{\Gamma \vdash \forall v_1:T_1 \cdots v_p:T_p A}{\Gamma \vdash A[v_1 := t_1, \dots, v_p := t_p]} \quad \frac{\Gamma, v_1:T_1 \cdots v_p:T_p \vdash A}{\Gamma \vdash \forall v_1:T_1 \cdots v_p:T_p A}$$

400 **let elim/intro:** If no free variable of  $t$  is bound in  $A[\ ]$ , then

$$401 \quad \frac{\Gamma \vdash A[\mathbf{let} v := t:T \mathbf{in} u]}{\Gamma, v := t:T \vdash A[u]} \quad \frac{\Gamma, v := t:T \vdash A[u]}{\Gamma \vdash A[\mathbf{let} v := t:T \mathbf{in} u]}$$

402 **def:** If  $v$  does not occur in  $A$ , then

$$403 \quad \frac{\Gamma, v := t:T \vdash A}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A}{\Gamma, v := t:T \vdash A}$$

404 Note that if  $v$  occurs in  $A$ , the definition can be used as a reduction.

**eqrepl:**

$$405 \quad \frac{\Gamma \vdash t_1 \approx t_2 \quad \Gamma \vdash A[t_1]}{\Gamma \vdash A[t_2]}$$

### 3 Example Developments

In the rest of this paper, we give examples of developments with PHOLI. We start with a couple of fundamental definitions, predicates that specify whether a property is well-behaved, i.e. never returns **e**. We call this property *strictness*. We try to translate the informal notion of well-typedness into strictness as much as possible, and handle as much as possible inside the logic. Because PHOLI is a higher-order logic, not all informal well-typedness can be translated into strictness, so that PHOLI still has a rudimentary type system based on structural types. This rarely causes problems, but it does cause problems in automata theory, when one wants to perform the subset construction. Subsets of states are of a higher logical order than states themselves, but need to be mapped to single states again to obtain a valid automaton.

In the next section, we define a few predicates that specify the behaviour of functors. In Section 3.3 we explain how to introduce functions. Function introduction is problematic. It is a fundamental operation that is not easily expressible as a deduction rule. We solve the problem partially by defining a struct containing a function introduction operator. Unfortunately, due to the way that PHOLI is organized, one has to assume this operator everywhere where a function is introduced, which becomes unpleasant very quickly. The argument in favour of doing this, would be the fact that one could replace it by another function introduction operator, but we are not using choice functions, so there is only one way in which functions can be introduced. The only freedom that one has, is what value should be returned when the underlying relation is not functional. In Section 4 we give a rudimentary introduction of natural numbers. It shows how PHOLI is intended to be used. Due to the fact that it has no global objects, everything that is introduced, is automatically polymorphic. This also applies to natural numbers, which automatically become polymorphic over the zero, and the successor function. While this is unusual, it might have advantages too, because one can pick an arbitrary constant, and unary function, and as soon as they satisfy the Peano axioms, they become natural numbers.

#### 3.1 Strictness Predicates

A predicate is strict if it never results in **e**. We use the following predicates for expressing strictness:

$$\begin{aligned} \mathbf{strict}(P: \mathbf{T}) &:= \#P \\ \mathbf{strict}(P: \mathbf{T}(\mathbf{O})) &:= \forall x_1: \mathbf{O} \#P(x_1) \\ \mathbf{strict}(P: \mathbf{T}(\mathbf{O}, \mathbf{O})) &:= \forall x_1, x_2: \mathbf{O} \#P(x_1, x_2) \\ \mathbf{strict}(P: \mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O})) &:= \forall x_1, x_2, x_3: \mathbf{O} \#P(x_1, x_2, x_3) \end{aligned}$$

Here we see PHOLI's possibility of overloading defined/declared identifiers in action. The type of the argument makes sure that overload resolution picks the right version **strict**.

Similarly, it is convenient to state that some predicate is well-behaved (does not return **e**) everywhere, where some other predicate is defined:

$$\begin{aligned} \mathbf{stricton}(P, Q: \mathbf{T}(\mathbf{O})) &:= \forall x_1: \mathbf{O} P(x_1) \rightarrow \#Q(x_1) \\ \mathbf{stricton}(P, Q: \mathbf{T}(\mathbf{O}, \mathbf{O})) &:= \forall x_1, x_2: \mathbf{O} P(x_1, x_2) \rightarrow \#Q(x_1, x_2) \\ \mathbf{stricton}(P, Q: \mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O})) &:= \forall x_1, x_2, x_3: \mathbf{O} P(x_1, x_2, x_3) \rightarrow \#Q(x_1, x_2, x_3) \end{aligned}$$

$$\begin{aligned} \mathbf{prod}(P_1, P_2: \mathbf{T}(\mathbf{O}))(x_1, x_2: \mathbf{O}) &:= P_1(x_1) \wedge P_2(x_2) \\ \mathbf{prod}(P_1, P_2, P_3: \mathbf{T}(\mathbf{O}))(x_1, x_2, x_3: \mathbf{O}) &:= P_1(x_1) \wedge P_2(x_2) \wedge P_3(x_3) \end{aligned}$$

443 For example, if one wants to specify that a predicate  $\text{Nat}:\mathbf{T}(\mathbf{O})$  is always defined, one can  
 444 state  $\text{strict}(\text{Nat})$ . If one wants to state that  $\text{leq}$  (less or equal than) is defined whenever  
 445 both of its arguments are natural numbers, one can write  $\text{stricton}(\text{prod}(\text{nat}, \text{nat}), \text{leq})$ .  
 446 Expanding the definitions results in

$$447 \quad \forall x_1, x_2: \mathbf{O} \text{ Nat}(x) \wedge \text{Nat}(y) \rightarrow \# \text{leq}(x, y).$$

### 448 3.2 Specifying the Behaviour of Functions

449 If one wants to state that  $\text{succ}:\mathbf{T}(\mathbf{O})$  is a function from  $\text{Nat}$  to  $\text{Nat}$ , one can write  $\forall x: \mathbf{O} \text{ Nat}(x) \rightarrow$   
 450  $\text{Nat}(\text{succ}(x))$ . Since such formulas are tedious to write and hard to read, we introduce  
 451 definitions:

$$\begin{aligned} & \text{fromto}(P:\mathbf{T}(\mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O}))(f:\mathbf{O}(\mathbf{O})) := \\ & \quad \forall x_1: \mathbf{O} P(x_1) \rightarrow Q(f(x_1)) \\ 452 \quad & \text{fromto}(P:\mathbf{T}(\mathbf{O}, \mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}))(f:\mathbf{O}(\mathbf{O}, \mathbf{O})) := \\ & \quad \forall x_1, x_2: \mathbf{O} P(x_1, x_2) \rightarrow Q(f(x_1, x_2)) \\ & \text{fromto}(P:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}, \mathbf{O}))(f:\mathbf{O}(\mathbf{O}, \mathbf{O}, \mathbf{O})) := \\ & \quad \forall x_1, x_2, x_3: \mathbf{O} P(x_1, x_2, x_3) \rightarrow Q(f(x_1, x_2, x_3)) \end{aligned}$$

453 Now one can write  $\text{func}(\text{Nat}, \text{Nat}, \text{succ})$ , or  $\text{func}(\text{prod}(\text{Nat}, \text{Nat}), \text{Nat}, \text{sum})$ . If one wants  
 454 express that  $\text{minus}(x, y)$  is defined only when  $\text{leq}(y, x)$ , one has to write

$$455 \quad \forall x, y: \mathbf{O} (\{\text{Nat}(x) \wedge \text{Nat}(y)\} \wedge \text{leq}(y, x)) \rightarrow \text{Nat}(\text{minus}(x, y)).$$

456 In order to make it possible to insert preconditions, we define:

$$\begin{aligned} & \text{with}(P_1, P_2:\mathbf{T}(\mathbf{O}))(x_1:\mathbf{O}) := \langle P_1(x_1) \rangle P_2(x_1) \\ 457 \quad & \text{with}(P_1, P_2:\mathbf{T}(\mathbf{O}, \mathbf{O}))(x_1, x_2:\mathbf{O}) := \langle P_1(x_1, x_2) \rangle P_2(x_1, x_2) \\ & \text{with}(P_1, P_2:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}))(x_1, x_2, x_3:\mathbf{O}) := \langle P_1(x_1, x_2, x_3) \rangle P_2(x_1, x_2, x_3) \end{aligned}$$

458 As for the name **with**, one needs a name that expresses that both are true, but which at  
 459 the same is clearly noncommutative. This is why we came up with **with**. Using **with**, the  
 460 domain, the preconditions and the result of  $\text{minus}$  can be expressed as:

$$461 \quad \text{fromto}(\text{with}(\text{prod}(\text{Nat}, \text{Nat}), \lambda x, y: \mathbf{O} \text{ leq}(y, x)), \text{Nat}, \text{minus}).$$

### 462 3.3 Function Introduction

463 We need a mechanism for introducing functions from relations. As explained above, we do  
 464 not know an approach that is entirely satisfactory. In all cases, we want to avoid using a  
 465 mechanism that relies on global choice, like Isabelle or HOL-light has. We start by defining

$$466 \quad \text{struct Finder} := (u:\mathbf{O}(\mathbf{T}(\mathbf{O}), \mathbf{O}), d:\mathbf{O})$$

467 The field  $u$  is a function that takes a predicate and a default value. If the predicate is strict,  
 468 and  $\mathbf{t}$  for exactly one value, it returns this value, otherwise its second argument. The field  $d$   
 469 is a possible default value that one can optionally use as default value for  $u$ . We define

$$470 \quad \text{unique}(P:\mathbf{T}(\mathbf{O})) := \exists y: \mathbf{O} \langle \#P(y) \rangle P(y) \wedge \forall y': \mathbf{O} [\#P(y')] P(y') \rightarrow y \approx y'$$

471 A predicate  $P$  is unique if it is well-behaved and true at exactly one point. One could try  
 472 other definitions, like requiring strictness (well-behavedness everywhere) and truth at exactly  
 473 one point, but it won't make the usage more pleasant.

$$474 \quad \text{good}(f:\text{Finder}) := \begin{cases} \forall z:\mathbf{O}, Q:\mathbf{T}(\mathbf{O}) \neg \text{unique}(Q) \rightarrow f.u(Q, z) \approx z \\ \forall z:\mathbf{O}, Q:\mathbf{T}(\mathbf{O}) \text{unique}(Q) \rightarrow \langle \#Q(f.u(Q, z)) \rangle Q(f.u(Q, z)) \end{cases}$$

$$\begin{aligned} & \text{serial}(Q:\mathbf{T}(\mathbf{O})) := \exists y:\mathbf{O} Q(y) \\ & \text{serial}(P:\mathbf{T}(\mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O})) := \forall x_1:\mathbf{O} [P(x_1)] \exists y:\mathbf{O} Q(x_1, y) \\ 475 & \text{serial}(P:\mathbf{T}(\mathbf{O}, \mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O})) := \forall x_1, x_2, x_3:\mathbf{O} [P(x_1, x_2)] \exists y:\mathbf{O} Q(x_1, x_2, y) \\ & \text{serial}(P:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}, \mathbf{O})) := \\ & \quad \forall x_1, x_2, x_3:\mathbf{O} [P(x_1, x_2, x_3)] \exists y:\mathbf{O} Q(x_1, x_2, x_3, y) \\ 476 & \text{linear}(Q:\mathbf{T}(\mathbf{O})) := \forall y_1, y_2:\mathbf{O} Q(y_1) \wedge Q(y_2) \rightarrow y_1 \approx y_2 \\ & \text{linear}(P:\mathbf{T}(\mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O})) := \forall x_1:\mathbf{O} [P(x_1)] \forall y_1, y_2:\mathbf{O} Q(x_1, y_1) \wedge Q(x_1, y_2) \rightarrow y_1 \approx y_2 \\ & \text{linear}(P:\mathbf{T}(\mathbf{O}, \mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O})) := \\ 477 & \quad \forall x_1, x_2:\mathbf{O} [P(x_1, x_2)] \forall y_1, y_2:\mathbf{O} Q(x_1, x_2, y_1) \wedge Q(x_2, y_2) \rightarrow y_1 \approx y_2 \\ & \text{linear}(P:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}), Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}, \mathbf{O})) := \\ & \quad \forall x_1, x_2, x_3:\mathbf{O} [P(x_1, x_2, x_3)] \forall y_1, y_2:\mathbf{O} Q(x_1, x_2, x_3, y_1) \wedge Q(x_1, x_2, x_3, y_2) \rightarrow y_1 \approx y_2 \\ 478 & \text{exists}(Q:\mathbf{T}(\mathbf{O})) := \exists y:\mathbf{O} Q(y) \\ & \text{exists}(Q:\mathbf{T}(\mathbf{O}, \mathbf{O}))(x_1:\mathbf{O}) := \exists y:\mathbf{O} Q(x_1, y) \\ 479 & \text{exists}(Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}))(x_1, x_2:\mathbf{O}) := \exists y:\mathbf{O} Q(x_1, x_2, y) \\ & \text{exists}(Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}, \mathbf{O}))(x_1, x_2, x_3:\mathbf{O}) := \text{functional} \exists y:\mathbf{O} Q(x_1, x_2, x_3, y) \\ & \forall f:\text{Finder } P:\mathbf{T}(\mathbf{O}, \mathbf{O}) Q:\mathbf{T}(\mathbf{O}, \mathbf{O}, \mathbf{O}) \\ 480 & \quad [\text{strict}(P)] [\text{stricton}(P, \text{exists}(Q))] \text{serial}(P, Q) \wedge \text{linear}(P, Q) \\ & \quad \rightarrow \forall x_1, x_2:\mathbf{O} [P(x_1, x_2)] Q(x_1, x_2, \text{find}(f, Q, x_1, x_2)) \end{aligned}$$

## 481 **4 Natural Numbers**

482 Apart from the booleans and the unit type, the natural numbers are probably the simplest  
 483 inductive type. The natural numbers are freely generated from 0 and a successor function  
 484 succ. We introduce natural numbers from first principles. The way that we do this shows the  
 485 peculiarities of PHOLI. We do not define or assume 0 and succ globally. Instead we define a  
 486 **struct** containing them both:

$$487 \quad \text{struct Nat} := ( 0:\mathbf{O}; \text{succ}:\mathbf{O}(\mathbf{O}) )$$

488 A Nat can be built from every constant of type  $\mathbf{O}$ , and every function of type  $\mathbf{O}(\mathbf{O})$ . If  $s$  has  
 489 type Nat, then  $s.0$  is its constant, and  $s.\text{succ}$  is its function. In order to construct a Nat,  
 490 one can write  $\text{Nat}(c, f)$ . If everything goes well, Nat will resolve into the constructor of Nat,  
 491 which has type  $\text{Nat}(\mathbf{O}, \mathbf{O}(\mathbf{O}))$ .

492 Some systems fix an existing constant and an existing function satisfying the Peano  
 493 axioms, and define the natural numbers as the smallest set that contains the constant and  
 494 is closed under the function. This is typically done in systems based on set theory, like for  
 495 example  $\mathcal{A}$ etnaNova/Referee ([9]) or Mizar ([2]). One can take  $0 = \emptyset$ , and  $\text{succ}(x) = x \cup \{x\}$ .

496 In other systems, like Isabelle ([8]) or HOL light([5]), natural numbers are defined as an  
 497 inductive type, generated by 0 and succ. In this way the type of natural numbers, 0 and

498 succ are introduced simultaneously and automatically satisfy the Peano axioms. In Isabelle,  
 499 consistency of the axioms has to be proven outside of the system ([7]), while in HOL light,  
 500 consistency of natural numbers is built-in into the system, and other types are reduced to it.

501 In PHOLI, proving properties of natural numbers is separated from using them. When  
 502 proving properties, we simply assume a Seq together with the Peano axioms (or different  
 503 axioms) and prove theorems about it. The Seq is polymorphic, because it can be constructed  
 504 from every constant and function. If one wants to apply natural numbers somewhere, one  
 505 has to construct a Seq, and prove that this Seq satisfies the necessary axioms.

506 The PHOLI approach is somewhat similar to the theories in the *ÆtnaNova/Referee*  
 507 system (see [9] or to locales in Isabelle ([1]), but there are a few differences: Firstly, in  
 508 PHOLI, there is complete separation between structure (definitions and fields) and properties.  
 509 A Locale (or a theory) consists of a sequence of new objects together with required properties  
 510 of these objects, followed by a sequence of consequences. In PHOLI, nearly all types are  
 511 just predicates used in a special way, hence they should not be defined simultaneously with  
 512 structure. As a consequence **struct** Nat defined above, does not necessarily define natural  
 513 numbrs. In order to make it represent natural numbers, the struct needs to satisfy additional  
 514 predicates. Once this approach is adopted, theories are not linear any more. Instead one  
 515 creates small blocks of assumptions that can be used and combined freely as preconditions of  
 516 theorems. A surprising consequence of this approach is that it is convenient to use the range  
 517 of recursion as a Nat.

518 ► **Definition 18.** *Let  $s:\text{Nat}$ . We define when a predicate  $P:\mathbf{T}(\mathbf{O})$  is closed:*

$$519 \quad \text{isclosed}(s:\text{Nat}, P:\mathbf{T}(\mathbf{O})) := P(s.0) \wedge \forall x:\mathbf{O} P(x) \rightarrow P(s.\text{succ}(x)).$$

520 Note that isclosed is well-typed inside the structural type system, but not inside the logic:  
 521 There is nothing a priori that guarantees  $\# \text{isclosed}(s, P)$ . However, it is easy to prove

$$522 \quad \forall s:\text{Nat} P:\mathbf{T}(\mathbf{O}) \text{strict}(P) \rightarrow \# \text{isclosed}(P).$$

523 ► **Definition 19.** *An object is generated by  $s$  if every strict predicate that is closed holds on*  
 524 *it:*

$$525 \quad \text{gen}(s:\text{Nat})(x:\mathbf{O}) := \forall P:\mathbf{T}(\mathbf{O}) [\text{strict}(P)] \text{isclosed}(s, P) \rightarrow P(x).$$

526 Using member notation, one can write  $s.\text{contains}(x)$  instead of  $\text{contains}(s, x)$ . The following  
 527 properties are easy to prove:

$$528 \quad \begin{aligned} & \forall s:\text{Nat} \text{strict}(\text{gen}(s)). \\ & \forall s:\text{Nat} s.\text{gen}(s.0) \\ & \forall s:\text{Nat} s.\text{gen}(x) \rightarrow s.\text{gen}(s.\text{succ}(x)) \end{aligned}$$

529 In set theory, natural numbers can be defined as the smallest set containing 0, and closed  
 530 under successor. In that case, the induction principle is automatic. Definition 19 is similar,  
 531 because we define  $s.\text{gen}$  as the smallest set containing  $s.0$  and closed under  $s.\text{succ}$ , but it  
 532 is not immediately usable as induction principle, because it quantifies over predicates that  
 533 are well-behaved everywhere. We want to be able to use induction with predicates that are  
 534 well-behaved only on the natural numbers themselves, not necessarily anywhere else. For  
 535 this purpose, we prove the following:

$$536 \quad \begin{aligned} & \forall s:\text{Nat} \forall Q:\mathbf{T}(\mathbf{O}) \text{stricton}(s.\text{gen}, Q) \\ & \rightarrow Q(s.0) \rightarrow (\forall x:\mathbf{O} [s.\text{gen}(x)] Q(x) \rightarrow Q(s.\text{succ}(x))) \rightarrow \forall x:\mathbf{O} [s.\text{gen}(x)] \rightarrow Q(x) \end{aligned}$$

537 It can be proven by substituting  $\lambda x: \mathbf{O} \langle s.\text{gen}(x) \rangle Q(x)$  in  $\text{gen}$ .

538 In order to make sure that  $s:\text{Nat}$  is really a set of natural numbers, function  $s.\text{succ}$  must  
 539 be not cyclic. There are two ways to obtain this, one can either require the Peano axioms, or  
 540 require minimality, which means that there exists a functional homomorphic relation into  
 541 every (possibly different)  $\text{Nat}$ . The Peano axioms are as follows:

542 ► **Definition 20.** We define the Peano axioms:

$$543 \text{peano}(s:\text{Nat}) := \begin{cases} \forall x: \mathbf{O} \ s.\text{gen}(x) \rightarrow s.\text{succ}(x) \not\approx s.0 \\ \forall x_1, x_2: \mathbf{O} \ s.\text{gen}(x_1) \wedge s.\text{gen}(x_2) \rightarrow s.\text{succ}(x_1) \approx s.\text{succ}(x_2) \rightarrow x_1 \approx x_2 \end{cases}$$

544 Instead of requiring that a  $\text{Nat}$  satisfies the Peano axioms, one can also require that the  
 545  $\text{Nat}$  is minimal, in the sense that there always exists a functional homomorphic relation into  
 546 every  $\text{Nat}$ . If this is always possible, then the  $\text{Nat}$  must be freely generating, and hence its  
 547  $\text{gen}$  predicate defines the natural numbers. We first define homomorphisms:

548 ► **Definition 21.** We define when a relation between two  $\text{Nat}$ -s is homomorphic:

$$549 \text{homrel}(s_1:\text{Nat}, s_2:\text{Nat})(R: \mathbf{T}(\mathbf{O}, \mathbf{O})) := \bigwedge \begin{cases} R(s_1.0, s_2.0) \\ \forall x_1, x_2: \mathbf{O} \ [s_1.\text{gen}(x_1) \wedge s_2.\text{gen}(x_2)] \\ R(x_1, x_2) \rightarrow R(s_1.\text{succ}(x_1), s_2.\text{succ}(x_2)) \end{cases}$$

550 Using homomorphic relations, one can define that  $\text{Nat}$  is freely generating if there exists a  
 551 linear homomorphic relation into every  $\text{Nat}$ . There are different ways to do this, but the  
 552 easiest way is to use the minimal homomorphic relation, which is the intersection of all  
 553 homomorphic relations. Using

$$554 \begin{aligned} \text{minimal}(P: \mathbf{T}(\mathbf{O}))(C: \mathbf{T}(\mathbf{T}(\mathbf{O}))) (x: \mathbf{O}) &:= \forall R: \mathbf{T}(\mathbf{O}) \ [\text{stricton}(P, R)] \ C(R) \rightarrow R(x) \\ \text{minimal}(P: \mathbf{T}(\mathbf{O}, \mathbf{O}))(C: \mathbf{T}(\mathbf{T}(\mathbf{O}, \mathbf{O}))) (x_1, x_2: \mathbf{O}) &:= \forall R: \mathbf{T}(\mathbf{O}, \mathbf{O}) \ [\text{stricton}(P, R)] \ C(R) \rightarrow R(x_1, x_2) \end{aligned}$$

555 one can define:

► **Definition 22.**

$$556 \begin{aligned} \text{minhomrel}(s_1, s_2:\text{Nat}) &:= \text{minimal}(\text{prod}(s_1.\text{gen}, s_2.\text{gen}), \text{homrel}(s_1, s_2)) \\ \text{freegen}(s:\text{Nat}) &:= \forall s': \text{Nat} \ \text{linear}(s.\text{gen}, \lambda x, y: \mathbf{O} \langle s'.\text{gen} \rangle \text{minhomrel}(s, s')) \end{aligned}$$

557 One can prove the following basic properties:

$$558 \begin{aligned} \forall s_1, s_2:\text{Nat} \ \forall x_1, x_2: \mathbf{O} \ s_1.\text{gen}(x_1) \wedge s_2.\text{gen}(x_2) &\rightarrow \# \text{minhomrel}(s_1, s_2, x_1, x_2) \\ \forall s_1, x_2: \mathbf{O} \ \text{minhomrel}(s_1, s_2, s_1.0, s_2.0) & \\ \forall s_1, s_2: \mathbf{O} \ x_1, x_2: \mathbf{O} \ [s_1.\text{gen}(x_1) \wedge s_2.\text{gen}(x_2)] & \\ \text{minhomrel}(s_1, s_2, x_1, x_2) \rightarrow \text{minhomrel}(s_1, s_2, s_1.\text{succ}(x_1), s_2.\text{succ}(x_2)) & \\ \forall s_1, x_2: \mathbf{O} \ [\text{stricton}(\text{prod}(s_1.\text{gen}, s_2.\text{gen}), R)] \ \text{homrel}(s_1, s_2, R) &\rightarrow \text{serial}(\lambda x, y: \mathbf{O} \langle s_2.\text{gen} \rangle \text{homrel}(s_1, s_2)) \\ \forall s:\text{Nat} \ \text{peano}(s) \rightarrow \text{freegen}(s). & \end{aligned}$$

559 We define a recursion operator:

► **Definition 23.**

$$560 \text{rec}(f: \text{Finder})(s_1, s_2:\text{Nat})(n: \mathbf{O}) := f.u(\lambda n': \mathbf{O} \langle s_2.\text{gen}(n') \rangle \text{minhomrel}(s_1, s_2, n, n'), f.d)$$

561 We can now define a Theory of Natural Numbers:

$$562 \text{NatTheory} := \text{struct}(N: \mathbf{T}(\mathbf{O}), 0: \mathbf{O}, \text{succ}: \mathbf{O}(\mathbf{O}), \text{rec}: \mathbf{O}(\mathbf{O})(\mathbf{T}(\mathbf{O}), \mathbf{O}, \mathbf{O}(\mathbf{O})))$$

$$\begin{aligned}
& \langle \mathbf{strict}(n.N) \rangle \langle N(0) \wedge \mathbf{fromto}(n.N, n.N, n.Nat) \rangle \\
& \forall P: \mathbf{T}(\mathbf{O}) [\mathbf{stricton}(n.N, P)] P(0) \\
& \quad \rightarrow \forall x: \mathbf{O} [n.Nat(x)] P(x) \rightarrow P(n.succ(x)) \\
563 \quad \text{good}(n: \text{NatTheory}) := & \quad \rightarrow \forall x: \mathbf{O} [n.Nat(x)] P(x) \\
& \forall D: \mathbf{T}(\mathbf{O}) d: \mathbf{O} f: \mathbf{O}(\mathbf{O}) [\mathbf{strict}(D)] [D(d) \wedge \mathbf{fromto}(D, D, f)] \\
& \quad \wedge \begin{cases} n.\text{rec}(D, c, f, n.0) \approx c \\ \forall x: \mathbf{O} n.\text{rec}(D, c, f, n.succ(x)) \approx f(n.\text{rec}(D, C, f, x)) \end{cases}
\end{aligned}$$

564 Existence of Natural Numbers follows from basic principles:

$$565 \quad \forall f: \text{Finder} \forall s: \text{Nat} \text{freegen}(n) \rightarrow \text{good}(\text{NatTheory}(n.\text{gen}, n.0, n.\text{succ}))$$

566 If one defines

$$\begin{aligned}
567 \quad \text{sum}(n: \text{NatTheory}) & := \lambda x, y: \mathbf{O} \text{rec}(n.\text{Nat}, x, n.\text{succ}, y) \\
\text{mul}(n: \text{NatTheory}) & := \lambda x, y: \mathbf{O} \text{rec}(n.\text{Nat}, n.0, \lambda z: \mathbf{O} n.\text{sum}(x, z), y)
\end{aligned}$$

568 then one can prove things like:

$$\begin{aligned}
& \forall n: \text{NatTheory} [\text{good}(n)] \forall x_1, x_2: \mathbf{O} [n.N(x_1) \wedge n.N(x_2)] n.\text{sum}(x_1, x_2) \approx n.\text{sum}(x_2, x_1) \\
& \forall n: \text{NatTheory} [\text{good}(n)] \forall x_1, x_2, x_3: \mathbf{O} [n.N(x_1) \wedge n.N(x_2) \wedge n.N(x_3)] \\
& \quad n.\text{sum}(n.\text{sum}(x_1, x_2), x_3) \approx n.\text{sum}(x_1, n.\text{sum}(x_2, x_3)) \\
& \forall n: \text{NatTheory} [\text{good}(n)] \forall x_1, x_2: \mathbf{O} [n.N(x_1) \wedge n.N(x_2)] n.\text{mul}(x_1, x_2) \approx n.\text{mul}(x_2, x_1) \\
569 \quad \forall n: \text{NatTheory} [\text{good}(n)] \forall x_1, x_2, x_3: \mathbf{O} [n.N(x_1) \wedge n.N(x_2) \wedge n.N(x_3)] \\
& \quad n.\text{mul}(n.\text{mul}(x_1, x_2), x_3) \approx n.\text{mul}(x_1, n.\text{mul}(x_2, x_3)) \\
& \forall n: \text{NatTheory} [\text{good}(n)] \forall x_1, x_2, x_3: \mathbf{O} \\
& \quad [n.N(x_1) \wedge n.N(x_2) \wedge n.N(x_3)] \\
& \quad n.\text{mul}(x_1, n.\text{sum}(x_2, x_3)) \approx n.\text{sum}(n.\text{mul}(x_1, x_2), n.\text{mul}(x_1, x_3))
\end{aligned}$$

## 570 **5 Conclusions and Future Work**

571 We have given an overview of a logic designed with the aim of becoming an expressive and  
572 user-friendly framework for representation of mathematics. We called it PHOLI, which stands  
573 for *partial higher-order logic* with *interfaces*. Interfaces are not present any more in the  
574 current versions, but the name sounds nice. The logic is intended to be used for organizing  
575 and checking proofs that occur in theoretical computer science, like formal language theory,  
576 automata theory, logic. etc.

577 In Section 2.5, we gave a compact description of an axiomatic calculus for PHOLI. We  
578 think that this calculus describes PHOLI completely, but unfortunately, it not suitable for  
579 interactive proof checking. It can serve as a framework for checking proofs obtained in other  
580 ways.

581 At the moment of writing we have a prototype implementation. The bottleneck remains  
582 proof editing. The speed of proof editing needs to be shorted by one order of magnitude at  
583 least, in order to become usable. Problems arise from lack of our own experience, lack of a nice  
584 user interface, and lack of a good interactive calculus. We have made steady improvements  
585 in developing a user-friendly calculus, and expect to make more improvements.

586 In the version presented here, it is allowed that an environment holds declarations and  
587 assumptions. This is inherently problematic, because global declarations and assumptions  
588 can be used anywhere, without showing the dependency. We may decide to remove this  
589 feature or allow only a fixed, small set of global declarations and assumptions.

590 **6 AI Statement**

591 No generative AI was used for any part of this paper, nor the accompanying code.

592 **7 About the Submitted Code**

593 The subdirectory **logic** is quite finished. It contains the term data structure, basic functions  
 594 like substitution, contexts, the environment (called *beliefstate*), reductions, and printing  
 595 functions. Subdirectory **parsing** is finished, but uses a slightly different syntax than the  
 596 paper. Subdirectory **calc** implements an interactive calculus that is somewhat usable, but it  
 597 needs to be rewritten. The calculus itself will not change too much, but the normal forms  
 598 will change, and the current **sequent** data structure is problematic. File **tests.cpp** is pure  
 599 chaos. The axiomatic calculus in Section 2.5 is a recent development and nothing of it is  
 600 implemented. Please do not share the code for other purposes than reviewing this paper.

601 **References**

- 
- 602 1 Clemens Ballarin. Tutorial to locales and locale interpretation. In *Contribuciones Científicas*  
 603 *en honor de Miriam Andrés*. Servicio de Publicaciones de la Universidad de La Rioja, Logroño,  
 604 Spain, 2010.
- 605 2 Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Kornilowicz, Roman  
 606 Matuszewski, Adam Naumowicz, Karol Pąk, and Josef Urban. Mizar: State-of-the-art  
 607 and beyond. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and  
 608 Volker Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM*  
 609 *2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in*  
 610 *Computer Science*, pages 261–279. Springer, 2015.
- 611 3 Hans de Nivelle. Theorem proving for classical logic with partial functions by reduction to  
 612 Kleene logic. *Journal of Logic and Computation*, 27(2):509–548, 2017. (Accepted April 2014).
- 613 4 William Farmer. Andrews’ type system with undefinedness. In C. Benzmüller, C. Brown,  
 614 J. Siekmann, and R. Statman, editors, *Festschrift in the honour of Peter Andrews on his 70th*  
 615 *birthday*, 2009. forthcoming.
- 616 5 John Harisson. The hol light theorem prover. <http://www.cl.cam.ac.uk/~jrh13/hol-light/>.
- 617 6 Stephen Cole Kleene. *Introduction to Metamathematics*. Wolters-Noordhoff Publishing,  
 618 North-Holland Publishing Company, American Elsevier Publishing Company, 1971.
- 619 7 Ondřej Kunčar and Andrei Popescu. Safety and conservativity of definitions in HOL and  
 620 isabelle/HOL. In *Principles of Programming Languages (POPL)*. ACM SIGPLAN, 2018.
- 621 8 Larry Paulson, Tobias Nipkow, and Makarius Wenzel. Isabelle. <http://isabelle.in.tum.de/>.
- 622 9 Jacob Schwartz, Domenico Cantone, and Eugenio G. Omodeo. *Computational Logic and Set*  
 623 *Theory*. Springer Verlag, 2011.
- 624 10 Michael Sipser. *Introduction to the Theory of Computation (Third Edition)*. CENGAGE  
 625 Learning, 2013.