Mechanized HOL Reasoning in Set Theory

Simon Guilloud
EPFL, Laboratory for Automated Reasoning and Analysis, Switzerland

Sankalp Gambhir
EPFL, Laboratory for Automated Reasoning and Analysis, Switzerland

Andrea Gilot
EPFL, Laboratory for Automated Reasoning and Analysis, Switzerland

Viktor Kunčak
EPFL, Laboratory for Automated Reasoning and Analysis, Switzerland

Abstract

We present a mechanized embedding of higher-order logic (HOL) and algebraic data types (ADT) into first-order logic with ZFC axioms. We implement this in the Lisa proof assistant for schematic first-order logic and its library based on axiomatic set theory. HOL proof steps are implemented as proof producing tactics in Lisa, and the types are interpreted as sets, with function (or arrow) types coinciding with set-theoretic function spaces. The embedded HOL proofs, as opposed to being a layer over the existing proofs, are interoperable with the existing library. This yields a form of soft type system supporting top-level polymorphism and ADTs over set theory, and offer tools to reason about functions in set theory.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Proof assistant, First Order Logic, Set Theory, Higher Order Logic

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The interactions and combinations of higher-order logic (HOL) with set theory in the context of proof systems have been a long-standing topic of study (e.g. [7, 10, 17, 6, 2]). Set theory (in particular ZF and ZFC) is the prototypical and oldest formalized foundation of mathematics, but it does not naturally admit a concept of "typed" expressions, which are widely used in informal mathematics, for guiding automated proof search, and in programming languages. HOL on the other hand naturally supports typed expressions, type checking and reasoning for simply typed lambda calculus with top level polymorphism.

The first goal of the present work is to study a syntactic embedding of HOL into first-order set theory. It is well known that the Zermelo-Franekel axioms (ZF) imply the existence of a set that is a model of higher-order logic, where types are interpreted as sets, type judgement as set membership and λ-terms as set-theoretic functions [10]. However, the mere fact that models of set theory contain a model of HOL (that is, a semantic embedding) is of little use by itself in practice. To be able to write the syntax and simulate the features of HOL, we need a *syntactic* embedding that transforms expressions in HOL into terms and formulas of first order set theory.

Some proof assistants have explored mixing features of set theory and HOL in various ways in their foundations, such as Egal [5], Isabelle/HOLZF [24] or ProofPeer [25]. A translation of statements and proofs from HOL to set theory has been done for some systems, for example between Isabelle/HOL and Isabelle/ZF [17], between Isabelle/HOL and Isabelle/Mizar [16, 7] or between HOL Light and Metamath [8]. However, both Isabelle/ZF and Metamath, as well as other systems using higher-order Tarski-Grothendieck [7], the Hilbert ϵ operator (as suggested in [10]) or built-in notations for replacement and comprehension allow, in

some form, writing *terms* containing *bound variables*. This is impossible in syntactically strict first-order logic as described in mathematical text books and in first-order automated theorem provers and proof assistants, where only the universal and existential quantifiers, $\forall$ and $\exists$, can bind variables. This makes it impossible to naturally express arbitrary λ-terms of the form $\lambda x.t$ as standalone first-order terms.

Nonetheless, we show that it is possible to embed higher-order logic and λ-terms in first-order set theory without these constructs using a notion of *contexts*, i.e. formulas that gives local assumptions about terms. We study how this embedding impacts decision procedures for type checking and simulating the proof steps of HOL and implement the embedding in the Lisa proof assistant [12], whose foundations are built on first-order set theory. We obtain from this a form of soft type system over set theory, and support for reasoning about functions with HOL-like proofs steps in Lisa. Specifically, we implement the proof steps of HOL Light [13], for the simplicity of its foundations. Incidentally, we hoped this implementation would allow automatic import of theorems from the HOL Light library. However, while this embedding this works well in practice for human-written proofs, which typically don't contain a many high towers of nested λ-abstractions, our initial tests suggest that the embedding has too much overhead in the size of the proofs for the translation of large proofs whose basic building blocks are such λ-abstractions to be of practical use.

In the second part of this paper, with the same motivation of simulating features from type systems into set theory, we describe how Algebraic Data Types (ADTs) can be encoded into set theory. ADTs are types defined inductively by their constructors, one of the simplest examples is that of singly-linked lists of integers, given by $\text{List} = \text{Nil} \mid \text{Cons}(\text{head} : \mathbb{Z},\ \text{tail} : \text{List})$. ADTs and their generalizations are essential constructs in type theory based proof assistants (such as Coq [28] and Lean [21]) and in functional programming languages. Given the description of an ADT in terms of the type signature of its constructors, we show how to define the set corresponding to the type and functions representing the constructors, deriving the desired theorems about induction and injectivity. We show that expressions using ADTs and their constructors can be type checked by the same procedure as expressions from higher-order logic embedded in Lisa. Finally, we extend these results to polymorphic ADTs.

1.1 Contributions

The contribution of this paper is to present a practical embedding of simply typed lambda calculus with polymorphism, of the proof steps of HOL Light and of ADTs into classical ZFC within first-order logic, and its implementation in the Lisa proof assistant.

- We describe how to embed simply typed lambda calculus (and in particular lambda abstractions, which cannot be syntactically expressed in first-order logic) into set theory. Our approach is based on maintaining a context of local definitions, expressing the desired properties about the subterms of λ-terms of the form $\lambda x.t$. If t contains variables other than x, i.e. free variables, we need to encode the closure of $\lambda x.t$ instead, similar to the compilation of programs containing nested function declaration.
- We explain how this encoding allows simulating proof steps and type checking from HOL by producing the corresponding proofs in set theory.
- We implement this embedding and the proof-producing tactics in the Lisa proof assistant[1], allowing reasoning about set-theoretic functions using HOL proof steps.

[1] Disclaimer: as of March 2024, the library of results from Lisa is still under development. As such, some intermediate results regarding set-theoretic function space and the recursion theorem are still "assumed".

- We try to use this embedding to import (parts of) the library of HOL Light, but obtain a negative result because the simulation is too complex and indirect.
- We describe how ADTs can be automatically defined in ZF set theory and how to obtain their key recursive properties derived from the recursion theorem in ZF set theory. We mechanize this system in Lisa, making ADTs and their constructors fully compatible with implemented HOL tactics.

In the present work we picked HOL Light as our reference system for HOL, but with some additional work, the results can be translated to other proof assistants in the HOL-family of proof assistants, such as HOL4 [27], Isabelle/HOL [23], or Candle [1].

Similarly, our target system was Lisa, but none of the results are specific to Lisa, and they transfer to any system using axiomatic set theory over first-order logic, though those are not quite as common, such as Mizar [22]. Our implementation can be found in `https://github.com/epfl-lara/lisa/tree/itp2024-archive`.

2 Preliminaries

There exist several different variations of higher-order logic (HOL). We consider it as it is defined in HOL Light. Its language is the simply typed lambda calculus with top-level polymorphism (or, the Hindley-Milner type system). Each variable in an HOL term is associated with a type, which may contain type variables. The deduction rules of HOL Light are described in Figure 1[2]. We denote by $\mathcal{B}$ the type of Booleans containing two elements, by i the infinite type of individuals, and by $=^{\lambda}$ the built-in function representing equality.

$$\frac{}{\vdash t =_A t}\ \texttt{REFL} \qquad \frac{\Gamma \vdash s =_A t \qquad \Delta \vdash t =_A u}{\Gamma, \Delta \vdash s =_A u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s =_A t \qquad \Delta \vdash u =_A v}{\Gamma, \Delta \vdash s(u) =_A t(v)}\ \texttt{MK\_COMB} \qquad \frac{\Gamma \vdash s =_B t}{\Gamma \vdash (\lambda x : A.s) =_{A \to B} \lambda(x : A.t)}\ \texttt{ABS}$$

$$\frac{}{\vdash \lambda(x : A.t)x =_B t}\ \texttt{BETA} \qquad \frac{}{p \vdash p}\ \texttt{ASSUME}$$

$$\frac{\Gamma \vdash p =_{\mathcal{B}} q \qquad \Delta \vdash p}{\Gamma, \Delta \vdash q}\ \texttt{EQ\_MP} \qquad \frac{\Gamma, q \vdash p \qquad \Delta, p \vdash q}{\Gamma, \Delta \vdash p =_{\mathcal{B}} q}\ \texttt{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma \vdash p}{\Gamma[\vec{x} := \vec{t}] \vdash p[\vec{x} := \vec{t}]}\ \texttt{INST} \qquad \frac{\Gamma \vdash p}{\Gamma[\vec{X} := \vec{A}] \vdash p[\vec{X} := \vec{A}]}\ \texttt{INSTTYPE}$$

$$\frac{}{\vdash (\lambda x.tx) =_A t}\ \texttt{ETA}$$

Figure 1 Deduction rules for higher-order logic as implemented in HOL Light.

Note that the Infinity axiom asserts that i is infinite. It requires definition of logical connectives in HOL, which we do not explain in the present work.

We also assume familiarity with the syntax and deduction rules of first-order logic (see, for example, [20]). We assume Sequent Calculus [9] as our proof system, which is used by our

[2] HOL Light also admits a choice function and an infinity axiom later in the library development, which are justified in ZFC by the choice axiom and infinity axiom. These two additional axioms are largely tangential to the concerns of the present work and hence outside the scope. While ETA is also formally an axiom in HOL Light, we consider it because set-theoretic functions are naturally extensional, and to handle alpha-equivalence in Definition 3.1.

proof assistant Lisa, but the results transfer to other proof systems for first-order logic (FOL). We call *first-order set theory* (FOST) the axiomatic system of ZFC [14, 18] in first order logic. This is also the foundation of the list Lisa proof system [12] in which we implement our result.

In both HOL (see [3]) and FOL, the language can be extended conservatively using the concept of *extension by definition*, as described (for example) in [18, Section 2.10] and [11].

▶ **Theorem 2.1** (Extension by Definition for First Order Logic)**.** Let K be a first order theory (for example, FOST), and ϕ a formula with free variables $y, x_1, ..., x_n$. Suppose $\vdash_K \forall x_1, ..., x_n \exists! y.\phi$ and let the theory K' be K with the addition of a function symbol f of arity n and the axiom $\forall y, y = f(x_1, ..., x_n) \iff \phi$. Then K' is fully conservative over K'

2.1 Set-Theoretic Semantics of HOL

To motivate our translation from HOL to set theory, we review classical set-theoretic semantics of HOL. This allows us to focus first on the semantics of functions and types, without having to worry about if a certain set is expressible, efficiently or at all, as a term in FOST. We interpret types as sets and HOL functions as total set theoretic functions.

As is usual in set theoretic foundations, we identify a function $f : A \to B$ with its graph, that is, as a subset of $A \times B$ such that for every element x of A, there exists exactly one element $y \in B$ where $(x, y) \in f$. We write isFunction(f, A, B) to denote that the set f is a total and functional relation (or simply, a function) from A to B.

We define an operator app such that, for all f such that isFunction(f, A, B) and for all $x \in A$, app$(f, x) = y$ iff $(x, y) \in f$.

▶ **Definition 2.2** (Set theoretic universe)**.** We use the following concepts to give a classical semantics to HOL.

- Fix U to be a set that is a universe of Zermelo set theory, i.e., a containing an infinite set, and closed under powersets, unions, and subsets defined by set comprehension (separation axiom). Consequently, U is closed under Cartesian products. For example, we can take U to be the set $V_{\omega+\omega}$ of the cummulative (von Neumann) hierarchy [14, Chapter 6].
- Let app and isFunction be as above.
- For $A, B \in U$, let $A \Rightarrow B$ denote $\{r \in \mathcal{P}(A \times B) \mid \text{isFunctional}(r, A, B)\}$
- Let $\mathbb{N}$ be the set of natural numbers.
- Let $\bot = \emptyset$, $\top = \{\emptyset\}$ and $\mathbb{B} = \{\bot, \top\}$
- For $A \in U$, let $E(A) =$

  $$\{(x, f) \in (A \times (A \Rightarrow \mathbb{B})) \mid f = \{(y, b) \in (A \times \mathbb{B}) \mid (x = y \to b = \top) \wedge (x \neq y \to b = \bot)\}\}$$

  Note that for all A, $E(A) \in (A \Rightarrow (A \Rightarrow \mathbb{B}))$

▶ **Definition 2.3** (Semantics of HOL)**.** An assignment $\alpha : (V^\lambda \cup T^\lambda) \to U$ is a function such that for all $x : A \in v^\lambda$, $\alpha(x : A) \in \alpha(A)$. We define an interpretation of HOL terms with respect to an assignment:

$$\begin{array}{lcl}
[\![X]\!]_\alpha & = & \alpha(X) \\
[\![T_1^\lambda \to T_2^\lambda]\!]_\alpha & = & [\![T_1^\lambda]\!]_\alpha \Rightarrow [\![T_1^\lambda]\!]_\alpha \\
[\![\mathcal{B}]\!]_\alpha & = & \mathbb{B} \\
[\![i]\!]_\alpha & = & \mathbb{N} \\
[\![x : A]\!]_\alpha & = & \alpha(x) \\
[\![=^\lambda : A \to A \to \mathcal{B}]\!] & = & E([\![A]\!]_\alpha) \\
[\![(f : A \to B)(t : A) : B]\!]_\alpha & = & \text{app}([\![f : A \to B]\!]_\alpha, [\![t : A]\!]_\alpha) \\
[\![(\lambda x : A.\ t : B) : A \to B]\!]_\alpha & = & \{(y, z) \in ([\![A]\!]_\alpha \times [\![B]\!]_\alpha) \mid z = [\![t]\!]_{\alpha[x \mapsto y]}\}
\end{array}$$

▶ **Definition 2.4** (Syntactic and Semantic truth).
For any FOST sequent $s = (l_1, ..., l_n) \vdash (r_1, ..., r_n)$, we write:
- $\vdash s$ if s is provable in FOST
- $U \models s$ if U satisfies $(l_1 = \top \wedge .. \wedge l_n = \top) \implies (r_1 = \top \vee .. \vee r_n = \top)$ in the usual sense of first order models.

For any HOL sequent $s = (l_1, ..., l_n) \vdash r$, we write:
- $\vdash s$ if s is provable in HOL
- $U \models s$ if, for every assignment α, $(\llbracket l_1 \rrbracket_\alpha = \top \wedge .. \wedge \llbracket l_n \rrbracket_\alpha = \top) \implies \llbracket r \rrbracket_\alpha = \top$ holds in U

▶ **Theorem 2.5.** For any term $s : A \in t^\lambda$ and assignment α, $\llbracket s : A \rrbracket_\alpha \in \llbracket A \rrbracket_\alpha$

Proof. By induction on the structure of t. ◀

We can show that all rules of HOL from Figure 1 hold in ZFC, giving the following theorem:

▶ **Theorem 2.6.** For any assignment α and HOL terms $s_1 : \mathcal{B}, ...s_n : \mathcal{B}$ and $t : \mathcal{B} \in t^\lambda$,

$$\text{if } (s_1, ..., s_n \vdash t \text{ and } \forall i.\ \llbracket s_i : \mathcal{B} \rrbracket_\alpha = \top) \text{ then } \llbracket t : \mathcal{B} \rrbracket_\alpha = \top$$

While the above argument shows that HOL has an interpretation into first order set theory, it does not immediately give us a mechanical translation from an HOL proof system to proofs in mechanized set theory. In particular, note that in Definition 2.3, the right-hand side of the lambda case cannot be expressed in the syntax of first order logic. It does not either tell us if and how we can automate the translation of an HOL proof into a FOST proof, or the production of proofs of a statement $t \in A$ that would correspond to type checking. However, note also that the embedding is shallow, in the sense that HOL functions are interpreted as usual set theoretic functions and types of functions as sets of set theoretic functions.

3 From HOL Formulas to First-Order Set Theory Formulas

We wish to define a translation $(\!|\cdot|\!)$ from HOL sequents to FOL sequents such that if an HOL sequent s is provable in HOL then $(\!|s|\!)$ is provable in FOST. Technically, a trivial such embedding would map all sequents to the trivially true sequent. We cannot require that the embedding maps unprovable sequents of HOL to unprovable sequents of FOST, because FOST is strictly more powerful and can prove additional statements. But, we can require that the embedding does not map semantically false statements to provable sequents. This means, for every sequent s of HOL:

1. $\vdash s \implies \vdash (\!|s|\!)$
2. $\vdash (\!|s|\!) \implies U \models s$

Moreover, for the embedding to be of practical use in theorem proving and for import of proofs, we would like the embedding to be as natural as possible, so that we ideally have an embedding $(\!|\cdot|\!) : t^\lambda \to t$, i.e. from terms of HOL to terms of FOST, such that, for every assignment α, we have $\llbracket (\!|s : A|\!) \rrbracket_\alpha = \llbracket s : A \rrbracket_\alpha$. Unfortunately, the syntax of FOST terms does not support λ-abstractions. Of course, the set we denote by

$$\{(y, z) \in (\llbracket A \rrbracket_\alpha \times \llbracket B \rrbracket_\alpha) \mid z = \llbracket t \rrbracket_{\alpha[x \mapsto y]}\}$$

is guaranteed to exist by the Comprehension axiom, but the above expression is not a term in first-order logic. In particular, any variable that appears in a term has to be free, but here, we would want y and z to be bound. While the symbol E was defined similarly with a

comprehension, we need to show the existence and uniqueness of E only once to introduce it with a definitional extension once and for all, as in Theorem 2.1.

We represent λ-abstractions using a variable which is only valid under some *context*, a set of formulas. For example, we can represent the term $\lambda x : \mathcal{B}.x$ using:

- a *variable* λ_1, along with the corresponding
- *context*, the formula $\lambda_1 \in (\mathbb{B} \Rightarrow \mathbb{B}) \ \wedge \ \forall x \in \mathbb{B}.\ \lambda_1(x) = x$

Formally, using the mechanism of extension by definition, we first extend FOST with constant and functional symbols for $\Rightarrow, \mathbb{B}, \mathbb{N}, E$, and app, according to Definition 2.2. We then define $(\!|\cdot|\!)$ as follows:

▶ **Definition 3.1** (Embedding of HOL into FOST). Reserve a special set of variables $\Lambda = \{\lambda_1, \lambda_2, ...\}$ that are used to represent lambda expressions and associate to every HOL term t a single i. In practice, we use a global counter. We use the standard application notation for FOST Terms, so that, for example, $\lambda_2\, y\, y$ really means $\mathrm{app}(\mathrm{app}(\lambda_2, y), y)$.

$$
\begin{array}{lcl}
(\!|X|\!) & = & X \\
(\!|T_1^\lambda \to T_2^\lambda|\!) & = & (\!|T_1^\lambda|\!) \Rightarrow (\!|T_1^\lambda|\!) \\
(\!|\mathcal{B}|\!) & = & \mathbb{B} \\
(\!|i|\!) & = & \mathbb{N} \\
(\!|x : A|\!) & = & x \\
(\!|=^\lambda : A \to A \to \mathcal{B}|\!) & = & E((\!|A|\!)) \\
(\!|(f : A \to B)(t : A) : B|\!) & = & (\!|f : A \to B|\!)\ (\!|t : B|\!) \\
(\!|(\lambda x : A.\ t : B) : A \to B|\!) & = & \lambda_i\, y_1 \ldots y_n \\
 & & \text{Where } y_1,...,y_n \text{ are the free variables of } \lambda x.t, \\
 & & \text{and } \lambda_i \text{ is a variable symbol associated with the term } \lambda x.t
\end{array}
$$

In the last line, λ_i is a representation of the *closure* of the lambda term $\lambda x.t$, and is intended to only be valid under the appropriate defining assumption.

There is another issue with this encoding, which is that we are losing type information associated to variables, as well as the HOL assumption that type variables cannot represent empty types. Fortunately, this can also be solved with contexts.

3.1 HOL in FOST Using Contexts

To translate propositions of HOL into FOST, we need to compute *contexts* of HOL terms. We will need a *non-emptiness context*, to handle type variables, a *typing context*, to carry over information regarding types of variables, a *definition context* to handle abstractions.

The following definition defines ctx^N (non-emptiness), ctx^T (variable typing) and ctx^D (definitions). Assume for simplicity and without loss of generality that variables typed differently have different identifiers, so that, for example x, $x : A$ and $x : B$ do not appear together in the same proof.

▶ **Definition 3.2** (Non-Emptiness Context). The typing context of an HOL term is the set of assumptions $A \neq \emptyset$ for every type variable in the term. This also includes type variables in the type signature of polymorphic constant symbols.

▶ **Definition 3.3** (Typing Context). The typing context of an HOL term is a set of FOST

formulas of the form $x \in T$ and is computed recursively as follows:

$$\begin{array}{lcl} \mathrm{ctx}^T(x:T) & = & \{x \in T\} \\ \mathrm{ctx}^T(c) & = & \emptyset \text{ for } c \text{ a constant symbol} \\ \mathrm{ctx}^T(f\,t) & = & \mathrm{ctx}^T(f) \cup \mathrm{ctx}^T(t) \\ \mathrm{ctx}^T(\lambda x : T.\, t) & = & \mathrm{ctx}^T(t) - \{x \in T\} \end{array}$$

▶ **Definition 3.4** (Definitional Context)**.** The definition context of an HOL term is a set of FOST formulas whose free variables are from Λ and from the set of type variables. It is computed as follows:

$$\begin{array}{lcl} \mathrm{ctx}^D(x:T) & = & \emptyset \\ \mathrm{ctx}^D(c) & = & \emptyset \\ \mathrm{ctx}^D(f\,t) & = & \mathrm{ctx}^D(t) \cup \mathrm{ctx}^D(t) \\ \mathrm{ctx}^D(\lambda x : T.t) & = & \mathrm{ctx}^D(t) \cup \\ & & \{(\lambda_i \in (\!|T_1|\!) \Rightarrow ... \Rightarrow (\!|T_n|\!) \Rightarrow (\!|T|\!) \Rightarrow (\!|\mathrm{type}\,(t)|\!)) \wedge \\ & & \quad \forall y_1 \in T_1, ..., \forall y_n \in T_n . \lambda_i\, y_i \,...\, y_n\, x = (\!|t|\!)\} \\ & & \text{where } y_1 : T_1, ..., y_n : T_n \text{ are the free variables of } t \text{ (without } x). \end{array}$$

λ_i represents the closure of the λ-expression, as in the supercombinator compilation of functional programming languages [15, Chapter 13]. Having λ_i represent the closure of the lambda abstraction rather than the abstraction itself is necessary because otherwise the y_i's would be free in the definition. But this should not be the case if they are supposed to be bound in an outer term. This is illustrated in the third formulas in the following Example 3.5.

The *context*, $\mathrm{ctx}(t)$, of an HOL term t, is $\mathrm{ctx}^N(t) \cup \mathrm{ctx}^T(t) \cup \mathrm{ctx}^D(t)$.

▶ **Example 3.5.** Let $x : X$, $y : Y$, $f : Y \to X$, $g : X \to Y$. We omit type annotations from lambda terms.

$$\begin{array}{lcl} (\!|\lambda x.x|\!) & = & \lambda_1 \\ \mathrm{ctx}(\lambda x.x) & = & \{X \neq \emptyset, \lambda_1 \in X \Rightarrow X \wedge \forall x \in X.(\lambda_1\, x) = x\} \end{array}$$

$$\begin{array}{lcl} (\!|(\lambda x.y)\,(f\,y)|\!) & = & \lambda_2\, y\, (f\, y) \\ \mathrm{ctx}((\lambda x.y)\,(f\,y)) & = & \{X \neq \emptyset, Y \neq \emptyset, y \in Y, g \in X \Rightarrow Y \\ & & (\lambda_2 \in Y \Rightarrow X \Rightarrow Y) \wedge \forall y \in Y. \forall x \in X.(\lambda_2\, y\, x) = y\} \end{array}$$

$$\begin{array}{lcl} (\!|(\lambda y.(\lambda x.y) =^{\lambda}\, g)|\!) & = & \lambda_4\, g \\ \mathrm{ctx}((\lambda y.(\lambda x.y) =^{\lambda}\, g)\, y) & = & \{X \neq \emptyset, Y \neq \emptyset, f \in Y \Rightarrow X, \\ & & (\lambda_2 \in Y \Rightarrow X \Rightarrow Y) \wedge \forall y \in Y. \forall x \in X.(\lambda_2\, y\, x) = y, \\ & & (\lambda_4 \in (X \Rightarrow Y) \Rightarrow Y \Rightarrow \mathbb{B}) \wedge \\ & & \forall g \in X \Rightarrow Y. \forall y \in Y.(\lambda_4\, g\, y = E(A)\,(\lambda_2\, y)\, g)\} \end{array}$$

Note that in the last example, the definition of λ_4 refers to λ_2, and binds the variable y which is free in the λ-abstraction represented by λ_2. Without the closure, y would be free in the definition of λ_2 and could not be bound in the definition of λ_4. Recall that $E(A)$ denotes the meaning of a (curried) equality relation on A.

We can now define the embedding of sequents:

▶ **Definition 3.6.** Let $s = t_1, ..., t_n \vdash t$ be an HOL sequent. Define the embedding $(\!|s|\!)$ as

$$\mathrm{ctx}\,(t_1), ..., \mathrm{ctx}\,(t_n), \mathrm{ctx}\,(t), (\!|t_1|\!) = \top, ..., (\!|t_n|\!) = \top \quad \vdash \quad (\!|t|\!) = \top.$$

3.2 Proof of Type Checking

To produce proofs corresponding to type checking, we define a Lisa proof tactic `ProofType(t: Term)` which for any term t of type T outputs a proof of

$$\mathrm{ctx}(t) \vdash (\!|t|\!) \in (\!|T|\!)$$

As abstractions are represented by typed variables applied to some other free variables in our encoding, the tactic only has to type applicative terms. For example, consider the term $(\lambda x : A.\, y : A)(z : A)$. The corresponding theorem of first-order set theory is:

$$(\lambda_1 \in A \Rightarrow A \Rightarrow A) \wedge (\forall y \in A. \forall x \in A. \lambda_1\, x\, y = y), y \in A, z \in A \;\; \vdash \;\; (\lambda_1\, y)\, z \in A$$

for which our approach generates a proof by recursing on the structure of t and T, using the definition of function spaces.

Polymorphism

More interesting is the typing of polymorphic constants such as HOL equality $=^\lambda$. Its HOL type is $A \to A \to \mathcal{B}$ and its interpretation according to Definition 3.1 is $E(A)$. Hence, the corresponding typing judgement proven by `ProofType` should be $E(A) \in A \to A \to \mathcal{B}$. In simply typed lambda calculus with explicit polymorphism (like System F, see for example [4]), $=^\lambda$ would be given the type $\Lambda A.A \to A \to \mathcal{B}$ to E. The corresponding property for E in FOST is $\forall A.E(A) \in A \Rightarrow A \Rightarrow \mathbb{B}$. This is easy to represent in our translation using free set variables in sequents. We added support for such top-level polymorphism to the `ProofType` tactic, so that it can automatically type polymorphic constants embedded this way.

3.3 Simulating HOL Proofs

The goal of the section is to demonstrate that HOL Light proof steps can be simulated by proofs in our encoding.

▶ **Theorem 3.7** (Simulating HOL Proofs in FOST)**.** Let

$$\frac{s_1 \qquad \ldots \qquad s_n}{s}$$

be an instance of a deduction rule of HOL from Figure 1. Then

$$\frac{(\!|s_1|\!) \qquad \ldots \qquad (\!|s_n|\!)}{(\!|s|\!)}$$

is admissible in FOST (rules of sequent calculus and axioms of set theory).

We state three auxiliary theorems of FOST which will be necessary for the simulation:

▶ **Lemma 3.8.** The following statements are theorems of FOST:

$$x \in A, y \in A \;\vdash\; (E(A)\, x\, y = \top) \Leftrightarrow (x = y) \qquad \text{(Correctness of } E\text{)}$$

$$f \in A \Rightarrow B, g \in A \Rightarrow B, \forall x \in A. f\, x = g\, x \;\vdash\; f = g \qquad \text{(Functional Extensionality)}$$

$$p \in \mathbb{B}, q \in \mathbb{B}, (p = \top) \Leftrightarrow (q = \top) \;\vdash\; p = q \qquad \text{(Propositional Extensionality)}$$

The simulation of a proof step can in general be split in two parts: first produce a proof under arbitrary typing and context assumptions, and then handle the modifications in context. For example, let $x : \mathcal{B}, f : \mathcal{B} \to \mathcal{B}, g : \mathcal{B} \to \mathcal{B}$ and consider a `TRANS` step deducing

$$\frac{\Gamma \vdash x =^\lambda f\,x \qquad \Gamma \vdash f\,x =^\lambda g\,x}{\Gamma \vdash x =^\lambda g\,x}$$

and let $c_\Gamma = \mathrm{ctx}(\Gamma)$. We wish to obtain a proof of

$$\frac{x \in \mathbb{B}, f \in \mathbb{B}{\Rightarrow}\mathbb{B}, c_\Gamma, (\!|\Gamma|\!) \vdash (x =^\lambda f\,x) = \top \qquad x \in \mathbb{B}, f \in \mathbb{B}{\Rightarrow}\mathbb{B}, g \in \mathbb{B}{\Rightarrow}\mathbb{B}, c_\Gamma, (\!|\Gamma|\!) \vdash (f\,x =^\lambda g\,x) = \top}{x \in \mathbb{B}, g \in \mathbb{B}{\Rightarrow}\mathbb{B}, c_\Gamma, (\!|\Gamma|\!) \vdash (x =^\lambda g\,x) = \top}$$

This should follow from applying Correctness of E (Lemma 3.8) to each premise, using transitivity of first order equality, and applying back Correctness of E (Lemma 3.8) to the result. However, to apply this lemma, we need the facts $f\,x \in \mathbb{B}$ and $g\,x \in \mathbb{B}$. Moreover, the $f \in \mathbb{B}{\Rightarrow}\mathbb{B}$ assumption from the premise will stay in the conclusion, yielding:

$$x \in \mathbb{B}, f \in \mathbb{B}{\Rightarrow}\mathbb{B}, g \in \mathbb{B}{\Rightarrow}\mathbb{B}, f\,x \in \mathbb{B}, g\,x \in \mathbb{B}, c_\Gamma, (\!|\Gamma|\!) \vdash (x =^\lambda g\,x) = \top$$

which is a correct conclusion but contains too many assumptions. Fortunately, these assumptions can be eliminated.

Eliminating lingering assumptions

Pursuing the example above, let $L = \{x \in \mathbb{B}, g \in \mathbb{B}{\Rightarrow}\mathbb{B}, c_\Gamma, (\!|\Gamma|\!)\}$ and $R = (x =^\lambda g\,x) = \top$. We want to simulate the following proof step:

$$\frac{f \in \mathbb{B}{\Rightarrow}\mathbb{B}, f\,x \in \mathbb{B}, g\,x \in \mathbb{B}, L \vdash R}{L \vdash R}$$

First, we prove (automatically) the non-elementary typing assumptions $(x \in \mathbb{B}, f \in \mathbb{B}{\Rightarrow}\mathbb{B}) \vdash f\,x \in \mathbb{B}$ by induction over the structure of $f\,x$ (as in Subsection 3.2) and similarly for g. Then, note that f is free everywhere but in its typing assumption: we can quantify it to $\exists f. f \in \mathbb{B}{\Rightarrow}\mathbb{B}$ using the LeftExists rule from first order logic. Now this statement is provable: It can be deduced from the non-emptiness of $\mathbb{B}$. Formally, we obtain the following proof:

$$\dfrac{\dfrac{f \in \mathbb{B}{\Rightarrow}\mathbb{B}, f\,x \in \mathbb{B}, g\,x \in \mathbb{B}, L \vdash R \qquad \dfrac{\cdots}{x \in \mathbb{B}, f \in \mathbb{B}{\Rightarrow}\mathbb{B} \vdash f\,x \in \mathbb{B}}}{f \in \mathbb{B}{\Rightarrow}\mathbb{B}, g\,x \in \mathbb{B}, L \vdash R}\ \text{Cut}}{\vdots}$$

$$\dfrac{\dfrac{\overline{f \in \mathbb{B}{\Rightarrow}\mathbb{B}, L \vdash R}}{\exists f. f \in \mathbb{B}{\Rightarrow}\mathbb{B}, L \vdash R}\ \text{LeftExists} \qquad \dfrac{\cdots}{\vdash \exists f. f \in \mathbb{B}{\Rightarrow}\mathbb{B}}}{L \vdash R}\ \text{Cut}$$

This example covers statements corresponding to type judgement and typing context. In general, there are three kinds of context formulas we need to eliminate: lambda definitions, variable type assignment, type variables non-emptiness. We implement a proof tactic, called `CLEAN` which eliminates context formulas iteratively:

1. Find in the context a definition $def(\lambda_i)$ such that λ_i does not appear anywhere else. Then, using LeftExists, generalize the left handside to $\exists \lambda_i. def(\lambda_i)$. Prove $\exists \lambda_i. def(\lambda_i)$. This is always possible using the adequate type-nonemptiness assumption. Eliminate $\exists \lambda_i. def(\lambda_i)$ using the Cut rule. Iterate on the next definition.
2. Find a variable type assignment $x \in T$. Using LeftExists, generalize to $\exists x. x \in T$. Using the type variables non-Emptiness assumptions, prove that $\exists x. x \in T$ (i.e. T is non-empty). Eliminate $\exists x. x \in T$. Iterate on next unused variable.
3. Find a non-Emptiness assumption $A \neq \emptyset$ for a type variable that does not appear anywhere else. Using LeftExists, generalize to $\exists A. A \neq \emptyset$, which is of course provable without assumption, and eliminate it. Iterate on the next unused type variable.

We make every tactic that possibly eliminates subterms (that is, `TRANS` , `ABS` , `EQ_MP` , `INST` and `INSTTYPE`) call `CLEAN` to eliminate lingering assumptions.

Simulating HOL steps

We briefly hint at how steps of HOL can be simulated in FOST, leaving implicit concerns regarding proofs of type checking and context elimination, which were addressed above.

- `REFL` is simulated with Correctness of E and reflexivity of first-order equality
- `TRANS` is similarly simulated with Correctness of E and reflexivity of first order equality. Note that in HOL Light, `TRANS` requires only alpha-equivalence of the shared terms of the two premises. We explain how this can be handled without assuming that all alpha-equivalent expressions are represented by the same λ_i in the next paragraph.
- `MK_COMB` is simulated with from Correctness of E and substitution of equals for equals of first-order logic.
- `ABS` and `ETA` follow from the definition of the λ_i and from Functional Extensionality (Lemma 3.8).
- `BETA` steps are proven directly from the definition of the λ_i.
- `ASSUME` is simply a Hypothesis step in Sequent calculus
- `EQ_MP` is simulated with Correctness of E and substitution of equals. Similar `TRANS` , it is subsequently made to support alpha-equivalence.
- `DEDUCT_ANTISYM_RULE` steps are proven with Propositional Extensionality
- `INST` follows from instantiation of free variables in first order logic, except that doing so changes the shape of embeddings of abstractions to a non-canonical representation, which need to be transformed back into a canonical representation. We explain this mechanism in detail in the following paragraph.
- `INSTTYPE` is simply instantiation of free variables.

Alpha Equivalence

The steps `TRANS` and `EQ_MP` each take 2 premises with the added requirement that they share some subterm. For concreteness, consider the `TRANS` step:

$$\frac{\Gamma \vdash s =_A t \qquad \Delta \vdash t =_A u}{\Gamma, \Delta \vdash s =_A u}\ \texttt{TRANS}$$

In HOL Light the two terms t_1 and t_2 in the premises are required to be identical *up to alpha equivalence*. However, alpha equivalence does not naturally hold in our encoding: two alpha-equivalent abstractions may be represented by different variables w_i from Definition 3.1. (In fact, in the absence of memoization, even two occurrences of the same lambda can be represented by different variables. In practice, for our import from HOL Light we perform memoization in the constructor of abstractions `λ(x:Var, body: Term)` using de Brujin indices so that alpha equivalent terms HOL terms are mapped to the exact same FOST expression FOST for efficiency. That said, we still wish to show how to support alpha equivalence as a rule.)

For example, consider symbols λ_1 and λ_2, representing abstractions, with the definitions:

$$(\lambda_1 \in A \Rightarrow A) \wedge (\forall x \in A. \lambda_1\, x = x)$$

$$(\lambda_2 \in A \Rightarrow A) \wedge (\forall y \in A. \lambda_2\, y = y)$$

Here, we can use the fact that our local definitions of lambda terms ensure not only existence, but also uniqueness. In particular, under those two assumptions, $\lambda_1 = \lambda_2$ is a consequence

of the extensionality of set-theoretic functions. In fact, using the Eta axiom from HOL (implemented instead as a deduction step `ETA`), alpha equivalence is provable and does not need to be assumed.

▶ **Definition 3.9** (Tactic for Alpha-Conversion)**.** Let `_TRANS` and `_EQ_MP` be restrictions of `TRANS` and `EQ_MP` not supporting alpha-equivalence. We implement a tactic

$$\frac{}{\vdash \lambda x.t = \lambda y.t[x := y]}\ \texttt{ALPHA\_CONV}$$

where

$$\texttt{ALPHA\_CONV}\ x\, y\, t = \texttt{\_TRANS}\ (\texttt{ETA}\ y\, (\lambda x.\, t))\, (\texttt{ABS}\ (\texttt{INST}\ (\texttt{BETA}\ x\, t)\, x\, y)\, y)$$

Note that the first argument of `_TRANS` proves $\lambda x.t = \lambda y.(\lambda x.t)y$ and the second proves $\lambda y.(\lambda x.t)y = \lambda y.t[x := y]$

We can then define a tactic proving the following:

$$\frac{}{\vdash t = u}\ \texttt{ALPHA\_EQUIVALENCE}\quad (\text{if } t \text{ and } u \text{ are alpha-equivalent})$$

which proves the equality by applying `ALPHA_CONV` recursively on t and u. Finally, we can define the complete versions `TRANS` and `EQ_MP` , which apply `ALPHA_EQUIVALENCE` if the shared terms in the input are not strictly equal.

INST Proof Step and Normalization

It may seem at first glance that `INST` is a very easily simulated step: first-order logic admits instantiation of free variables across a sequent (in fact, LISA offers this as a built-in proof step). This however fails to preserve the structure of the embedding. For concreteness again, let $x : A, y : A, p : \mathcal{B}$ and consider the following simple provable HOL statement and its embedding in FOST:

$$\begin{array}{rcl} S & = & \vdash (\lambda x.p = p)y \\ (\!|S|\!) & = & p \in \mathcal{B}, y \in A, def_{\lambda_1} \vdash (\lambda_1\, p\, y) = \top \end{array}$$

where $def_{\lambda_1} = (\lambda_1 \in B \Rightarrow A \Rightarrow \mathcal{B}) \wedge (\forall p \in B.\forall x \in A.\lambda_1\, p\, x = (p =^{\lambda} p))$. Now, suppose $f :: B \Rightarrow B$ is also a variable and consider the effect of the instantiation $p := (fp)$

$$\begin{array}{rcl} S_{[p:=(fp)]} & = & \vdash (\lambda x.(fp) = (fp))y \\ (\!|S|\!)_{[p:=(fp)]} & = & p \in \mathcal{B}, y \in A, def_{\lambda_1} \vdash (\lambda_1\, (fp)\, y) = \top \end{array}$$

But on the other hand, we have

$$(\!|S_{[p:=(fp)]}|\!) \quad = \quad p \in \mathcal{B}, y \in A, def_{\lambda_2} \vdash (\lambda_1\, f\, p\, y) = \top \ , \text{ where}$$

$def_{\lambda_2} = (\lambda_2 \in (B \Rightarrow B) \Rightarrow B \Rightarrow A \Rightarrow \mathcal{B}) \wedge (\forall f \in (B \Rightarrow B).\forall p \in B.\forall x \in A.\lambda_2\, f\, p\, x = (p =^{\lambda} p))$. So, instantiation and embedding do not commute. Moreover, the shape of $S_{[p:=(fp)]}$ does not correspond to the canonical specification of the embedding of HOL terms described in Definition 3.1.

▶ **Definition 3.10.** (The embedding of) an abstraction term t is in *closure-canonical* form if it is of the form $\lambda_i\, x\, y\, z...$ where $x, y, z...$ are the free variables of t and λ_i is a symbol whose local context is as defined by Definition 3.4. A term is in closure-canonical form if all its subterms are in closure-canonical form.

$\lambda_2\, f\, p\, x$ is in closure-canonical form, as any term produced by Definition 3.1, but $(\!|S_{[p:=(fp)]}|\!)$ is not, because the subterm $\lambda_1\,(fp)\,y$ is not in canonical form. Hence, even though it denotes an equivalent statement, $S_{[p:=(fp)]}$ is not a legal expression whose shape other proof tactics expect to receive. To solve this, we implemented a tactic that recursively transforms any non-canonical representation of an HOL term into its closure-canonical form and prove equality between the two. This then allows the `INST` tactic to output a statement in canonical form.

This concludes our simulation of the various proof tactics in FOST leading to Theorem 3.7.

▶ **Corollary 3.11.** Let s be an HOL sequent. Then a proof of s in HOL can be transformed in a proof of $(\!|s|\!)$ in FOST.

We have implemented the transformation and the above tactics in Lisa.

3.4 Defining new constants

HOL Light allows the introduction of new definitions which serve as shorthand for existing terms. This is also possible in Lisa, where if we produce a theorem of the form $\exists!x.P(x)$, we obtain a new constant c and the property $\forall x.P(x) \iff x = c$. However, for this extension to be sound, the definition of a constant can not contain free variables, as otherwise defining $c := x$ would allow proving $\forall x.c = x$. For definitions from HOL, consider for example the term defining the universal quantifier ! in HOL Light (`bool.ml: 243`):

$$\lambda P : A \to \mathcal{B}.P = \lambda x : A.\top$$

It is represented as a certain variable λ_2 while the subterm $\lambda x : A.\top$ is represented by some symbol λ_1. As in the elimination of contexts, we can prove $\exists!\lambda_2.\,\mathrm{ctx}\,(\lambda_2)$, which matches the requirement of extension by definition. The type variable A is reflected as an explicit parameter of the constant (which means that ! is an applied function symbol $!(A)$). However, λ_1 will be free in $\mathrm{ctx}\,(\lambda_2)$, so we need to bundle with the definition of λ_2 all the context definitions that it refers to and prove existence and uniqueness

It is easy to prove, that such a symbol exists under the assumption of the usual context:

$$\mathrm{ctx}\,(\lambda_1), \mathrm{ctx}\,(\lambda_2) \vdash \exists!c.\ c = \lambda_2$$

However when quantifying all assumptions, this will only yield

$$\forall\lambda_1.\ \mathrm{ctx}\,(\lambda_1) \implies \forall\lambda_2.\ \mathrm{ctx}\,(\lambda_2) \implies \exists!c.\ c = \lambda_2$$

while the mechanism of extension by definition requires the $\exists!$ quantifier to be first in the definition. We use an additional FOL theorem that allows us to swap the universal and unique-existential quantifiers

$$\exists!x.P(x) \implies ((\forall x.\ P(x) \implies \exists!y.\ Q(x,y)) \iff (\exists!y.\ \forall x.P(x) \implies Q(x,y)))$$

This fact, alongside proofs that the terms λ_i are uniquely defined by their contexts, we can swap the quantifiers one-by-one to produce the final justification for the definition:

$$\exists!c.\ \forall\lambda_1.\ \mathrm{ctx}\,(\lambda_1) \implies \forall\lambda_2.\ \mathrm{ctx}\,(\lambda_2) \implies c = \lambda_2\ .$$

We generate this proof automatically in our implementation. The proof of typing is generated alongside the symbol by type checking the definition. The theorem corresponding to the definition under appropriate context $\mathrm{ctx}\,(\lambda_1), \mathrm{ctx}\,(\lambda_2) \vdash !(A) = \lambda_2$ is also generated automaticaly.

4 Formalizing Algebraic Data Types

We want to bring the benefits of types into FOST. In particular, algebraic data types are useful when reasoning about inductive data structures such as lists or trees. Their encoding is generally hidden to the user, who only obtains access to their characteristic theorems and definitional mechanisms. We want Lisa to incorporate such mechanisms. Even though ADTs can be encoded within HOL [19], we choose instead for Lisa's implementation to directly define them in FOST. We therefore avoid going through an intermediate encoding but also lay the foundations of further generalization. We start by giving a syntactic definition of algebraic data types.

▶ **Definition 4.1** (Algebraic data types)**.** An *algebraic data type* in set theory is a set A equipped with a finite set of functions $c_i : T_1^i \Rightarrow \cdots \Rightarrow T_n^i \Rightarrow A$ referred to as constructors. All elements of A have to be in the image of one of exactly one of its constructors. T_j^i can refer to A itself, giving algebraic data types their recursive behavior.

We want to allow defining and reasoning about ADTs directly in FOST. Specifically, consider an ADT specification $\{c_i : (S_1^i, \ldots, S_n^i)\}_{i \leq m}$, where S_j^i is either a term with no variables, or a special symbol referring to the ADT itself. We want to output a set A and functions $\{c_i\}_{i \leq m}$ with the following properties:

- (Typing) For all $i \leq m$, $c_i \in (S_1^i \Rightarrow \cdots \Rightarrow S_n^i \Rightarrow A)$
- (Injectivity 1) Every c_i is injective.
- (Injectivity 2) For $x, y \in A$, if $x \in \mathrm{Im}(c_i)$, $y \in \mathrm{Im}(c_j)$ and $i \neq j$ then $x \neq y$
- (Structural induction) A is the smallest set closed under the constructors c_i's. This allows us to write proofs by induction on the structure of the ADT.

▶ **Example 4.2.** Consider the type of boolean linked lists, with specification

$$\mathrm{listbool} = \{\mathrm{nil} : (), \mathrm{cons} : (\mathbb{B}, \mathrm{listbool})\}$$

We want to generate a set listbool and constructors nil and cons such that nil $\in$ listbool and cons $\in \mathbb{B} \Rightarrow$ listbool $\Rightarrow$ listbool. The above properties should hold; for instance cons $\top$ nil $\neq$ cons $\bot$ nil.

We next present our formalization as implemented in Lisa using set theoretic axiomatization. We represent an algebraic data type as a set A of tuples containing the tag of the constructor and the arguments given to it. This ensures that elements of A are in the image of exactly one constructor. For the listbool example, cons $\top$ nil is hence represented as the tuple $(\mathrm{tag}_{\mathrm{cons}}, \top, (\mathrm{tag}_{\mathrm{nil}}))$. In this setting, tags are arbitrary terms that differentiate constructors. They can, for example, be natural numbers or some encoding of the name of the constructor. We define the set A as the least fixpoint of the function

$$F(H) = \bigcup_{i \leq m} \left\{ (\mathrm{tag}_{c_i}, x_1, \ldots, x_n) \;\middle|\; x_k \in \begin{cases} H & \text{if } S_k^i \text{ is a self-reference} \\ S_k^i & \text{otherwise} \end{cases} \right\}$$

The existence of $F(S)$ for every set S is guaranteed by the replacement and the union axioms. In order to characterize the least fixpoint of F, we use the recursion theorem schema of ZF to obtain a unique function f with domain $\mathbb{N}$ (which is also the smallest infinite ordinal ω) such that

$$f(0) = \emptyset$$
$$f(n+1) = F(f(n))$$

Intuitively, $f(n)$ corresponds to all instances of A of height smaller or equal than n. For example, for list of booleans, $f(2)$ is the set

$$\{(\mathrm{tag}_{\mathrm{nil}}), (\mathrm{tag}_{\mathrm{cons}}, \top, (\mathrm{tag}_{\mathrm{nil}})), (\mathrm{tag}_{\mathrm{cons}}, \bot, (\mathrm{tag}_{\mathrm{nil}}))\}$$

▶ **Lemma 4.3.** The class function F admits a least fixpoint given by $A := \bigcup_{n\in\omega} f(n)$.

Proof. If $x_k \in A$ then there is a $n_k \in \omega$ such that $x_k \in f(n_k)$. Since F is monotonic $x_k \in f(\max_{x_k\in A}\, n_k)$. Therefore, for every $(\mathrm{tag}_{c_i}, x_1, \ldots, x_n) \in F(A)$, we have $(\mathrm{tag}_{c_i}, x_1, \ldots, x_n) \in f(\max_{x_k\in A}\, n_k + 1) \subseteq A$. ◀

▶ **Definition 4.4.** We define c_i as the function in $S_1^i \Rightarrow \cdots \Rightarrow S_n^i \Rightarrow A$ such that

$$c_i\, x_1\, \ldots\, x_n = (\mathrm{tag}_{c_i}, x_1, \ldots, x_n)$$

Now that we have a formal definition of A and c_i, we prove that they fulfill the above properties.

▶ **Theorem 4.5.** Let A and $\{c_i\}_{i\leq m}$ constructed as above. The following statements hold in FOST

$$x_1 \in S_1^i, \ldots, x_n \in S_n^i \vdash c_i\, x_1\, \ldots\, x_n \in A \qquad \text{(Typing)}$$

$$x_k \in S_k^i,\, y_k \in S_k^i,\, c_i\, x_1\, \ldots\, x_n = c_i\, y_1\, \ldots\, y_n \vdash \bigwedge_{k\leq n} x_k = y_k \qquad \text{(Injectivity 1)}$$

$$x_k \in S_k^i,\, y_k \in S_k^j,\, i \neq j \vdash c_i\, x_1\, \ldots\, x_n \neq c_j\, y_1\, \ldots\, y_{n'} \qquad \text{(Injectivity 2)}$$

Proof. For typing, we have $c_i\, x_1\, \ldots\, x_n = (\mathrm{tag}_{c_i}, x_1, \ldots, x_n) \in F(A) = A$.
We know that tuples are injective, that is

$$\vdash (\mathrm{tag}_{c_i}, x_1, \ldots, x_n) = (\mathrm{tag}_{c_j}, y_1, \ldots, y_n) \iff \textstyle\bigwedge_{k\leq n} x_k = y_k \wedge \mathrm{tag}_{c_i} = \mathrm{tag}_{c_j}$$

Moreover, tuples of different arities are not equal. Injectivity 1 follows from the right implication while Injectivity 2 from the left one and the fact that tags are uniquely assigned to constructors. ◀

▶ **Theorem 4.6.** Structural induction schema holds on A.

$$\bigwedge_{i\leq m} \left(\forall x_1^i \in S_1^i.\ \hat{P}(x_1^i) \Longrightarrow \cdots \Longrightarrow \forall x_n^i \in S_n^i.\ \hat{P}(x_n^i) \Longrightarrow P(c_i\, x_1^i\, \ldots\, x_n^i)\right) \vdash \forall x \in A.\ P(x)$$

where $\hat{P}(x_k^i) = \begin{cases} P(x_k^i) & \text{if } S_k^i = A \\ \top & \text{if } S_k^i \neq A \end{cases}$

Proof. We first show that for every $n \in \mathbb{N}$, the theorem holds when replacing A by $f(n)$. This follows by induction on n. Since by definition of A, every $x \in A$ is in $f(n)$ for some n, the statement holds for every element of A. ◀

Polymorphic algebraic data types

Algebraic data types can be polymorphic, meaning that the specification of the constructors contain type parameters. This allows, for example, reasoning over generic lists instead of lists of a specific type. We extend our mechanization of ADTs to support such polymorphism. To do so, we generalize A and $\{c_i\}$ to be class functions instead of constant symbols.

Formally, let $\{c_i : (S_1^i, \ldots, S_n^i)\}_{i \leq m}$ be the specification of an algebraic data type that possibly contains variable symbols $X_1, \ldots, X_l$. We define $A(X_1, \ldots, X_l)$ as the fixpoint of $F(X_1, \ldots, X_l)$ where the construction of F carries the same behavior as above. Using this definition, all the properties about ADT are preserved and the constructions is essentially the same.

As in Subsection 3.2, we give a top level polymorphic type to the function symbols A and c_i, so that they can similarly be typechecked. This also implies that all the tactics from the previous section are compatible with terms referring some ADT A and its constructors. To conclude, we show an example of polymorphic lists in Lisa.

▶ **Example 4.7.** The user can define polymorphic lists with the following syntax:

```
1   val define(list: ADT, constructors(nil, cons)) = () | (T, list)
```

where `list`, `nil`, and `cons` are new function symbols. `list` is such that `list(T)` is a set containing all lists over `T` `nil`is a constructor taking one type parameter and no argument and `cons(T)` is a constructor taking one type parameter `T` as well as an element of type `T` and an element of `list(T)` as arguments. This declaration also automatically proves Theorem 4.5 and Theorem 4.6 Our typing tactics can use these properties to type check any expressions containing list, nil and cons.

5 Importing Proofs from HOL Light

While the embedding of HOL as described above allows writing HOL proofs directly in Lisa, we also implement a prototype to attempt the automatic import of theorems and definitions from HOL Light. We chose HOL Light for our import due to its simple foundations, its large library and easily accessible proof export. With some additional work in matching proof steps, the same method can be adapted to other members of the HOL family.

Since the HOL Light kernel does not keep track of proof objects by default, we rely on the `ProofTrace` export system packaged in the HOL Light repository [26]. The system provides a patch to the HOL Light kernel to track every proven statement in the system's execution. The stored proofs are finally exported to JSON files. We modified the existing JSON output syntax slightly to allow its automatic import using standard JSON libraries for Scala. The terms are exported as strings with a simple and unambiguous grammar, and are parsed back by Lisa. After reading the JSON files, the proofs (with indexed steps) are transformed into proof DAGs in an intermediate representation, and finally transformed to Lisa theorems.

Given the Lisa tactics we developed for this purpose, the translation of HOL Light theorems is straightforward, and proceeds by recursing on the proof DAG obtained from HOL Light, translating each proof step to the equivalent Lisa tactic call. Although constant definitions also appear as a single proof step, `DEFINITION`, they must be dealt with separately, as in Subsection 3.4.

After defining a polymorphic constant, however, we change its signature compared to the HOL Light version. For example, the universal quantifier `!: (A -> bool) -> bool` becomes a class function `!` such that $\forall A$. `!(A)` $\in$ (`A` $\Rightarrow \mathcal{B}) \Rightarrow \mathcal{B}$. On subsequent occurrences of `!` in the import, it occurs with an ascribed type, say `!: (T -> bool) -> bool`. This instantiated type is matched against the original, polymorphic type to find the substitution `A` $\mapsto$ `T`. The definition is correspondingly instantiated, and the occurrence is replaced by the Lisa term `!(T)`.

Our prototype implementation of the embedding described here produces a large overhead during this import. We process the first 15 named definitions and theorems as defined in the

HOL Light library, producing 1716 HOL Light kernel steps. These are expanded to just over 300,000 Lisa kernel steps, with the reconstruction and verification taking 93 seconds on a laptop running Linux with an i7-1165G7 CPU and 16GB RAM.

6 Conclusion

We have demonstrated how to embed HOL into conventional first-order logic axiomatization of set theory. Our translation maintains local definitions (at the level of the sequent) of the closure of abstraction terms. We showed that this encoding allows simulating all the core proof steps of higher-order logic. We mechanized this encoding in Lisa, and obtained an interface and tactics for reasoning with typed expressions and set-theoretic functions. We then demonstrated how (possibly polymorphic) ADTs can be mechanized in first-order set theory, and that their representation is compatible with the tactics and type checking we developed for HOL functions.

We also considered alternative encodings of lambda terms. Instead of defining lambdas at the level of the whole sequent, we could place the definition right after the first predicate symbol. In particular, definitions of lambdas become nested, instead of being independent. On one hand, this means that we do not have to compute closures. On the other hand, the defining property of a lambda would often be deep in a formula, and its use would require deconstructing and reconstructing the formula to use the context. Alternatively, we could use an embedding of λ-terms based on combinators from combinatory logic [4]. We did not use fixed combinators such as SKI due to growth in formula size; in the future we may explore the use of parametric combinator families.

While the results we obtain are of practical use and we expect them to become part of the standard Lisa release, the encoding is somewhat complicated and even if most of the machinery can be hidden, it may confuse new users. There is a significant overhead in the size of formulas and the simulation of proofs. This overhead can in the worst case be linear in the size of the formulas, and even if those do not tend to grow indefinitely, a large constant factor may be less than ideal in practice. This also prevented us from importing a larger number of theorems from the HOL Light library. The syntactic restrictions on terms of FOL is the main source of complexity in the translation. For Lisa, this suggests considering extensions of FOL with terms that refer to formulas, such as the definite description operator $\iota x.P$, denoting an individual that is uniquely characterized by the predicate P.

References

1 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A Verified Implementation of HOL Light. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2022.3*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.3.

2 Sten Agerholm and Mike Gordon. Experiments with zf set theory in hol and isabelle. In E. Thomas Schubert, Philip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 32–45, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

3 Rob Arthan. HOL Constant Definition Done Right. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 531–536, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08970-6_34.

4 Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

5 Chad Brown. The Egal Manual, 2014.

6 Chad E. Brown. Combining type theory and untyped set theory. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, pages 205–219, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

7 Chad E. Brown, C. Kaliszyk, and Karol Pak. Higher-Order Tarski Grothendieck as a Foundation for Formal Proof. In *ITP*, 2019. doi:10.4230/LIPIcs.ITP.2019.9.

8 Mario M Carneiro. Conversion of hol light proofs into metamath. *Journal of Formalized Reasoning*, 9(1):187–200, Jan. 2016. URL: https://jfr.unibo.it/article/view/4596, doi:10.6092/issn.1972-5787/4596.

9 G. Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39:176–210, 1935.

10 Mike Gordon. Set theory, higher order logic or both? In W. Brauer, D. Gries, J. Stoer, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125, pages 191–201. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. doi:10.1007/BFb0105405.

11 Simon Guilloud. LISA Reference Manual. EPFL-LARA, February 2023.

12 Simon Guilloud, Sankalp Gambhir, and Viktor Kuncak. LISA – A Modern Proof System. In *14th Conference on Interactive Theorem Proving*, Leibniz International Proceedings in Informatics, pages 17:1–17:19, Bialystok, 2023. Daghstuhl.

13 John Harrison. HOL Light: An Overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674, pages 60–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-03359-9_4.

14 Thomas Jech. *Set theory: The third millennium edition, revised and expanded*. Springer, 2003.

15 Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

16 Cezary Kaliszyk and Karol Pąk. Combining higher-order logic with set theory formalizations. *Journal of Automated Reasoning*, 67(2):20, May 2023. doi:10.1007/s10817-023-09663-5.

17 Alexander Krauss and Andreas Schropp. A mechanized translation from higher-order logic to set theory. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 323–338, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

18 Kenneth Kunen. *Set Theory An Introduction To Independence Proofs*. North Holland, Amsterdam Heidelberg, reprint edition edition, December 1983.

19 Thomas F. Melham. *Automating Recursive Type Definitions in Higher Order Logic*, pages 341–386. Springer New York, New York, NY, 1989. doi:10.1007/978-1-4612-3658-0_9.

20 Elliott Mendelson. *Introduction to Mathematical Logic*. Springer US, Boston, MA, 1987. doi:10.1007/978-1-4615-7288-6.

21 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

22 Adam Naumowicz and Artur Kornilowicz. A Brief Overview of Mizar. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674, pages 67–72, August 2009. doi:10.1007/978-3-642-03359-9_5.

23 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

24 Steven Obua. Partizan games in Isabelle/HOLZF. pages 272–286, November 2006. doi:10.1007/11921240_19.

25 Steven Obua, Jacques Fleuriot, Phil Scott, and David Aspinall. Type inference for zfh. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors,

Intelligent Computer Mathematics, pages 87–101, Cham, 2015. Springer International Publishing.

26 Stanislas Polu. HOL Light / ProofTrace: Modern proof steps recording for hol light. https://github.com/jrh13/hol-light/tree/master/ProofTrace. Accessed: 2024-03-18.

27 Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-71067-7_6.

28 The Coq Development Team. The coq proof assistant, July 2023. doi:10.5281/zenodo.8161141.

A Appendix

A.1 Deduction Rules and axioms for FOL and ZF

$$\frac{}{\Gamma, \phi \vdash \phi, \Delta}\ \text{Hypothesis} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Sigma, \phi \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \text{Cut}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, \phi \vdash \Delta}\ \text{LeftWeakening} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \phi, \Delta}\ \text{RightWeakening}$$

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta}\ \text{LeftAnd} \qquad \frac{\Gamma \vdash \phi, \Delta \qquad \Sigma \vdash \psi, \Pi}{\Gamma, \Sigma \vdash \phi \wedge \psi, \Delta, \Pi}\ \text{RightAnd}$$

$$\frac{\Gamma, \phi \vdash \Delta \qquad \Sigma, \psi \vdash \Pi}{\Gamma, \Sigma, \phi \vee \psi \vdash \Delta, \Pi}\ \text{LeftOr} \qquad \frac{\Gamma \vdash \phi, \psi \Delta}{\Gamma \vdash \phi \vee \psi, \Delta}\ \text{RightOr}$$

$$\frac{\Gamma \vdash \phi, \Delta}{\Gamma, \neg\phi \vdash \Delta}\ \text{LeftNot} \qquad \frac{\Gamma, \phi \vdash \Delta}{\Gamma \vdash \neg\phi, \Delta}\ \text{RightNot}$$

$$\frac{\Gamma, \phi_x[t := x] \vdash \Delta}{\Gamma, \forall x.\phi_x \vdash \Delta}\ \text{LeftForall} \qquad \frac{\Gamma \vdash \phi_x, \Delta}{\Gamma \vdash \forall x.\phi_x, \Delta}\ \text{RightForall}$$

$$\frac{\Gamma, \phi_x \vdash \Delta}{\Gamma, \exists x.\phi_x \vdash \Delta}\ \text{LeftExists} \qquad \frac{\Gamma \vdash \phi_x[x := t], \Delta}{\Gamma \vdash \exists x.\phi_x, \Delta}\ \text{RightExists}$$

$$\frac{\Gamma, \phi[x := t] \vdash \Delta}{\Gamma, t = u, \phi[x := u] \vdash \Delta}\ \text{LeftSubstEq} \qquad \frac{\Gamma \vdash \phi[x := t], \Delta}{\Gamma, t = u \vdash \phi[x := u], \Delta}\ \text{RightSubstEq}$$

$$\frac{}{\vdash t = t}\ \text{Refl} \qquad \frac{\Gamma \vdash \Delta}{\Gamma[\vec{x} := \vec{t}] \vdash \Delta[\vec{x} := \vec{t}]}\ \text{Inst}$$

Figure 2 Deduction rules for first order logic with equality.

$$\frac{}{\forall x.x \notin \emptyset}\ \text{EmptySet} \qquad \frac{}{(\forall z.z \in x \iff z \in y) \iff (x = y)}\ \text{Extensionality}$$

$$\frac{}{x \subset y \iff \forall z.z \in x \implies z \in y}\ \text{Subset} \qquad \frac{}{(z \in \{x, y\}) \iff ((x = z) \vee (y = z))}\ \text{Pair}$$

$$\frac{}{(z \in \mathrm{U}(x)) \iff (\exists y.(y \in x) \wedge (z \in y))}\ \text{Union} \qquad \frac{}{(x \in \mathcal{P}(y)) \iff (x \subset y)}\ \text{Powerset}$$

$$\frac{}{\forall x.(x \neq \emptyset) \implies (\exists y.(y \in x) \wedge (\forall z.z \in x))}\ \text{Foundation} \qquad \frac{}{\exists y.\forall x.x \in y \iff (x \in z \wedge \phi(x))}\ \text{Comprehension}$$

$$\frac{}{\exists x.\emptyset \in x \wedge (\forall y.y \in x \implies \mathrm{U}(\{y, \{y, y\}\}) \in x)}\ \text{Infinity}$$

$$\frac{}{\forall x.(x \in a) \implies \forall y, z.(\psi(x, y) \wedge \psi(x, y)) \implies y = z \implies (\exists b.\forall y.(y \in B) \implies (\exists x.(x \in a) \wedge \psi(x, y)))}\ \text{Replacement}$$

Figure 3 Axioms for Zermelo-Fraenkel set theory.

A.2 ZF Transfinite Recursion Schema

▶ **Theorem A.1** (Transfinite Recursion)**.** Let F be a class function and α an ordinal. There exists a unique function f with domain α such that

$$\forall \beta \in \alpha.\ f(\beta) = F(f|_\beta)$$

Proof. See [14, p. 22], [18, p. 25]. ◀

In Lisa this translates into

```
Theorem( ordinal(a) |-
   existsOne(f, functionalOver(f, a) /\
      forall(b, in(b, a) ==>
         (app(f, b) === F(restrictedFunction(f, b)))
      )
   )
)
```