Noname manuscript No.
(will be inserted by the editor)

A Consistent Foundation for Isabelle/HOL

Ondřej Kunčar · **Andrei Popescu**

the date of receipt and acceptance should be inserted later

Abstract The interactive theorem prover Isabelle/HOL is based on the well understood Higher-Order Logic (HOL), which is widely believed to be consistent (and provably consistent in set theory by a standard semantic argument). However, Isabelle/HOL brings its own personal touch to HOL: *overloaded constant definitions*, used to provide the users with *Haskell-like type classes*. These features are a delight for the users, but unfortunately are not easy to get right as an extension of HOL—they have a history of inconsistent behavior. It has been an open question under which criteria overloaded constant definitions and type definitions can be combined together while still guaranteeing consistency. This paper presents a solution to this problem: non-overlapping definitions and termination of the definition-dependency relation (tracked not only through constants but also through types) ensures relative consistency of Isabelle/HOL.

1 Introduction

Polymorphic HOL, more precisely, Classic Higher-Order Logic with Infinity, Hilbert Choice and Rank-1 Polymorphism, endowed with a mechanism for constant and type definitions, was proposed at the end of the eighties as a logic for interactive theorem provers by Mike Gordon, who also implemented the seminal HOL theorem prover [12]. This system has produced many successors and emulators known under the umbrella term "HOL-based provers" (e.g., HOL4 [2], HOL Light [15], ProofPower [5] and HOL Zero [3]), launching a very successful paradigm in interactive theorem proving.

A main strength of HOL-based provers is a sweet spot in expressiveness versus complexity: on the one hand HOL is sufficient for most mainstream mathematics and computer science applications, and on the other hand it is a well-understood logic. In particular, the

This is an extended version of the conference paper [22]. It includes detailed proofs of the results.

Ondřej Kunčar
Fakultät für Informatik, Technische Universität München, Germany
E-mail: kuncar@in.tum.de

Andrei Popescu
Department of Computer Science, School of Science and Technology, Middlesex University, UK
E-mail: a.popescu@mdx.ac.uk

consistency of HOL has a standard semantic argument, comprehensible to any science graduate: one interprets its types as sets, in particular the function types as sets of functions, and the terms as elements of these sets, in a natural way; the rules of the logic are easily seen to hold in this model.

The definitional mechanism has two flavors:

- New constants c are introduced by equations $c \equiv t$, with t a closed term not containing c
- New types τ are introduced by typedef equations $\tau \equiv t$, where $t : \sigma \to \mathsf{bool}$ is a predicate on an existing type σ (not containing τ anywhere in the types of its subterms)

Again, this mechanism is manifestly consistent by an immediate semantic argument [33]; alternatively, its consistency can be established by regarding definitions as mere abbreviations (which here are non-cyclic by construction).

1.1 Polymorphic HOL with Ad Hoc Overloading

Isabelle/HOL [30, 39] adds its personal touch to the aforementioned sweet spot: it extends polymorphic HOL with a mechanism for (ad hoc) overloading. As an example, consider the following Nominal-style [36] definitions, where prm is the type of finite-support bijections on an infinite type atom, and where we write apply $\pi\ a$ for the application of a bijection π to an atom a and π^{-1} for the inverse of π. The intended behavior of the constant $\bullet : \mathsf{prm} \to \alpha \to \alpha$ (which we use with infix notation) is the application of a permutation to all atoms contained in an element of a type α:

Example 1.

$$\begin{aligned}
&\texttt{consts}\ \ \bullet : \mathsf{prm} \to \alpha \to \alpha\\
&\texttt{defs}\ \ \bullet_{\mathsf{prm}\to\mathsf{atom}\to\mathsf{atom}} = \lambda\pi\ a.\ \mathsf{apply}\ \pi\ a\\
&\texttt{defs}\ \ \bullet_{\mathsf{prm}\to\mathsf{nat}\to\mathsf{nat}} = \lambda\pi\ n.\ n\\
&\texttt{defs}\ \ \bullet_{\mathsf{prm}\to\alpha\ \mathsf{list}\to\alpha\ \mathsf{list}} = \lambda\pi\ xs.\ \mathsf{map}\ (\lambda x.\ \pi \bullet x)\ xs\\
&\texttt{defs}\ \ \bullet_{\mathsf{prm}\to(\alpha\to\beta)\to\alpha\to\beta} = \lambda\pi\ f\ x.\ \pi \bullet f\ (\pi^{-1} \bullet x)
\end{aligned}$$

Above, the constant $\bullet$ is declared using the keyword `consts`. Then, using the keyword `defs`, several overloaded definitions of $\bullet$ are performed for different instances of α. For atoms, $\bullet$ applies the permutation; for numbers (which don't have atoms), $\bullet$ is the identity function; for α list and $\alpha \to \beta$, the instance of $\bullet$ is defined in terms of the instance for the components α and β. All these definitions are non-overlapping and their type-based recursion is terminating, hence Isabelle is fine with them.

1.2 Inconsistency

Of course, one may not be able to specify all the relevant instances immediately after declaring a constant like $\bullet$. For example, at a later point a user may define their own atom-container type, such as

$$\texttt{datatype}\ \ \mathsf{myTree} = \mathsf{Leaf\ atom} \mid \mathsf{LNode\ atom\ list} \mid \mathsf{FNode\ nat} \to \mathsf{atom}$$

and instantiate $\bullet$ for this type. (In fact, the Nominal tool automates instantiations for user-requested datatypes, including terms with bindings.) To support such delayed instantiations, which are also crucial for the implementation of type classes, Isabelle/HOL allows intermixing

definitions of instances of an overloaded constant with definitions of other constants and types. Unfortunately, the improper management of the intermixture leads to inconsistency. Isabelle/HOL used to accept the following definitions:[1]

Example 2.

$$\begin{aligned}&\texttt{consts c} : \alpha\\ &\texttt{typedef } \tau = \{\mathsf{True}, \mathsf{c}_{\mathsf{bool}}\} \texttt{ by blast}\\ &\texttt{defs } \mathsf{c}_{\mathsf{bool}} = \neg\,(\forall x_\tau\ y.\ x = y)\end{aligned}$$

Thus, one first declares a constant c of type α; then, one defines a type that contains the following elements of bool: True and $\mathsf{c}_{\mathsf{bool}}$, the latter being the instance of c for type bool. (The "by blast" part performs a trivial proof of non-emptiness of the set $\{\mathsf{True}, \mathsf{c}_{\mathsf{bool}}\}$, which is required for registering τ as a type.) How many elements does τ have? It has either one, in case $\mathsf{c}_{\mathsf{bool}}$ is True, or two, otherwise. Finally, one defines the instance $\mathsf{c}_{\mathsf{bool}}$ to stand in contrast with the above cardinality analysis for τ: c_{bool} will be the truth value of "the type τ does not have precisely one element." This immediately yield a proof of False:

$$\begin{aligned}&\texttt{lemma L} : (\forall x_\tau\ y.\ x = y) \longleftrightarrow \mathsf{c}\\ &\quad \texttt{using } \mathsf{Rep}_\tau\ \mathsf{Rep}_\tau\texttt{\_inject } \mathsf{Abs}_\tau\texttt{\_inject by } (\texttt{cases } \mathsf{c}_{\mathsf{bool}})\ \texttt{force+}\\ &\texttt{theorem False}\\ &\quad \texttt{using L unfolding c\_bool\_def by auto}\end{aligned}$$

The inconsistency argument takes advantage of the circularity $\tau \rightsquigarrow \mathsf{c}_{\mathsf{bool}} \rightsquigarrow \tau$ in the dependencies introduced by the definitions: one first defines τ to contain only one element just in case $\mathsf{c}_{\mathsf{bool}}$ is True, and then defines $\mathsf{c}_{\mathsf{bool}}$ to be True just in case τ contains more than one element.

1.3 Design Decisions

Before settling the consistency problem for the overloading mechanism, it is instructive to analyze the design decisions that have led to this feature.

Why allow constant overloading in the first place? Because overloading lies at the heart of type classes, which turned out to be a very useful and popular feature of Isabelle/HOL. Substantial developments such as the Nominal [18, 36] and HOLCF [28] tools and Isabelle's mathematical analysis library [17] rely heavily on type classes. One of Isabelle's power users writes [26]: "Thanks to type classes [...] our light-weight framework is flexible, extensible, and easy to use."

Why allow types to depend on constants that are undefined or partially defined (via overloading)? (This is essentially the offending feature in Example 2.) The answer is: To enable users to take full advantage of the type-class convenience. For example, the system allows to reason about types that carry a ring structure (which is achieved by overloading the constants $+, *, 0, 1$ and implicitly assuming the ring axioms for them). But then it is useful to also allow constructions that operate generically on rings, such as the type of polynomials. Besides algebraic structures, other motivating examples come from the world of data structures. Thus, the type (α, β) rbt, of red-black trees storing β-values accessible via α-keys, is carved out of the type of binary trees over $\alpha \times \beta$ by a condition involving a linear

[1] This example works in versions of Isabelle prior to Isabelle2016. A correction patch [1], based on the results reported in this paper and in [21], has been integrated in Isabelle2016.

order on α. In other words, the definition of the type (α, β) rbt depends on the overloaded constant $\leq_{\alpha\to\alpha\to\mathsf{bool}}$:

$$\texttt{typedef}\ \ (\alpha,\beta)\ \mathsf{rbt} = \{t_{(\alpha\times\beta)\ \mathsf{binary\_tree}} \mid \ldots \leq_{\alpha\to\alpha\to\mathsf{bool}} \ldots\}$$

Why not derive type-class overloading from the existing HOL mechanisms? Indeed, reducing overloading to the (overloading-free) logic core would prevent consistency problems. For example, in the Coq proof assistant, type-class overloading (implemented by Sozeau and Oury [35]) uses a dictionary construction. Unfortunately, the dictionary construction is not expressible in HOL. Indeed, with this approach the type rbt would depend on a term variable *le* that represents the linear order:

$$\texttt{typedef}\ \ (\alpha,\beta,le_{\alpha\to\alpha\to\mathsf{bool}})\ \mathsf{rbt} = \{t_{(\alpha\times\beta)\ \mathsf{binary\_tree}} \mid \ldots\ le\ \ldots\}$$

It appears that dependent types, not supported by HOL (and to a large extent incompatible with the HOL philosophy of keeping all types non-empty), would be necessary for compiling out overloading from the logic.

1.4 Our Contribution

In this paper, we provide the following, in the context of polymorphic HOL extended with ad hoc overloading (Section 3):

- a definitional dependency relation that factors in both constant and type definitions in a sensible fashion (Section 4.1)
- a proof of consistency of any set of constant and type definitions whose dependency relation satisfies reasonable conditions, which accept Example 1 and reject Example 2 (Section 4)
- a new semantics for polymorphic HOL (Section 4.4) which guides both our definition of the dependency relation and our proof of consistency

We hope that our work settles the consistency problem for Isabelle/HOL's extension of HOL, while showing that the mechanisms of this logic admit a natural and well-understandable semantics. We start with a discussion of related work, including previous attempts to establish consistency (Section 2). Later we also show how this work fits together with previous work by the first author (Section 5).

2 Related Work

2.1 Type Classes and Overloading

Type classes were introduced in Haskell by Wadler and Blott [37]. They allow programmers to write functions that act generically on types endowed with operations. For example, assuming a type α which is a semigroup (i.e., comes with a binary associative operation $+$), one can write a program that computes the sum of all the elements in a non-empty α-list. Then the program can be run on any concrete type T that replaces α if T provides this binary operation $+$. Prover-powered type classes in Isabelle were introduced by Nipkow and Snelting [31]—in addition to programming language type classes, these enable verifiability of the type-class conditions upon instantiation: a type T is accepted as a member of the semigroup class only if associativity can be proved for its $+$ operation.

Isabelle's type classes rely on arbitrary ad hoc overloading and axiomatic type classes, two primitives of Isabelle's logic, as follows: to introduce the semigroup class, the system declares a global constant $+ : \alpha \to \alpha \to \alpha$ and defines an axiomatic type class that consists of the associativity predicate; then different instance types T are registered after defining the corresponding overloaded operation $+ : T \to T \to T$ and verifying the condition.

Overloading is most useful in conjunction with axiomatic type classes, but it really is an orthogonal feature, which is used also independently of type classes in Isabelle. Furthermore, it is overloading that has been causing consistency problems in Isabelle/HOL, not the axiomatic type classes. Moreover, Wenzel [38] showed how to compile out axiomatic type classes by interpreting them as predicates on types in Isabelle/Pure and therefore; as Wenzel explains, this mechanism is only an addition to the user convenience aspect, without changing the expressiveness of the logic. This is why our current paper focuses not on the Isabelle/HOL type classes, but on the consistency of the mechanism of ad hoc overloading which makes them possible.

We have already mentioned that overloading was introduced in Coq by Sozeau and Oury [35] and uses the dictionary construction to compile out overloaded constants.

Mizar provides overloading for functions, types and other entities of the system (see a description by Grabowski et al. [13]). Moreover, there are two kinds of overloading: ad hoc and parameter overloading. The whole mechanism of retrieving the meaning of an overloaded symbol is rather involved. However, after the theory has been processed, each overloaded symbol is resolved to a unique logical symbol.

Concerning other proof assistants, to the best of our knowledge, there exists no notion of overloading in ACL2, Agda, HOL4, HOL Light, Lean or PVS.

2.2 Previous Consistency Attempts

Establishing the consistency of the mechanism for ad hoc overloading has been previously attempted by Wenzel [38] and Obua [32]. In 1997, Wenzel defined a notion of a safe theory extension and showed that overloading conforms to this notion. But he did not consider type definitions and worked with a simplified version of the system where all overloadings for a constant c are provided at once. Years later, when Obua took over the problem, he found that the overloadings were almost completely unchecked—the following trivial inconsistency was accepted by Isabelle2005:

Example 3.

$$\begin{aligned}
&\texttt{consts } \mathsf{c} : \alpha \to \mathsf{bool}\\
&\texttt{defs } \mathsf{c}_{\alpha\ \mathsf{list}\times\alpha\to\mathsf{bool}} = \lambda x.\ \mathsf{c}(\mathsf{snd}\ x\ \#\ \mathsf{fst}\ x)\\
&\texttt{defs } \mathsf{c}_{\alpha\ \mathsf{list}\to\mathsf{bool}} \quad = \lambda x.\ \neg\ \mathsf{c}(\mathsf{tail}\ x, \mathsf{head}\ x)\\
&\texttt{lemma } \mathsf{c}\ [x] = \neg\ \mathsf{c}([], x) = \neg\ \mathsf{c}[x]
\end{aligned}$$

Obua noticed that the rewrite system produced by the definitions has to terminate to avoid inconsistency, and implemented a separate extension based on a termination checker. He did consider intermixing overloaded constant definitions and type definitions but his syntactic proof sketch failed to consider inconsistency through type definitions.

Triggered by Obua's observations, Wenzel implemented a simpler and more modular solution based on the work of Haftmann, Obua and Urban: fewer overloadings are accepted in order to make the consistency/termination problem decidable (which Obua's original problem is not). Wenzel's solution has been part of the kernel starting from Isabelle2007—aspects

of this solution (which still does not consider dependencies through types) are described by Haftmann and Wenzel [14].

In 2014, we discovered that the dependencies through types are not covered (Example 2), as well as an unrelated issue in the termination checker that led to an inconsistency even without exploiting types. Kunčar [21] amended the latter issue by presenting a modified version of the termination checker and proving its correctness. The proof is general enough to cover termination of the definition dependency relation through types as well. In the conference paper [22] (of which the current paper is an extended version), we complement this result by showing that termination and orthogonality lead to consistency—as detailed in the current paper, the proof of consistency is based on a non-standard notion of model. More recently, we have provided an alternative proof of consistency by means of a syntactic construction [23].

Consistency is a crucial, but rather weak property. In very recent work [24], we prove conservativity of definitions in HOL and Isabelle/HOL, by first proving a stronger property, stating that all definitions can be compiled away from proofs, reducing them to proofs in pure HOL (without definitions). This property generalizes Wenzel's safe theory extension property [38] to cope with type definitions in addition to constant definitions.

2.3 Inconsistency Club

Inconsistency problems arise quite frequently with provers that step outside the safety of a simple and well-understood logic kernel. The various proofs of False in the early PVS system [34] are folklore. Coq's [8] previous stable version[2] is inconsistent in the presence of Propositional Extensionality; this problem stood undetected by the Coq users and developers for 17 years; interestingly, just like the Isabelle/HOL problem under scrutiny, it is due to an error in the termination checker [11]. Agda [10] suffers from similar problems [27]. The recent Dafny prover [25] proposes an innovative combination of recursion and corecursion whose initial version turned out to be inconsistent [9].

Of course, such "dangerous" experiments are often motivated by better support for the users' formal developments. As we already mentioned, the Isabelle/HOL type class experiment was practically successful.

2.4 Consistency Club

Members of this select club try to avoid inconsistencies by impressive efforts of proving soundness of logics and provers by means of interactive theorem provers themselves. Harrison's pioneering work [16] uses HOL Light to give semantic proofs of soundness of the HOL logic without definitional mechanisms, in two flavors: either after removing the infinity axiom from the object HOL logic, or after adding a "universe" axiom to HOL Light; a proof that the OCaml implementation of the core of HOL Light correctly implements this logic is also included.

Kumar et al. [20] formalized in HOL4 the semantics and the soundness proof of HOL, with its definitional principles included; from this formalization, they extract a verified implementation of a HOL theorem prover in CakeML, an ML-like language featuring a verified compiler. None of the above verified systems factor in ad hoc overloading, the starting point of our work.

[2] Namely, Coq 8.4pl6; the inconsistency is now fixed in Coq 8.5.

Krauss and Schropp [19] implemented an automated translation of theories from Isabelle/HOL to Isabelle/ZF—Zermelo–Fraenkel set theory with the axiom of choice. They translate recorded proof terms (the translated proofs are rechecked) and in principle follow the standard semantics approach [33]. Overloaded constants are compiled out by the dictionary construction but their implementation does not support types depending on overloaded constants.

Outside the HOL-based prover family, there are formalizations of Milawa [29], Nuprl [4] and fragments of Coq [6, 7].

3 Polymorphic HOL with Ad Hoc Overloading

Next we present syntactic aspects of our logic of interest (syntax, deduction and definitions) and formulate its *consistency problem*.

3.1 Syntax

In what follows, by "countable set" we mean "either finite or countably infinite set." Throughout the development, we fix the following:

- A countably infinite set TVar, of *type variables*, ranged over by α, β.
- A countably infinite set Var, of *(term) variables*, ranged over by x, y, z
- A countable set K of symbols, ranged over by k, called *type constructors*, containing four special symbols: "bool", "ind" and "$\to$" (aimed at representing the type of booleans, an infinite type, a set type and the function type constructor, respectively).

We also fix a function $\mathsf{arOf} : K \to \mathbb{N}$ associating an arity to each type constructor, such that $\mathsf{arOf}(\mathsf{bool}) = \mathsf{arOf}(\mathsf{ind}) = 0$ and $\mathsf{arOf}(\to) = 2$. We define the set Type, ranged over by σ, τ, of *types*, inductively as follows:

- $\mathsf{TVar} \subseteq \mathsf{Type}$
- $(\sigma_1, \ldots, \sigma_n)\, k \in \mathsf{Type}$ if $\sigma_1, \ldots, \sigma_n \in \mathsf{Type}$ and $k \in K$ such that $\mathsf{arOf}(k) = n$

Thus, we use postfix notation for the application of an n-ary type constructor k to the types $\sigma_1, \ldots, \sigma_n$. If $n = 1$, instead of $(\sigma)\, k$ we write $\sigma\, k$ (e.g., σ list).

A *typed variable* is a pair of a term variable x and a type σ, written x_σ. Given $T \subseteq \mathsf{Type}$, we write Var_T for the set of typed variables x_σ with $\sigma \in T$. Finally, we fix the following:

- A countable set Const, ranged over by c, of symbols called *constants*, containing five special symbols: "$\longrightarrow$", "$=$", "ε", "zero", "suc" (aimed at representing logical implication, equality, Hilbert choice of some element from a type, zero and successor, respectively)
- A function $\mathsf{tpOf} : \mathsf{Const} \to \mathsf{Type}$ associating a type to every constant, such that:
  $\mathsf{tpOf}(\longrightarrow) = \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$
  $\mathsf{tpOf}(=) = \alpha \to \alpha \to \mathsf{bool}$
  $\mathsf{tpOf}(\varepsilon) = (\alpha \to \mathsf{bool}) \to \alpha$
  $\mathsf{tpOf}(\mathsf{zero}) = \mathsf{ind}$
  $\mathsf{tpOf}(\mathsf{suc}) = \mathsf{ind} \to \mathsf{ind}$

We define the type variables of a type, $\mathsf{TV} : \mathsf{Type} \to \mathcal{P}(\mathsf{TVar})$, as expected: $\mathsf{TV}(\alpha) = \{\alpha\}$, $\mathsf{TV}((\sigma_1, \ldots, \sigma_n)\, k) = \bigcup_{1 \le i \le n} \mathsf{TV}(\sigma_i)$.

A *type substitution* is a function $\rho : \mathsf{TVar} \to \mathsf{Type}$; we let TSubst denote the set of type substitutions. Each $\rho \in \mathsf{TSubst}$ extends to a homonymous function $\rho : \mathsf{Type} \to \mathsf{Type}$ by defining $\rho((\sigma_1, \ldots, \sigma_n)\, k) = (\rho(\sigma_1), \ldots, \rho(\sigma_n))\, k$.

We say that σ is an *instance* of τ *via a substitution* of $\rho \in \mathsf{TSubst}$, written $\sigma \leq_\rho \tau$, if $\rho(\tau) = \sigma$. We say that σ is an *instance* of τ, written $\sigma \leq \tau$, if there exists $\rho \in \mathsf{TSubst}$ such that $\sigma \leq_\rho \tau$.

We write $\sigma[\tau/\alpha]$ for $\rho(\sigma)$, where ρ is the type substitution that sends α to τ and each $\beta \neq \alpha$ to β. Thus, $\sigma[\tau/\alpha]$ is obtained from σ by substituting τ for all occurrences of α.

Two types τ_1 and τ_2 are called *orthogonal*, written $\tau_1 \,\#\, \tau_2$, if they have no common instance; i.e., there exists no τ such that $\tau \leq \tau_1$ and $\tau \leq \tau_2$.

Given $c \in \mathsf{Const}$ and $\sigma \in \mathsf{Type}$ such that $\sigma \leq \mathsf{tpOf}(c)$, we call the pair (c, σ), written c_σ, an *instance of* c. A *constant instance* is therefore any such pair c_σ. We let CInst be the set of all constant instances, and we extend the notions of being an instance ($\leq$) and being orthogonal (#) from types to constant instances, as follows:

$$c_\tau \leq d_\sigma \text{ iff } c = d \text{ and } \tau \leq \sigma \qquad\qquad c_\tau \,\#\, d_\sigma \text{ iff } c \neq d \text{ or } \tau \,\#\, \sigma$$

We also extend tpOf to constant instances by $\mathsf{tpOf}(c_\sigma) = \sigma$.

The tuple $(K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf})$, which will be fixed in what follows, is called a *signature*. This signature's *pre-terms* are defined by the grammar:

$$t = x_\sigma \mid c_\sigma \mid t_1\, t_2 \mid \lambda x_\sigma.\, t$$

Thus, a pre-term is either a typed variable, or a constant instance, or an application, or an abstraction. As usual, we identify pre-terms modulo alpha-equivalence.

The typing relation for pre-terms $t : \sigma$ is defined inductively in the expected way:

$$\frac{}{x_\sigma : \sigma} \qquad \frac{}{c_\sigma : \sigma} \qquad \frac{t_1 : \sigma \to \tau \qquad t_2 : \sigma}{t_1\, t_2 : \tau} \qquad \frac{t : \tau}{\lambda x_\sigma.\, t : \sigma \to \tau}$$

A *term* is a well-typed pre-term, and Term, ranged over by s and t, denotes the set of terms. Given $t \in \mathsf{Term}$, we write $\mathsf{tpOf}(t)$ for its (uniquely determined) type and $\mathsf{FV}(t)$ for the set of its free (term) variables. We call t *closed* if $\mathsf{FV}(t) = \emptyset$. We let $\mathsf{TV}(t)$ denote the set of type variables occurring in t.

We can apply a type substitution ρ to a term t, written $\rho(t)$, by applying ρ to all the type variables occurring in t with the proviso that if two distinct bound variables become identified, we replace the term by an alpha-equivalent term where the variables stay distinct. A *well-typed term substitution* is function $\delta : \mathsf{Var}_{\mathsf{Type}} \to \mathsf{Term}$ such that $\mathsf{tpOf}(\delta(x_\sigma)) = \sigma$ for all $x_\sigma \in \mathsf{Var}_{\mathsf{Type}}$. The application of a substitution δ to a term t, written $\delta(t)$, is defined as the term obtained from t by simultaneously substituting each free variable x_σ with the term $\delta(x_\sigma)$ and renaming bound variables if they get captured. Thus, whereas types σ can be subjects to type substitutions, $\rho(\sigma)$, terms t can be subjects to both type substitutions, $\rho(t)$, and (well-typed) term substitutions, $\delta(t)$.

We write $s[t/x_\sigma]$ for $\delta(s)$, where δ is the type substitution that sends each x_σ to t and all other type variables to themselves. Thus, $s[t/x_\sigma]$ is the term obtained from s by substituting t for x_σ.

A *formula* is a term of type bool. We let Fmla, ranged over by φ and χ, denote the set of formulas. When writing concrete terms or formulas, we take the following conventions:

- We may omit redundantly indicating the types of the variables and constants in terms if they can be inferred by typing rules, e.g., we shall write $\lambda x.\ (y_{\sigma\to\tau}\ x)$ instead of $\lambda x_\sigma.\ (y_{\sigma\to\tau}\ x)$ and $\varepsilon\ (\lambda x_\sigma.\ p\ x)$ instead of $\varepsilon_{(\sigma\to\mathsf{bool})\to\sigma}(\lambda x_\sigma.\ p_{\sigma\to\mathsf{bool}}\ x)$.
- We write $\lambda x_\sigma\ y_\tau.\ t$ instead of $\lambda x_\sigma.\ \lambda y_\tau.\ t$.

- We apply the constants $\longrightarrow$ and $=$ in an infix manner, e.g., we write $t_\sigma = s$ instead of $= t_\sigma\ s$.
- We write $\longleftrightarrow$ for $=_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}$.

3.2 Deduction System

The formula connectives and quantifiers are defined in the standard way, starting from the implication and equality primitives:

$$\begin{array}{lcl}
\mathsf{True} & = & (\lambda x_{\mathsf{bool}}.\ x) = (\lambda x_{\mathsf{bool}}.\ x) \\
\mathsf{All} & = & \lambda p_{\alpha\to\mathsf{bool}}.\ (p = (\lambda x.\ \mathsf{True})) \\
\mathsf{Ex} & = & \lambda p_{\alpha\to\mathsf{bool}}.\ \mathsf{All}\ (\lambda q.\ (\mathsf{All}\ (\lambda x.\ p\ x \longrightarrow q)) \longrightarrow q) \\
\mathsf{False} & = & \mathsf{All}\ (\lambda p_{\mathsf{bool}}.\ p) \\
\mathsf{not} & = & \lambda p.\ p \longrightarrow \mathsf{False} \\
\mathsf{and} & = & \lambda p\ q.\ \mathsf{All}\ (\lambda r.\ (p \longrightarrow (q \longrightarrow r)) \longrightarrow r) \\
\mathsf{or} & = & \lambda p\ q.\ \mathsf{All}\ (\lambda r.\ (p \longrightarrow r) \longrightarrow ((q \longrightarrow r) \longrightarrow r))
\end{array}$$

It is easy to see that the above terms are closed and well-typed as follows:

- $\mathsf{tpOf}(\mathsf{True}) = \mathsf{tpOf}(\mathsf{False}) = \mathsf{bool}$
- $\mathsf{tpOf}(\mathsf{not}) = \mathsf{bool} \to \mathsf{bool}$
- $\mathsf{tpOf}(\mathsf{and}) = \mathsf{tpOf}(\mathsf{or}) = \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$
- $\mathsf{tpOf}(\mathsf{All}) = \mathsf{tpOf}(\mathsf{Ex}) = (\alpha \to \mathsf{bool}) \to \mathsf{bool}$

As customary, we shall write:

- $\forall x_\alpha.\ t$ instead of $\mathsf{All}\ (\lambda x_\alpha.\ t)$
- $\exists x_\alpha.\ t$ instead of $\mathsf{Ex}\ (\lambda x_\alpha.\ t)$
- $\neg\ \varphi$ instead of $\mathsf{not}\ \varphi$
- $\varphi \wedge \chi$ instead of $\mathsf{and}\ \varphi\ \chi$
- $\varphi \vee \chi$ instead of $\mathsf{or}\ \varphi\ \chi$

We consider the following formulas:

$$\begin{array}{lcl}
\mathsf{refl} & = & x_\alpha = x \\
\mathsf{subst} & = & x_\alpha = y_\alpha \longrightarrow p_{\alpha\to\mathsf{bool}}\ x \longrightarrow p_{\alpha\to\mathsf{bool}}\ y \\
\mathsf{iff} & = & (p_{\mathsf{bool}} \longrightarrow q_{\mathsf{bool}}) \longrightarrow (q \longrightarrow p) \longrightarrow (p \longleftrightarrow q) \\
\mathsf{True_or_False} & = & (b_{\mathsf{bool}} = \mathsf{True}) \vee (b = \mathsf{False}) \\
\mathsf{some_intro} & = & p_{\alpha\to\mathsf{bool}}\ x_\alpha \longrightarrow p\ (\varepsilon\ p) \\
\mathsf{suc_inj} & = & \mathsf{suc}\ x_{\mathsf{ind}} = \mathsf{suc}\ y_{\mathsf{ind}} \longrightarrow x = y \\
\mathsf{suc_not_zero} & = & \neg\ (\mathsf{suc}\ x_{\mathsf{ind}} = \mathsf{zero})
\end{array}$$

We let Ax denote the set of the above formulas, which we call *axioms*.

Deduction. A *theory* is a set of formulas. A *context* Γ is a finite set of closed formulas. We define *deduction* by the rules shown in Figure 1, as a ternary relation $\vdash$ between theories D, contexts Γ and formulas φ, written $D; \Gamma \vdash \varphi$. These rules are a variant of the standard ones for HOL (as in, e.g., [12, 16]).

Note that HOL deduction is parametrized by an arbitrary theory D. (However, we will only be interested in theories consisting of constant and type definitions—which will be defined later.) A theory D is called *consistent* if $D; \emptyset \nvdash \mathsf{False}$.

$$\frac{}{D;\Gamma\vdash\varphi}\ [\varphi\in\mathsf{Ax}\cup D]\ (\textsc{Fact}) \qquad \frac{}{D;\Gamma\vdash\varphi}\ [\varphi\in\Gamma]\ (\textsc{Assum})$$

$$\frac{D;\Gamma\vdash\varphi}{D;\Gamma\vdash\varphi[\sigma/\alpha]}\ [\alpha\notin\mathsf{TV}(\Gamma)]\ (\text{T-}\textsc{Inst}) \qquad \frac{D;\Gamma\vdash\varphi}{D;\Gamma\vdash\varphi[t/x_\sigma]}\ [x_\sigma\notin\mathsf{FV}(\Gamma)]\ (\textsc{Inst})$$

$$\frac{}{D;\Gamma\vdash(\lambda x_\sigma.\ t)\ s=t[s/x_\sigma]}\ (\textsc{Beta}) \qquad \frac{D;\Gamma\vdash\varphi\longrightarrow\chi \qquad D;\Gamma\vdash\varphi}{D;\Gamma\vdash\chi}\ (\textsc{MP})$$

$$\frac{D;\Gamma\cup\{\varphi\}\vdash\chi}{D;\Gamma\vdash\varphi\longrightarrow\chi}\ (\textsc{Impi}) \qquad \frac{D;\Gamma\vdash y_{\sigma\to\tau}\ x_\sigma=z_{\sigma\to\tau}\ x_\sigma}{D;\Gamma\vdash y_{\sigma\to\tau}=z_{\sigma\to\tau}}\ [x_\sigma\notin\mathsf{FV}(\Gamma)]\ (\textsc{Ext})$$

Fig. 1: HOL deduction

3.3 Built-In and Non-Built-In Types and Constants

The distinction between built-in and non-built-in types and constants will be important, since we will employ a non-standard semantics only for the latter.

A *built-in type* is any type of the form bool, ind or $\sigma\to\tau$ for $\sigma,\tau\in\mathsf{Type}$. We let $\mathsf{Type}^\bullet$ denote the set of *non-built-in types*, i.e., types that are *not* built-in. Note that we look only at the topmost type constructor to decide if a given type is built-in or non-built-in. Therefore, a non-built-in type can have a built-in type as a subexpression, and vice versa; e.g., if list is a type constructor, then bool list and $(\alpha\to\beta)$ list are non-built-in types, whereas $\alpha\to\beta$ list is a built-in type. Also, note that we consider type variables to be non-built-in types.

Given a type σ, we define $\mathsf{types}^\bullet(\sigma)$, the *set of non-built-in types* of σ, as follows:[3]

$$\begin{aligned}
&\mathsf{types}^\bullet(\alpha)=\{\alpha\}\\
&\mathsf{types}^\bullet(\mathsf{bool})=\emptyset\\
&\mathsf{types}^\bullet(\mathsf{ind})=\emptyset\\
&\mathsf{types}^\bullet(\sigma_1\to\sigma_2)=\mathsf{types}^\bullet(\sigma_1)\cup\mathsf{types}^\bullet(\sigma_2)\\
&\mathsf{types}^\bullet((\sigma_1,\ldots,\sigma_n)\,k)=\{(\sigma_1,\ldots,\sigma_n)\,k\},\quad \text{if } k\neq\mathsf{bool},\mathsf{ind},\to
\end{aligned}$$

Thus, $\mathsf{types}^\bullet(\sigma)$ is the smallest set of non-built-in types that can produce σ by repeated application of the built-in type constructors. For example, if the type constructors prm (nullary) and list (unary) are in the signature and if σ is $(\mathsf{bool}\to\alpha\ \mathsf{list})\to\mathsf{prm}\to(\mathsf{bool}\to\mathsf{ind})\ \mathsf{list}$, then $\mathsf{types}^\bullet(\sigma)$ has three elements: α list, prm and $(\mathsf{bool}\to\mathsf{ind})$ list.

A *built-in constant* is a constant of the form $\longrightarrow$, $=$, ε, zero or suc. We let $\mathsf{CInst}^\bullet$ be the set of constant instances that are *not* instances of built-in constants.

In our semantics (Section 4.4), we will stick to the standard interpretation of built-in items, whereas for non-built-in items we will allow an interpretation looser than customary.

Given a term t, we let $\mathsf{consts}^\bullet(t)\subseteq\mathsf{CInst}^\bullet$ be the set of all non-built-in constant instances occurring in t and $\mathsf{types}^\bullet(t)\subseteq\mathsf{Type}^\bullet$ be the set of all non-built-in types that compose the types of non-built-in constants and (free or bound) variables occurring in t. Note that the

[3] We shall consistently use $\bullet$ to indicate non-built-in items.

types$^\bullet$ operator is overloaded for types and terms.

$$\begin{array}{ll}
\mathsf{types}^\bullet(x_\sigma) = \mathsf{types}^\bullet(\sigma) & \mathsf{consts}^\bullet(x_\sigma) = \emptyset \\
\mathsf{types}^\bullet(c_\sigma) = \mathsf{types}^\bullet(\sigma) & \mathsf{consts}^\bullet(c_\sigma) = \begin{cases} \{c_\sigma\} & \text{if } c_\sigma \in \mathsf{CInst}^\bullet \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{types}^\bullet(t_1\ t_2) = \mathsf{types}^\bullet(t_1) \cup \mathsf{types}^\bullet(t_2) & \mathsf{consts}^\bullet(t_1\ t_2) = \mathsf{consts}^\bullet(t_1) \cup \mathsf{consts}^\bullet(t_2) \\
\mathsf{types}^\bullet(\lambda x_\sigma.\ t) = \mathsf{types}^\bullet(\sigma) \cup \mathsf{types}^\bullet(t) & \mathsf{consts}^\bullet(\lambda x_\sigma.\ t) = \mathsf{consts}^\bullet(t)
\end{array}$$

The operators consts$^\bullet$ and types$^\bullet$ commute with type substitutions and behave well w.r.t. free variables, subterms and term substitution.

Lemma 4 The following hold:
(1) $\mathsf{types}^\bullet(\rho(\tau)) = \{\rho(\sigma) \mid \sigma \in \mathsf{types}^\bullet(\tau)\}$
(2) $\mathsf{types}^\bullet(\rho(t)) = \{\rho(\sigma) \mid \sigma \in \mathsf{types}^\bullet(t)\}$
(3) $\mathsf{consts}^\bullet(\rho(t)) = \{c_{\rho(\sigma)} \mid c_\sigma \in \mathsf{consts}^\bullet(t)\}$
(4) If $x_\sigma \in \mathsf{FV}(t)$, then $\mathsf{types}^\bullet(\sigma) \subseteq \mathsf{types}^\bullet(t)$
(5) If t' is a subterm of t, then $\mathsf{types}^\bullet(t') \subseteq \mathsf{types}^\bullet(t)$ and $\mathsf{consts}^\bullet(t') \subseteq \mathsf{consts}^\bullet(t)$
(6) If $\mathsf{tpOf}(t') = \sigma$, then $\mathsf{types}^\bullet(t[t'/x_\sigma]) \subseteq \mathsf{types}^\bullet(t) \cup \mathsf{types}^\bullet(t')$ and $\mathsf{consts}^\bullet(t[t'/x_\sigma]) \subseteq \mathsf{consts}^\bullet(t) \cup \mathsf{consts}^\bullet(t')$

Proof. By straightforward induction on the type τ or the term t, noticing that the application of a type or term substitution leaves unchanged the top type or term constructor. For example, the non-built-in case in the induction for point (1) goes as follows. Assume $k \neq \mathsf{bool}, \mathsf{ind}, \to$. Then, applying the definitions of substitution and of types$^\bullet$ and writing $\rho(\overline{\tau})$ for the tuple obtained from $\overline{\tau}$ by applying ρ componentwise, we have:

$$\mathsf{types}^\bullet(\rho(\overline{\tau}\ k)) = \mathsf{types}^\bullet((\rho(\overline{\tau}))\ k) = \mathsf{types}^\bullet(\{(\rho(\overline{\tau}))\ k\}) = \{\rho(\sigma) \mid \sigma \in \mathsf{types}^\bullet(\overline{\tau}\ k)\}$$

□

3.4 Isabelle/HOL Definitions

We are interested in the consistency of theories arising from constant-instance and type definitions.

Given $c_\sigma \in \mathsf{CInst}^\bullet$ and a closed term t such that $\mathsf{tpOf}(t) = \sigma$, we let $c_\sigma \equiv t$ denote the formula $c_\sigma = t$. We call $c_\sigma \equiv t$ a *constant-instance definition* provided $\mathsf{TV}(t) \subseteq \mathsf{TV}(c_\sigma)$ (i.e., $\mathsf{TV}(t) \subseteq \mathsf{TV}(\sigma)$).

Given the types $\tau \in \mathsf{Type}^\bullet$ and $\sigma \in \mathsf{Type}$ and the closed term t such that $\mathsf{tpOf}(t) = \sigma \to \mathsf{bool}$, we let $\tau \equiv t$ denote the formula

$$\begin{array}{l}
(\exists x_\sigma.\ t\ x) \longrightarrow \\
\quad \exists \mathit{rep}_{\tau\to\sigma}.\ \exists \mathit{abs}_{\sigma\to\tau}. \\
\qquad (\forall x_\tau.\ t\ (\mathit{rep}\ x)) \wedge \\
\qquad (\forall x_\tau.\ \mathit{abs}\ (\mathit{rep}\ x) = x) \wedge \\
\qquad (\forall y_\sigma.\ t\ y \longrightarrow \mathit{rep}\ (\mathit{abs}\ y) = y).
\end{array}$$

We call $\tau \equiv t$ a *type definition*, provided $\mathsf{TV}(t) \subseteq \mathsf{TV}(\tau)$ (which also implies $\mathsf{TV}(\sigma) \subseteq \mathsf{TV}(\tau)$).

Note that we defined $\tau \equiv t$ *not* to mean:

(*): *The type τ is isomorphic, via abs and rep, to t, the subset of σ.*

as customary in most HOL-based systems, but rather to mean:

If t is a nonempty subset of σ, then (*) holds

Moreover, note that we do *not* require τ to have the form $(\alpha_1, \ldots, \alpha_n)\, k$, as is currently required in Isabelle/HOL and the other HOL provers, but, more generally, allow any $\tau \in \mathsf{Type}^\bullet$.[4] This enables an interesting feature: ad hoc overloading for type definitions. For example, given a unary type constructor tree, we can have totally different definitions for $\mathsf{nat\ tree}$, $\mathsf{bool\ tree}$ and $\alpha\ \mathsf{list\ tree}$.

In general, a *definition* will have the form $u \equiv t$, where u is either a constant instance or a type and t is a term (subject to the specific constraints of constant-instance and type definitions). u and t are said to be the left-hand and right-hand sides of the definition.

3.5 The Consistency Problem

An Isabelle/HOL development proceeds by:

1. declaring constants and types
2. defining constant instances and types
3. stating and proving theorems using the deduction rules of polymorphic HOL

Consequently, at any point in the development, one has:

1. a signature $(K, \mathsf{arOf} : K \to \mathbb{N}, \mathsf{Const}, \mathsf{tpOf} : \mathsf{Const} \to \mathsf{Type})$
2. a set of definitions D
3. other proved theorems (i.e., formulas φ such that $D; \emptyset \vdash \varphi$)

In our abstract formulation of Isabelle/HOL's logic, we do not represent explicitly point 3, namely the stored theorems that are not produced as a result of definitions, i.e., are not in D. The reason is that, in Isabelle/HOL, the theorems in D (i.e., the definitions) are not influenced by theorems from outside D. Note that this non-influence property does not hold in other HOL provers—there, a type definition $\tau \equiv t$ (with $\mathsf{tpOf}(t) = \sigma \to \mathsf{bool}$) is introduced directly in the unconditional form (*) after prompting the user to prove that t yields a nonempty subset (i.e., that $\exists x_\sigma.\ t\ x$ holds). Ultimately, Isabelle/HOL's behavior converges with standard HOL behavior since it also prompts the user to prove non-emptiness, after which (*) is inferred by the system—however, this last inference step is normal deduction, being completely decoupled from the definition mechanism. This very useful trick, due to Makarius Wenzel, cleanly separates definitions from proofs. In summary, we only need to guarantee the consistency of D:

> **The Consistency Problem:** Find a sufficient criterion for a set of definitions D to be consistent (while allowing flexible ad hoc overloading for constant definitions).

4 Our Solution to the Consistency Problem

Assume for a moment we have a proper dependency relation between defined items, where the defined items can be types or constant instances. Obviously, the closure of this relation under type substitutions needs to terminate, otherwise inconsistency arises immediately, as

[4] To ensure consistency, we will also require that τ has no common instance with the left-hand side of any other type definition.

shown in Example 3. Moreover, it is clear that the left-hand sides of the definitions need to be orthogonal: defining $c_{\alpha\times\mathsf{bool}\to\mathsf{bool}}$ to be $\lambda x_{\alpha\times\mathsf{bool}}.\mathsf{False}$ and $c_{\mathsf{bool}\times\alpha\to\mathsf{bool}}$ to be $\lambda x_{\mathsf{bool}\times\alpha}.\mathsf{True}$ yields $\lambda x_{\mathsf{bool}\times\mathsf{bool}}.\mathsf{False} = c_{\mathsf{bool}\times\mathsf{bool}\to\mathsf{bool}} = \lambda x_{\mathsf{bool}\times\mathsf{bool}}.\mathsf{True}$ and hence $\mathsf{False} = \mathsf{True}$.

It turns out that these necessary criteria are also *sufficient* for consistency. This was also believed by Wenzel and Obua; what they were missing was a proper dependency relation and a transparent argument for its consistency, which is what we provide next.

4.1 Definitional Dependency Relation

Given any binary relation R on $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$, we write R^+ for its transitive closure, R^* for its reflexive-transitive closure and $R^\downarrow$ for its (type-)substitutive closure, defined as follows: $p\ R^\downarrow\ q$ iff there exist p', q' and a type substitution ρ such that $p = \rho(p')$, $q = \rho(q')$ and $p'\ R\ q'$. We say that a relation R is *terminating* if there exists no sequence $(p_i)_{i\in\mathbb{N}}$ such that $p_i\ R\ p_{i+1}$ for all i.

Let us fix a set of definitions D. We say D is *orthogonal* if for all distinct definitions $u \equiv t$ and $u' \equiv t'$ in D, one of the following cases holds:

- either one of $\{u, u'\}$ is a type and the other is constant instance,
- or both u and u' are types and are orthogonal ($u \# u'$),
- or both u and u' are constant instances and are orthogonal ($u \# u'$).

We define the binary relation $\rightsquigarrow$ on $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$ by setting $u \rightsquigarrow v$ iff one of the following holds:

1. there exists a (constant-instance or type) definition in D of the form $u \equiv t$ such that $v \in \mathsf{types}^\bullet(t) \cup \mathsf{consts}^\bullet(t)$;
2. $u = c_{\mathsf{tpOf}(c)}$ and $v \in \mathsf{types}^\bullet(\mathsf{tpOf}(c))$ for some $c \in \mathsf{Const}^\bullet$.

We call $\rightsquigarrow$ the *dependency relation* (associated to D).

Thus, when defining an item u by means of t (as in $u \equiv t$), we naturally record that u depends on the constants and types appearing in t (clause 1); moreover, any constant c should depend on the components of its type (clause 2). But notice the bullets! We only record dependencies on the non-built-in items—intuitively, the built-in items have a predetermined semantics which cannot be redefined or overloaded, and hence by themselves cannot introduce inconsistencies. Moreover, we do not dig for dependencies under any non-built-in type constructor—this can be seen from the definition of the $\mathsf{types}^\bullet$ operator on types which yields a singleton whenever it meets a non-built-in type constructor; the rationale for this is that a non-built-in type constructor has an "opaque" semantics which does not expose the components (as does the function type constructor). These intuitions will be made precise by our semantics in Section 4.4.

Consider the following example, where the definition of $\alpha\ \mathsf{k}$ is omitted:

Example 5.

$$\texttt{consts c}:\alpha\ \texttt{d}:\alpha$$
$$\texttt{typedef}\ \alpha\,\mathsf{k} = \ldots$$
$$\texttt{defs}\ \mathsf{c}_{\mathsf{ind\ k}\to\mathsf{bool}} = \mathsf{d}_{\mathsf{bool\ k\ k}\to\mathsf{ind\ k}\to\mathsf{bool}}\ \mathsf{d}_{\mathsf{bool\ k\ k}}$$

We record that the constant $\mathsf{c}_{\mathsf{ind\ k}\to\mathsf{bool}}$ depends on the non-built-in constants $\mathsf{d}_{\mathsf{bool\ k\ k}}$ and $\mathsf{d}_{\mathsf{bool\ k\ k}\to\mathsf{ind\ k}\to\mathsf{bool}}$, and on the non-built-in types $\mathsf{bool\ k\ k}$ and $\mathsf{ind\ k}$. We do *not* record any dependency on the built-in types $\mathsf{bool\ k\ k} \to \mathsf{ind\ k} \to \mathsf{bool}$, $\mathsf{ind\ k} \to \mathsf{bool}$ or bool. Also, we do *not* record any dependency on $\mathsf{bool\ k}$, which can only be reached by digging under k in $\mathsf{bool\ k\ k}$.

4.2 The Consistency Theorem

We can now state our main result. We call a set of definitions D a *definitional theory*[5] if it is orthogonal and the substitutive closure of its dependency relation, $\rightsquigarrow^{\downarrow}$, is terminating.

Note that a definitional theory is allowed to contain definitions of two different (but orthogonal) instances of the same constant—as discussed, this ad hoc overloading facility is a distinguishing feature of Isabelle/HOL among the HOL provers.

Theorem 6 If D is a definitional theory, then D is consistent.

Previous attempts to prove consistency employed syntactic methods [32, 38]. Instead, we will give a semantic proof:

1. We define a new semantics of Polymorphic HOL that is suitable for overloading and in which False is not a valid formula (Section 4.4).
2. We prove that models of our semantics are preserved by Isabelle's deduction rules—soundness (Section 4.5).
3. We prove that D has a model according to our semantics (Section 4.6).

Then 1-3 immediately imply consistency.

4.3 Inadequacy of the Standard Semantics of Polymorphic HOL

But why define a new semantics? Recall that our goal is to make sense of definitions as in Example 1. In the standard (Pitts) semantics [33], one chooses a universe collection of sets $\mathcal{U}$ closed under suitable set operations (function space, an infinite set, etc.) and interprets:

1. the built-in type constructors and constants as their standard counterparts in $\mathcal{U}$:
   - $[\mathsf{bool}]$ and $[\mathsf{ind}]$ are some chosen two-element set and infinite set in $\mathcal{U}$
   - $[\to] : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$ takes two sets $A_1, A_2 \in \mathcal{U}$ to the set of functions $A_1 \to A_2$
   - $[\mathsf{True}]$ and $[\mathsf{False}]$ are the two distinct elements of $[\mathsf{bool}]$, etc.
2. the non-built-in type constructors and constants similarly:
   - defined type constructors such as prm (nullary) and list (unary) as elements $[\mathsf{prm}] \in \mathcal{U}$ and operators $[\mathsf{list}] : \mathcal{U} \to \mathcal{U}$, respectively, produced according to their typedef definitions
   - a polymorphic constant such as $\bullet : \mathsf{prm} \to \alpha \to \alpha$ as a family $[\bullet] \in \prod_{A \in \mathcal{U}} [\mathsf{prm}] \to A \to A$

In standard polymorphic HOL, $\bullet$ would be either completely unspecified, or completely defined in terms of previously existing constants—this has a faithful semantic counterpart in $\mathcal{U}$. But what should be the semantics in $\mathcal{U}$ of the overloaded definitions of $\bullet$ from Example 1? The natural attempt would be:

$$[\bullet]_{[\mathsf{atom}]}\ \pi\ a = [\mathsf{apply}]\ \pi\ a$$
$$[\bullet]_{[\mathsf{nat}]}\ \pi\ n = n$$
$$[\bullet]_{[\mathsf{list}](A)}\ \pi\ xs = [\mathsf{map}]_A\ ([\bullet]_A\ \pi)\ xs$$
$$[\bullet]_{A \to B}\ \pi\ f\ x = [\bullet]_B\ \pi\ (f\ ([\bullet]_A\ ([\mathsf{inv}]\ \pi)\ x))$$

[5] In the conference paper [22], what we call here "definitional theory" was called "well-formed definitional theory." We have slightly changed terminology in order to align more faithfully to the official Isabelle documentation [39].

There are several problems with this semantic definition attempt. First, given $B \in \mathcal{U}$, the value of $[\bullet]_B$ varies depending on whether B has the form $[\mathsf{atom}]$, or $[\mathsf{nat}]$, $[\mathsf{list}](A)$ or $A \to B$ for some $A, B \in \mathcal{U}$; hence the interpretations of the type constructors need to be non-overlapping—this is not guaranteed by the assumptions about $\mathcal{U}$, so we would need to perform some low-level set-theoretic tricks to achieve the desired property (such as labeling the interpretation sets with tokens identifying uniquely the intended type constructor). Second, even though the definitions are syntactically terminating, their semantic counterparts may not be: unless we again delve into low-level tricks in set theory (based on the axiom of foundation), it is not guaranteed that decomposing a set A_0 as $[\mathsf{list}](A_1)$, then A_1 as $[\mathsf{list}](A_2)$, and so on (as prescribed by the third equation for $[\bullet]$) is a terminating process.

Even worse, the desired semantic termination is in general a global property, simultaneously involving any number of constants and type constructors. In the following example, c and k are mutually defined (so that a copy of $\mathsf{e}_{\mathsf{bool}\,\mathsf{k}^n}$ is in $\mathsf{bool}\,\mathsf{k}^{n+1}$ iff n is even):

Example 7.

$$\begin{aligned}
&\texttt{consts}\ \mathsf{c} : \alpha \to \mathsf{bool} \quad \mathsf{d} : \alpha \quad \mathsf{e} : \alpha \\
&\texttt{typedef}\ \alpha\,\mathsf{k} \quad = \{\mathsf{d}_\alpha\} \cup \{\mathsf{e}_\alpha \mid \mathsf{c}\,\mathsf{d}_\alpha\} \\
&\texttt{defs}\ \mathsf{c}_{\alpha\,\mathsf{k}\to\mathsf{bool}} = \lambda x_{\alpha\,\mathsf{k}}.\, \neg\,(\mathsf{c}\,\mathsf{d}_\alpha) \\
&\texttt{defs}\ \mathsf{c}_{\mathsf{bool}\to\mathsf{bool}} = \lambda x.\, \mathsf{True}
\end{aligned}$$

The above would require a set-theoretic setting where such fixpoint equations have solutions; this is, in principle, possible, provided we tag the semantic equations with enough syntactic annotations to guide the fixpoint construction. However, such a construction seems excessive given the original intuitive justification: the definitions are "OK" because they do not overlap and they terminate. On the other hand, a purely syntactic (proof-theoretic) argument also seems difficult due to the mixture of constant definitions and (conditional) type definitions.

Therefore, we decide to go for a natural syntactic-semantic blend, which avoids stunt performance in set theory: we do *not* semantically interpret the polymorphic types, but only the ground types—a type σ is called *ground* if $\mathsf{TV}(\sigma) = \emptyset$ and we let GType be the set of ground types. We think of polymorphic types as "macros" for families of ground types. For example, $\alpha \to \alpha\ \mathsf{list}$ represents the family $(\tau \to \tau\ \mathsf{list})_{\tau \in \mathsf{GType}}$. Consequently, we think of the meaning of $\alpha \to \alpha\ \mathsf{list}$ not as $\prod_{A \in \mathcal{U}}(A \to [\mathsf{list}](A))$, but rather as $\prod_{\tau \in \mathsf{GType}}([\tau] \to [\tau\ \mathsf{list}])$. Moreover, a polymorphic formula φ of type, say, $(\alpha \to \alpha\ \mathsf{list}) \to \mathsf{bool}$, is considered true if and only if all its ground instances of types $(\tau \to \tau\ \mathsf{list}) \to \mathsf{bool}$ are true. It will turn out that (polymorphic) HOL deduction is perfectly happy with this view!

Another (small) departure from standard HOL semantics is motivated by our goal to construct a model for a definitional theory. Whereas in standard semantics one first interprets all type constructors and constants and only afterwards extends the interpretation to terms, here we need to interpret some of the terms eagerly *before* some of the types and constants. Namely, given a definition $u \equiv t$, we interpret t before we interpret u (according to t). This requires a straightforward refinement of the notion of semantic interpretation: to interpret a term, we only need the interpretations for a sufficient fragment of the signature containing all the items appearing in t.

4.4 Ground, Fragment-Localized Semantics

Let us start with definitions regarding the ground part of the semantics. Recall that GType is the set of ground types. We let $\mathsf{GType}^\bullet = \mathsf{GType} \cap \mathsf{Type}^\bullet$ denote the set of ground non-built-in types. Clearly $\mathsf{GType}^\bullet \subset \mathsf{Type}$. We let GCInst be the set of constant instances whose type is ground and $\mathsf{GCInst}^\bullet = \mathsf{GCInst} \cap \mathsf{CInst}^\bullet$ be its subset of ground non-built-in instances. As a general notation rule, the prefix "G" will consistently indicate ground items (while the superscript $\bullet$ will indicate non-built-in items), where an item can be either a type or a constant instance.

A *ground type substitution* is a function $\theta : \mathsf{TVar} \to \mathsf{GType}$, which again extends to a homonymous function $\theta : \mathsf{Type} \to \mathsf{GType}$. Note that ground type substitutions can be seen as a particular kind of substitutions, and therefore facts such as Lemma 4(1,2) hold for ground type substitutions as well.

A term t is called *ground* if $\mathsf{TV}(t) = \emptyset$. Thus, closedness refers to the absence of free (term) variables in a term, whereas groundness refers to the absence of type variables in a type or a term. Note that, for a term, being ground is a stronger condition than having a ground type: $(\lambda x_\alpha.\, c_{\mathsf{bool}})\ x_\alpha$ has the ground type bool, but is not ground.

Recall that we can apply a type substitution ρ to a term t, written $\rho(t)$, by applying ρ to all the type variables occurring in t; we use the same notation for ground type substitutions θ; note that $\theta(t)$ is always a ground term.

Given $T \subseteq \mathsf{Type}$, we define $\mathsf{Cl}(T) \subseteq \mathsf{Type}$, the *built-in closure* of T, inductively as follows:

$$\begin{array}{ll} T \subseteq \mathsf{Cl}(T) & \\ \{\mathsf{bool}, \mathsf{ind}\} \subseteq \mathsf{Cl}(T) & \\ \sigma \to \tau \in \mathsf{Cl}(T) & \text{if } \sigma \in \mathsf{Cl}(T) \text{ and } \tau \in \mathsf{Cl}(T) \end{array}$$

This means that $\mathsf{Cl}(T)$ is the smallest set of types built from T by repeatedly applying built-in type constructors.

Lemma 8 Assume $T \subseteq \mathsf{GType}^\bullet$. The following hold:
(1) If $\mathsf{types}^\bullet(\sigma) \subseteq T$, then $\sigma \in \mathsf{Cl}(T)$.
(2) If $\mathsf{types}^\bullet(t) \subseteq T$, then $\mathsf{tpOf}(t) \in \mathsf{Cl}(T)$.
(3) $\mathsf{Cl}(T) \subseteq \mathsf{GType}$.

Proof. (1) By a straightforward induction on the type σ.
(2) By induction on the term t. The base cases x_σ and c_σ follow from point (1).
The application case: Assume $\mathsf{types}^\bullet(t_1\ t_2) \subseteq T$. From the definition of $\mathsf{types}^\bullet$, we obtain $\mathsf{types}^\bullet(t_1) \subseteq T$, hence $\mathsf{tpOf}(t_1) \in \mathsf{Cl}(T)$ by the induction hypothesis. By the definition of typing, we have $\mathsf{tpOf}(t_1) = \mathsf{tpOf}(t_2) \to \mathsf{tpOf}(t_1\ t_2)$, and hence $\mathsf{tpOf}(t_2) \to \mathsf{tpOf}(t_1\ t_2) \in \mathsf{Cl}(T)$. From this, the definition of Cl and the fact that T does not contain types of the form $\sigma_1 \to \sigma_2$ (since $T \subseteq \mathsf{GType}^\bullet$), we obtain $\mathsf{tpOf}(t_1\ t_2) \in \mathsf{Cl}(T)$, as desired.
The λ-abstraction case: Assume $\mathsf{types}^\bullet(\lambda x_\sigma.\, t) \subseteq T$. From the definition of $\mathsf{types}^\bullet$, we obtain $\mathsf{types}^\bullet(\sigma) \subseteq T$ and $\mathsf{types}^\bullet(t) \subseteq T$, hence $\sigma \in \mathsf{Cl}(T)$ and $\mathsf{tpOf}(t) \in \mathsf{Cl}(T)$ by point (1) and the induction hypothesis. By the definitions of typing and of Cl, we obtain $\mathsf{tpOf}(\lambda x_\sigma.\, t) = \sigma \to \mathsf{tpOf}(t) \in \mathsf{Cl}(T)$, as desired.
(3): Immediate by induction on the definition of Cl. □

A *(signature) fragment* is a pair (T, C) with $T \subseteq \mathsf{GType}^\bullet$ and $C \subseteq \mathsf{GCInst}^\bullet$ such that $\sigma \in \mathsf{Cl}(T)$ for all $c_\sigma \in C$.

Let $F = (T, C)$ be a fragment. We write:

- Type^F, for the set of types generated by this fragment, namely $\mathsf{Cl}(T)$
- Term^F, for the set of terms that fall within this fragment, namely $\{t \in \mathsf{Term} \mid \mathsf{types}^\bullet(t) \subseteq T \wedge \mathsf{consts}^\bullet(t) \subseteq C\}$
- Fmla^F, for $\mathsf{Fmla} \cap \mathsf{Term}^F$

Lemma 9 The following hold:

(1) $\mathsf{Type}^F \subseteq \mathsf{GType}$.
(2) $\mathsf{Term}^F \subseteq \mathsf{GTerm}$.
(3) If $t \in \mathsf{Term}^F$, then $\mathsf{tpOf}(t) \in \mathsf{Type}^F$.
(4) If $t \in \mathsf{Term}^F$, then $\mathsf{FV}(t) \subseteq \mathsf{Term}^F$.
(5) If $t \in \mathsf{Term}^F$, then each subterm of t is also in Term^F.
(6) If $t, t' \in \mathsf{Term}^F$, $\mathsf{tpOf}(t') = \sigma$ and $x_\sigma \in \mathsf{Var}_{\mathsf{Type}^F}$, then $t[t'/x_\sigma] \in \mathsf{Term}^F$.

Proof. In what follows, we assume $F = (T, C)$.
(1): Since $T \subseteq \mathsf{GType}^\bullet$, we have $\mathsf{Type}^F = \mathsf{Cl}(T) \subseteq \mathsf{GType}$ by Lemma 8.(3).
(2): Let $t \in \mathsf{Term}^F$. Since $\mathsf{types}^\bullet(t) \subseteq T \subseteq \mathsf{GType}^\bullet$, in particular we have $t \in \mathsf{GTerm}$, as desired.
(3): Let $t \in \mathsf{Term}^F$. Since $\mathsf{types}^\bullet(t) \subseteq T \subseteq \mathsf{GType}^\bullet$, from Lemma 8.(2) we have $\mathsf{tpOf}(t) \in \mathsf{Cl}(T) = \mathsf{Type}^F$, as desired.
(4): Assume $t \in \mathsf{Term}^F$ and $x_\sigma \in \mathsf{FV}(t)$. Then $\sigma \in \mathsf{types}^\bullet(t) \subseteq T$, and therefore $\mathsf{types}^\bullet(x_\sigma) = \mathsf{types}^\bullet(\sigma) \subseteq T$ by Lemma 4.(4). And since $\mathsf{consts}^\bullet(x_\sigma) = \emptyset$, it follows that $x \in \mathsf{Term}^F$, as desired.
(5) If t' is a subterm of t, then $\mathsf{types}^\bullet(t') \subseteq \mathsf{types}^\bullet(t)$ and $\mathsf{consts}^\bullet(t') \subseteq \mathsf{consts}^\bullet(t)$ by Lemma 4.(5). Hence $t' \in \mathsf{Term}^F$ follows from $t \in \mathsf{Term}^F$.
(6) We have that $\mathsf{types}^\bullet(t[t'/x_\sigma]) \subseteq \mathsf{types}^\bullet(t) \cup \mathsf{types}^\bullet(t')$ and $\mathsf{consts}^\bullet(t[t'/x_\sigma]) \subseteq \mathsf{consts}^\bullet(t) \cup \mathsf{consts}^\bullet(t')$ by Lemma 4.(6). Hence $t[t'/x_\sigma] \in \mathsf{Term}^F$ follows from $t, t' \in \mathsf{Term}^F$. □

The above lemma shows that fragments F include only ground items (points (1) and (2)) and they are "autonomous" parts of signatures: the type of a term from F is also in F (3), and similarly for the free (term) variables (4), subterms (5) and substituted terms (6). This autonomy allows us to define semantic interpretations for fragments.

For the rest of this section, we fix the following:

- a singleton set $\{*\}$
- a two-element set $\{\mathsf{true}, \mathsf{false}\}$
- a global choice function, choice, that assigns to each nonempty set A an element $\mathsf{choice}(a) \in A$

Let $F = (T, C)$ be a fragment. An *F-interpretation* is a pair $\mathcal{I} = (([\tau])_{\tau \in T}, ([c_\tau])_{c_\tau \in C})$ such that:

1. $([\tau])_{\tau \in T}$ is a family such that $[\tau]$ is a nonempty set for all $\tau \in T$.
   We extend this to a family $([\tau])_{\tau \in \mathsf{Cl}(T)}$ by interpreting the built-in type constructors as expected:
   $[\mathsf{bool}] = \{\mathsf{true}, \mathsf{false}\}$
   $[\mathsf{ind}] = \mathbb{N}$ (the set of natural numbers)[6]
   $[\sigma \to \tau] = [\sigma] \to [\tau]$ (the set of functions from $[\sigma]$ to $[\tau]$)
2. $([c_\tau])_{c_\tau \in C}$ is a family such that $[c_\tau] \in [\tau]$ for all $c_\tau \in C$

[6] Any infinite (not necessarily countable) set would do here; we only choose $\mathbb{N}$ for simplicity.

(Note that, in condition 2 above, $[\tau]$ refers to the extension described at point 1.)

Let GBI^F be the set of ground built-in constant instances c_τ with $\tau \in \mathsf{Type}^F$. We extend the family $([c_\tau])_{c_\tau \in C}$ to a family $([c_\tau])_{c_\tau \in C \cup \mathsf{GBI}^F}$, by interpreting the built-in constants as expected:

- $[\longrightarrow_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}]$ as the logical implication on $\{\mathsf{true}, \mathsf{false}\}$
- $[=_{\tau\to\tau\to\mathsf{bool}}]$ as the equality predicate in $[\tau] \to [\tau] \to \{\mathsf{true}, \mathsf{false}\}$
- $[\varepsilon_{(\tau\to\mathsf{bool})\to\tau}]$ as the following function, where, for each $f : [\tau] \to \{\mathsf{true}, \mathsf{false}\}$, we let $A_f = \{a \in [\tau] \mid f(a) = \mathsf{true}\}$:
  $[\varepsilon_{(\tau\to\mathsf{bool})\to\tau}](f) = \begin{cases} \mathsf{choice}(A_f) & \text{if } A_f \text{ is nonempty} \\ \mathsf{choice}([\tau]) & \text{otherwise} \end{cases}$
- $[\mathsf{zero}_{\mathsf{ind}}]$ as 0 and $[\mathsf{suc}_{\mathsf{ind}\to\mathsf{ind}}]$ as the successor function for $\mathbb{N}$

To summarize, given an interpretation $\mathcal{I}$, which is a pair of families

$$((\![\tau]\!])_{\tau\in T}, ([c_\tau])_{c_\tau \in C}),$$

we can always obtain an extended pair of families

$$((\![\tau]\!])_{\tau\in \mathsf{Cl}(T)}, ([c_\tau])_{c_\tau \in C \cup \mathsf{GBI}^F}).$$

Now we are ready to interpret the terms in Term^F according to $\mathcal{I}$. A valuation $\xi : \mathsf{Var}_{\mathsf{Type}^F} \to \mathsf{Set}$ is called $\mathcal{I}$-*compatible* if $\xi(x_\sigma) \in [\sigma]^{\mathcal{I}}$ for all $x_\sigma \in \mathsf{Var}_{\mathsf{GType}}$. We write $\mathsf{Comp}^{\mathcal{I}}$ for the set of compatible valuations. For each $t \in \mathsf{Term}^F$, we define a function

$$[t] : \mathsf{Comp}^{\mathcal{I}} \to [\mathsf{tpOf}(t)]$$

recursively over terms as expected:

$$\begin{aligned} &[x_\sigma](\xi) = \xi(x_\sigma) \\ &[c_\sigma](\xi) = [c_\sigma] \\ &[t_1\ t_2](\xi) = [t_1](\xi)\ ([t_2](\xi)) \\ &[\lambda x_\sigma.\ t](\xi) = \Lambda_{a\in[\sigma]}[t](\xi\langle x_\sigma \leftarrow a\rangle) \end{aligned}$$

Above, $\Lambda_{a\in[\sigma]}([t](\xi\langle x_\sigma \leftarrow a\rangle))$ denotes the function sending each $a \in [\sigma]$ to $[t](\xi\langle x_\sigma \leftarrow a\rangle)$, where $\xi\langle x_\sigma \leftarrow a\rangle$ is ξ updated with a at x_σ. Note that the recursive definition of $[t]$ is correct thanks to Lemma 9.(5). The above concepts are parametrized by a fragment F and an F-interpretation $\mathcal{I}$. If $\mathcal{I}$ or F are not clear from the context, we may write $[u]^{\mathcal{I}}$ or $[u]^{F,\mathcal{I}}$ instead of $[u]$.

Given a formula $\varphi \in \mathsf{Fmla}^F$, we say that $\mathcal{I}$ *satisfies* φ *via* ξ, written $\mathcal{I} \models_\xi \varphi$, if $[\varphi]^{\mathcal{I}}(\xi) = \mathsf{true}$. We define $\mathcal{I} \models \varphi$, read $\mathcal{I}$ *satisfies* φ or $\mathcal{I}$ *is a model of* φ, to mean that $\mathcal{I} \models_\xi \varphi$ for all valuations $\xi \in \mathsf{Comp}^{\mathcal{I}}$.

The pairs $(F, \mathcal{I})$, consisting of fragments and associated interpretations, are naturally ordered: Given fragments $F_1 = (T_1, C_1)$ and $F_2 = (T_2, C_2)$, F_1-interpretation $\mathcal{I}_1$ and F_2-interpretation $\mathcal{I}_2$, we define $(F_1, \mathcal{I}_1) \leq (F_2, \mathcal{I}_2)$ to mean $T_1 \subseteq T_2$, $C_1 \subseteq C_2$ and $[u]^{\mathcal{I}_1} = [u]^{\mathcal{I}_2}$ for all $u \in T_1 \cup C_1$. The *total fragment* $\top = (\mathsf{GType}^\bullet, \mathsf{GInst}^\bullet)$ is the top element in this order. Note that $\mathsf{Type}^\top = \mathsf{GType}$ and $\mathsf{Term}^\top = \mathsf{GTerm}$.

Lemma 10 If $(F_1, \mathcal{I}_1) \leq (F_2, \mathcal{I}_2)$, then the following hold:

(1) $\mathsf{Type}^{F_1} \subseteq \mathsf{Type}^{F_2}$
(2) $\mathsf{Term}^{F_1} \subseteq \mathsf{Term}^{F_2}$
(3) $[\tau]^{F_1,\mathcal{I}_1} = [\tau]^{F_2,\mathcal{I}_2}$ for all $\tau \in \mathsf{Type}^{F_1}$
(4) $[t]^{F_1,\mathcal{I}_1} = [t]^{F_2,\mathcal{I}_2}$ for all $t \in \mathsf{Term}^{F_1}$

Proof. (1),(2): Immediate from definitions.
(3),(4): By straightforward inductions on τ and t. □

So far, $\mathcal{I} \models \varphi$, the notion of $\mathcal{I}$ being a model of φ, was only defined for formulas φ that belong to Fmla^F, in particular, that are ground formulas. As promised, we extend this to polymorphic formulas by quantifying universally over all ground type substitutions. However, we shall only be interested in the case where $\mathcal{I}$ is a $\top$-interpretation.

Let $\mathcal{I}$ be a $\top$-interpretation. Given a (possibly polymorphic) formula φ, a ground type substitution θ and a valuation $\xi \in \mathsf{Comp}^{\mathcal{I}}$, we say that $\mathcal{I}$ *satisfies* φ *via* θ *and* ξ, written $\mathcal{I} \models_{\theta,\xi} \varphi$, if $\mathcal{I} \models_{\xi} \theta(\varphi)$ (i.e., if $[\theta(\varphi)]^{\mathcal{I}}(\xi) = \mathsf{true}$). This is extended to sets Γ of formulas in the expected fashion: $\mathcal{I} \models_{\theta,\xi} \Gamma$ if $\mathcal{I} \models_{\theta,\xi} \varphi$ for all $\varphi \in \Gamma$. Furthermore, we define $\mathcal{I} \models_{\theta,\xi} (\Gamma, \varphi)$ to mean that $\mathcal{I} \models_{\theta,\xi} \Gamma$ implies $\mathcal{I} \models_{\theta,\xi} \varphi$.

We define $\mathcal{I} \models (\Gamma, \varphi)$, read $\mathcal{I}$ *satisfies* (Γ, φ) or $\mathcal{I}$ *is a model of* (Γ, φ), to mean that $\mathcal{I} \models_{\theta,\xi} (\Gamma, \varphi)$ for all ground type substitution θ and valuations $\xi \in \mathsf{Comp}^{\mathcal{I}}$. When $\Gamma = \emptyset$, we write $\mathcal{I} \models \varphi$ instead of $\mathcal{I} \models (\Gamma, \varphi)$. We define $\mathcal{I} \models D$ as $\mathcal{I} \models \varphi$ for all $\varphi \in D$. Finally, we define $\mathcal{I} \models (D; \Gamma, \varphi)$ to mean that $\mathcal{I} \models D$ implies $\mathcal{I} \models (\Gamma, \varphi)$.

Note that the definitions leading to $\mathcal{I} \models (D; \Gamma, \varphi)$ mirror the structure of the HOL deduction relation, $D; \Gamma \vdash \varphi$. Namely, Γ and φ are connected through their common type variables and free term variables (as seen in the definition of $\mathcal{I} \models (\Gamma, \varphi)$ which is based on $\mathcal{I} \models_{\theta,\xi} (\Gamma, \varphi)$), whereas D is disconnected from Γ and φ w.r.t. variables (as seen in the definition of $\mathcal{I} \models (D; \Gamma, \varphi)$).

4.5 Soundness

We shall only care about the soundness of the HOL rules w.r.t. the total fragment $\top$ (which corresponds to what is being considered in traditional HOL in the first place).

We say that a deduction rule

$$\frac{D; \Gamma_1 \vdash \varphi_1 \qquad \ldots \qquad D; \Gamma_n \vdash \varphi_n}{D; \Gamma \vdash \varphi}$$

is *sound* (w.r.t. our ground semantics) if, for every $\top$-interpretation $\mathcal{I}$, we have that $\mathcal{I} \models (D; \Gamma, \varphi)$ holds whenever $\mathcal{I} \models (D; \Gamma_i, \varphi_i)$ holds for all $i \in \{1, \ldots, n\}$.

In order to prove the soundness of the HOL rules, we need some basic facts about substitutions, valuations and free variables. These facts follow fairly standard patterns, showing how substitutions can be composed with substitutions and with term interpretations and stating that the substitution and interpretation of a term only depend on its free variables.

Lemma 11 Let θ, θ' be ground substitutions, $\mathcal{I}$ a $\top$-interpretation, ξ and ξ' $\mathcal{I}$-compatible valuations, σ type, τ ground type, s, t terms such that $\mathsf{tpOf}(s) = \sigma$, and s', t' ground terms such that $\mathsf{tpOf}(s') = \tau$. Then the following hold:[7]
(1) $\theta\ (t[\sigma/\alpha]) = \theta\langle\alpha \leftarrow \theta(\sigma)\rangle\ (t)$
(2) $\theta\ (t[s/x_\sigma]) = \theta(t)\ [\theta(s)/x_{\theta(\sigma)}]$
(3) $[t'[s'/x_\tau]]^{\mathcal{I}}(\xi) = [t']^{\mathcal{I}}(\xi\langle x_\tau \leftarrow [s']^{\mathcal{I}}(\xi)\rangle)$
(4) If θ and θ' coincide on $\mathsf{TV}(t)$, then $\theta(t) = \theta'(t)$
(5) If ξ and ξ' coincide on $\mathsf{FV}(t')$, then $[t']^{\mathcal{I}}(\xi) = [t']^{\mathcal{I}}(\xi')$

[7] Recall that $f\langle a \leftarrow b\rangle$ denotes function update: $f\langle a \leftarrow b\rangle$ is the function that acts like f except that it sends a to b.

Proof. All points follow by straightforward induction on t or t'. □

Thanks to Lemma 11(2), if t' is a ground term that is also closed, then $[t']^{\mathcal{I}}(\xi)$ does not really depend on ξ, and therefore we can write $[t']^{\mathcal{I}}$ instead of $[t']^{\mathcal{I}}(\xi)$. Now we are ready for the soundness proof.

Lemma 12 The following hold:
(1) The deduction rules of Isabelle/HOL are sound.
(2) If $D \vdash \varphi$ and $\mathcal{I}$ is a $\top$-interpretation such that $\mathcal{I} \models D$, then $\mathcal{I} \models \varphi$.

Proof. (1): The soundness of all the rules follows routinely from the definition of satisfaction. The only slightly interesting rules are (EXT), (T-INST) and (INST), which need Lemma 11.

(EXT): We assume $x_\sigma \notin \mathsf{FV}(\Gamma)$ (*) and $\mathcal{I} \models (D; \Gamma, y_{\sigma\to\tau}\ x_\sigma = z_{\sigma\to\tau}\ x_\sigma)$ (**), and need to prove $\mathcal{I} \models (D; \Gamma, y_{\sigma\to\tau} = z_{\sigma\to\tau})$; i.e., we assume $\mathcal{I} \models D$ (***) and need to prove $\mathcal{I} \models (\Gamma, y_{\sigma\to\tau} = z_{\sigma\to\tau})$.

To this end, let θ be a ground type substitution and $\xi \in \mathsf{Comp}^{\mathcal{I}}$; we assume $\mathcal{I} \models_{\theta,\xi} \Gamma$ (****) and need to prove $\mathcal{I} \models_{\theta,\xi} y_{\sigma\to\tau} = z_{\sigma\to\tau}$, i.e., $\xi(y_{\theta(\sigma)\to\theta(\tau)}) = \xi(z_{\theta(\sigma)\to\theta(\tau)})$. It would suffice to fix an arbitrary $a \in [\theta(\sigma)]^{\mathcal{I}}$ and prove that $\xi(y_{\theta(\sigma)\to\theta(\tau)})(a) = \xi(z_{\theta(\sigma)\to\theta(\tau)})(a)$.

In order to prove this last fact, let $\xi' = \xi\langle x_\sigma \leftarrow a\rangle$. From (**) and (***), we have $\mathcal{I} \models (\Gamma, y_{\sigma\to\tau}\ x_\sigma = z_{\sigma\to\tau}\ x_\sigma)$, hence $\mathcal{I} \models_{\theta,\xi'} (\Gamma, y_{\sigma\to\tau}\ x_\sigma = z_{\sigma\to\tau}\ x_\sigma)$ (*****). From (*), (****) and Lemma 11(4), we obtain $\mathcal{I} \models_{\theta,\xi'} \Gamma$. With (*****), we obtain $[y_{\theta(\sigma)\to\theta(\tau)}\ x_{\theta(\sigma)}]^{\mathcal{I}}(\xi') = [z_{\theta(\sigma)\to\theta(\tau)}\ x_{\theta(\sigma)}]^{\mathcal{I}}(\xi')$, i.e., $\xi'(y_{\theta(\sigma)\to\theta(\tau)})\ (\xi'(x_{\theta(\sigma)})) = \xi'(y_{\theta(\sigma)\to\theta(\tau)})\ (\xi'(x_{\theta(\sigma)}))$, i.e., $\xi(y_{\theta(\sigma)\to\theta(\tau)})(a) = \xi(z_{\theta(\sigma)\to\theta(\tau)})(a)$, as desired.

(T-INST): We assume $\alpha \notin \mathsf{FV}(\Gamma)$ (*) and $\mathcal{I} \models (D; \Gamma, \varphi)$ (**) and need to prove $\mathcal{I} \models (D; \Gamma, \varphi[\sigma/\alpha])$; i.e., assume $\mathcal{I} \models D$ (***) and need to prove $\mathcal{I} \models (\Gamma, \varphi[\sigma/\alpha])$. For this, let θ be a ground type substitution and $\xi \in \mathsf{Comp}^{\mathcal{I}}$; we assume $\mathcal{I} \models_{\theta,\xi} \Gamma$ (****) and need to prove $\mathcal{I} \models_{\theta,\xi} \varphi[\sigma/\alpha]$, i.e., $\mathcal{I} \models_{\xi} \theta(\varphi[\sigma/\alpha])$. By Lemma 11(1), this amounts to $\mathcal{I} \models_{\xi} (\theta\langle\alpha \leftarrow \theta(\sigma)\rangle)\ (\varphi)$, i.e., $\mathcal{I} \models_{\theta\langle\alpha\leftarrow\theta(\sigma)\rangle,\xi} \varphi$.

To prove this last fact, we note that, thanks to (*), we have that (****) is equivalent to $\mathcal{I} \models_{\theta\langle\alpha\leftarrow\theta(\sigma)\rangle,\xi} \Gamma$ (*****). (This is because, for all $\chi \in \Gamma$, Lemma 11(4) and (*) imply $\theta(\chi) = \theta\langle\alpha \leftarrow \ldots\rangle\ (\chi)$.) Finally, (**), (***) and (*****) imply $\mathcal{I} \models_{\theta\langle\alpha\leftarrow\theta(\sigma)\rangle,\xi} \varphi$, as desired.

(INST): We assume $x_\sigma \notin \mathsf{FV}(\Gamma)$ (*) and $\mathcal{I} \models (D; \Gamma, \varphi)$ (**) and need to prove that $\mathcal{I} \models (D; \Gamma, \varphi[t/x_\sigma])$; i.e., assume $\mathcal{I} \models D$ (***) and need to prove $\mathcal{I} \models (\Gamma, \varphi[t/x_\sigma])$. For this, let θ be a ground type substitution and $\xi \in \mathsf{Comp}^{\mathcal{I}}$; we assume $\mathcal{I} \models_{\theta,\xi} \Gamma$ (****) and need to prove $\mathcal{I} \models_{\theta,\xi} \varphi[t/x_\sigma]$, i.e., $\mathcal{I} \models_{\xi} \theta(\varphi[t/x_\sigma])$. By Lemma 11(2), this amounts to $\mathcal{I} \models_{\xi} \theta(\varphi)\ [\theta(t)/x_{\theta(\sigma)}]$; by Lemma 11(3), the latter amounts to $\mathcal{I} \models_{\xi\langle x_{\theta(\sigma)}\leftarrow[\theta(t)](\xi)\rangle} \theta(\varphi)$, i.e., $\mathcal{I} \models_{\theta,\xi\langle x_{\theta(\sigma)}\leftarrow[\theta(t)](\xi)\rangle} \varphi$.

To prove this last fact, we note that, thanks to (*), we have that (****) is equivalent to $\mathcal{I} \models_{\theta,\xi\langle x_{\theta(\sigma)}\leftarrow[\theta(t)](\xi)\rangle} \Gamma$ (*****). (This is because, for all $\chi \in \Gamma$, (*) implies $x_{\theta(\sigma)} \notin \mathsf{FV}(\theta(\chi))$, which by Lemma 11(5) further implies $[\theta(\chi)](\xi) = [\theta(\chi)](\xi\langle x_{\theta(\sigma)} \leftarrow \ldots\rangle)$.) Finally, (**), (***) and (*****) imply $\mathcal{I} \models_{\theta,\xi\langle x_\sigma\leftarrow[\theta(t)](\xi)\rangle} \varphi$, as desired.

(2): Follows immediately from (1). □

Theorem 13 Let D be a theory that has a total-fragment model, i.e., there exists a $\top$-interpretation $\mathcal{I}$ such that $\mathcal{I} \models D$. Then D is consistent.

Proof. Immediate from Lemma 12(2): Assume by absurd that D is inconsistent, in that $D \vdash \mathsf{False}$. Since $\mathcal{I} \models D$, by Lemma 12(2) it follows that $\mathcal{I} \models \mathsf{False}$, meaning $[\mathsf{False}]^{\mathcal{I}} = \mathsf{true}$. But this is a contradiction, since by definition $[\mathsf{False}]^{\mathcal{I}} = \mathsf{false} \neq \mathsf{true}$. □

4.6 The Model Construction

The only missing piece in the proof of consistency is the existence of a total-fragment model for a definitional theory D. For this, we need some preparations.

Lemma 14 Assume $c_\sigma \in \mathsf{CInst}^\bullet$ and $\tau \in \mathsf{types}^\bullet(\sigma)$. Then $c_\sigma \rightsquigarrow^\downarrow \tau$.

Proof. Let $\sigma' = \mathsf{tpOf}(c)$. Then $\sigma \leq \sigma'$, hence we obtain θ such that $\sigma = \theta(\sigma')$. By Lemma 4(1), from $\tau \in \mathsf{types}^\bullet(\theta(\sigma'))$ we obtain $\tau' \in \mathsf{types}^\bullet(\sigma')$ such that $\tau = \theta(\tau')$. Now, from the definition of $\rightsquigarrow$ we have $c_{\sigma'} \rightsquigarrow \tau'$, hence $c_{\theta(\sigma')} \rightsquigarrow^\downarrow \theta(\tau')$. But this means $c_\sigma \rightsquigarrow^\downarrow \tau$, as desired. □

Given $u \in \mathsf{Type}^\bullet \cup \mathsf{Const}^\bullet$, we define F_u, the *fragment generated by u (via D)*, as follows:

- $V_u = \{v \mid u \rightsquigarrow^{\downarrow +} v\}$
- $T_u = V_u \cap \mathsf{Type}$
- $C_u = V_u \cap \mathsf{CInst}$
- $F_u = (T_u, C_u)$

Lemma 15 (1) F_u is indeed a fragment.
(2) If $w \equiv s$ is in D and $u = \theta(w)$, then $\theta(s) \in \mathsf{Term}^{F_u}$.

Proof. (1): Let $c_\sigma \in C_u$. We need to prove $\sigma \in \mathsf{Cl}(T_u)$. By Lemma 8(1), it suffices that $\mathsf{types}^\bullet(\sigma) \subseteq T_u$. So let $\tau \in \mathsf{types}^\bullet(\sigma)$. By Lemma 14, we have $c_\sigma \rightsquigarrow^\downarrow \tau$. And since $u \rightsquigarrow^{\downarrow +} c_\sigma$, we obtain $u \rightsquigarrow^{\downarrow +} \tau$. But this means $\tau \in T_u$, as desired.
(2): Let $\tau \in \mathsf{types}^\bullet(\theta(s))$. By Lemma 4(2), we obtain $\tau' \in \mathsf{types}^\bullet(s)$ such that $\tau = \theta(\tau')$. From the definition of $\rightsquigarrow$, we have $w \rightsquigarrow \tau'$, hence $\theta(w) \rightsquigarrow^\downarrow \theta(\tau')$, i.e., $u \rightsquigarrow^\downarrow \tau$. We thus obtained $\mathsf{types}^\bullet(\theta(s)) \subseteq T_u$.

Similarly, but using Lemma 4(3) instead of Lemma 4(2), we obtain $\mathsf{consts}^\bullet(\theta(s)) \subseteq C_u$. Together with $\mathsf{types}^\bullet(\theta(s)) \subseteq T_u$, this means $\theta(s) \in \mathsf{Term}^{F_u}$, as desired. □

Now we are ready to prove the aforementioned missing piece:

Theorem 16 Assume D is a definitional theory. Then it has a total-fragment model, i.e., there exists a $\top$-interpretation $\mathcal{I}$ such that $\mathcal{I} \models D$.

Proof. Since $\rightsquigarrow^\downarrow$ is terminating, $\rightsquigarrow^{\downarrow +}$ is also terminating. By well-founded recursion and induction on $\rightsquigarrow^{\downarrow +}$, for each $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, we define $[u]$ simultaneously with proving that the following hold:

A_u: $\mathcal{I}_u = (([v])_{v \in T_u}, ([v])_{v \in C_u})$ is an F_u-interpretation.[8]
B_u: If $u \in \mathsf{GType}^\bullet$ (i.e., u is a type) then $[u] \neq \emptyset$; and if u has the form c_τ (i.e., u is a constant) then $[u] \in [\tau]$.

[8] Note that, by Lemma 15(1), we have that $F_u = (T_u, C_u)$ is a fragment.

We assume that, for all $v \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$ such that $u \rightsquigarrow^{\downarrow +} v$ holds, $[v]$ has been defined and A_v and B_v hold.

We first show that A_u holds. For this, we first assume $\tau \in T_u$ and need to prove $[\tau] \neq \emptyset$; but this is precisely B_{c_τ}, which holds by the induction hypothesis, since $u \rightsquigarrow^{\downarrow +} \tau$. Now, we assume $c_\tau \in C_u$ and need to prove $[c_\tau] \in [\tau]$; but this is precisely B_{c_τ}, which holds by the induction hypothesis, since $u \rightsquigarrow^{\downarrow +} c_\tau$.

Next, we define $[u]$. We say that a definition $w \equiv s$ matches u if there exists a ground type substitution θ with $u = \theta(w)$. We distinguish the following cases:

1. There exists no definition in D that matches u. Here we have two subcases:
   - $u \in \mathsf{GType}^\bullet$. Then we define $[u] = \{*\}$.
   - $u \in \mathsf{GCInst}^\bullet$. Say u has the form c_σ. By Lemma 14, we have that $c_\sigma \rightsquigarrow^{\downarrow} \tau$ for all $\tau \in \mathsf{types}^\bullet(\sigma)$. Hence $\mathsf{types}^\bullet(\sigma) \subseteq T_u$, which means $\sigma \in \mathsf{Type}^{F_u}$. Therefore, using also A_u, we can speak of the value $[\sigma]^{F_u,\mathcal{I}_u}$ (obtained from the F_u-interpretation $\mathcal{I}_u$). We define $[u] = \mathsf{choice}([\sigma]^{F_u,\mathcal{I}_u})$.
2. There exists a definition $w \equiv s$ in D that matches u. Then let θ be such that $u = \theta(w)$, and let $t = \theta(s)$. By Lemma 15(2) we have $t \in \mathsf{Term}^{F_u}$. Therefore, using also A_u, we can speak of the value $[t]^{F_u,\mathcal{I}_u}$. We have two subcases:
   - $u \in \mathsf{GCInst}^\bullet$. Then we define $[u] = [t]^{F_u,\mathcal{I}_u}$.
   - $u \in \mathsf{GType}^\bullet$. Then the type of t has the form $\sigma \to \mathsf{bool}$. And since $\mathsf{types}^\bullet(\sigma) \subseteq \mathsf{types}^\bullet(t) \subseteq T_u$, by Lemma 8(1) we obtain $\sigma \in \mathsf{Cl}(T_u)$, i.e., $\sigma \in \mathsf{Type}^{F_u}$. Therefore, using also A_u, we can speak of the value $[\sigma]^{F_u,\mathcal{I}_u}$. We have two subsubcases:
     - $[\exists x_\sigma.\ t\ x]^{F_u,\mathcal{I}_u} = \mathsf{false}$. Then we define $[u] = \{*\}$.
     - $[\exists x_\sigma.\ t\ x]^{F_u,\mathcal{I}_u} = \mathsf{true}$. Then we define $[u] = \{a \in [\sigma]^{F_u,\mathcal{I}_u} \mid [t](a) = \mathsf{true}\}$.

Next, we prove B_u. For this, first assume $u \in \mathsf{GType}^\bullet$; then $[u] \neq \emptyset$ follows immediately from the definition of $[u]$: in case 1 it is $\{*\}$ and in case 2 it is either $\{*\}$ or a nonempty set. Now, assume u has the form c_τ. If u matches no definition in D, then $[u] \in [\tau]$ holds by the definition of of the choice operator. On the other hand, if u matches a definition in D as in the above case 2, we have $[u] = [t]^{F_u,\mathcal{I}_u}$, and the desired fact follows from $[t]^{F_u,\mathcal{I}_u} \in [\mathsf{tpOf}(t)]^{F_u,\mathcal{I}_u}$ and $\mathsf{tpOf}(t) = \tau$. This concludes the definition of $[u]$ and the proof that A_u and B_u hold.

Next, we note that $\mathcal{I} = (([u])_{u \in \mathsf{GType}^\bullet}, ([u])_{u \in \mathsf{GCInst}^\bullet})$ is a $\top$-interpretation. Indeed, assuming $c_\tau \in \mathsf{GCInst}^\bullet$, we have that $[c_\tau] \in [\tau]$ holds thanks to B_{c_τ}.

It remains to show that $\mathcal{I} \models D$. To this end, let $w \equiv s$ be in D and let θ' be a ground type substitution. We need to show $\mathcal{I} \models \theta'(w \equiv s)$, i.e., $\mathcal{I} \models \theta'(w) \equiv \theta'(s)$.

Let $u = \theta'(w)$; then u matches $w \equiv s$, and by orthogonality this is the only definition in D that it matches. So the definition of $[u]$ proceeds with case 2 above, using $w \equiv s$—let θ be the ground type substitution considered there. Since $\theta'(w) = \theta(w)$, it follows that θ' and θ coincide on $\mathsf{TV}(w)$, and hence on $\mathsf{TV}(s)$ (since $\mathsf{TV}(s) \subseteq \mathsf{TV}(w)$ holds for any definition $s \equiv w$ in D); hence, by Lemma 11(4), $\theta'(s) = \theta(s)$.

So we need to prove $\mathcal{I} \models u \equiv t$, i.e., $[u \equiv t]^{\top,\mathcal{I}} = \mathsf{true}$, where $u = \theta(w)$ and $t = \theta(s)$. From case 2 of the definition of $[u]$, we can see that $[u]$ was chosen precisely so that $[u \equiv t]^{F_u,\mathcal{I}_u} = \mathsf{true}$. (In particular, for the case when u is a type and $[\exists x_\sigma.\ t\ x] = \mathsf{false}$, $[u \equiv t]^{F_u,\mathcal{I}_u} = \mathsf{true}$ holds thanks to the Makarius Wenzel trick, since the implication premise is false.) Finally, since $(F_u, \mathcal{I}_u) \leq (\top, \mathcal{I})$, by Lemma 10(3,4) we obtain $[u \equiv t]^{\top,\mathcal{I}} = \mathsf{true}$, as desired. □

5 Deciding Definitionality

We proved that every definitional theory is consistent. From the implementation perspective, we can ask ourselves how difficult it is to check that the given set of axioms D forms a definitional theory. We can check that D consists of definitions and is orthogonal by simple polynomial algorithms. On the other hand, Obua [32] showed that a dependency relation generated by overloaded definitions can encode the Post correspondence problem and therefore termination of such a relation is not even a semi-decidable problem.

Kunčar [21] takes the following approach: Let us impose a syntactic restriction, called *compositionality*, on accepted overloaded definitions which makes the termination of the dependency relation decidable while still permitting all use cases of overloading in Isabelle. Namely, let $\rightsquigarrow\!\!\!\!\rightarrow$ be the substitutive and transitive closure of the dependency relation $\rightsquigarrow$ (which is in fact equal to $\rightsquigarrow^{\downarrow+}$). Then D is called *composable* if for all u, u' that are left-hand sides of some definitions from D and for all v such that $u \rightsquigarrow\!\!\!\!\rightarrow v$, it holds that either $u' \leq v$, or $v \leq u'$, or $u' \# v$. Under composability, termination of $\rightsquigarrow\!\!\!\!\rightarrow$ is equivalent to acyclicity of $\rightsquigarrow$, which is a decidable condition.

Composability reduces the search space when we are looking for the cycle—it tells us that there exist three cases on how to extend a path (to possibly close a cycle): in two cases we can still (easily) extend the path ($v \leq u'$ or $u' \leq v$) and in one case we cannot ($v \# u'$). The fourth case (v and u' have a non-trivial common instance; formally $u' \not\leq v$ and $v \not\leq u'$ and there exists w such that $w \leq u'$, $w \leq v$), which complicates the extension of the path, is ruled out by composability. More about composability can be found in [21].

Theorem 17 The property "D is composable and is a definitional theory" is decidable.[9]

Proof. The paper [21] presents a quadratic algorithm (in the size of $\rightsquigarrow$), CHECK, that checks that D consists of definitions, is orthogonal and composable, and that its dependency relation $\rightsquigarrow\!\!\!\!\rightarrow$ terminates. The correctness proof is relatively general and works for any $\rightsquigarrow\!\!\!\!\rightarrow$: $\mathcal{U}_\Sigma \to \mathcal{U}_\Sigma \to \mathsf{bool}$ on a set $\mathcal{U}_\Sigma$ endowed with a certain structure—namely, three functions $= : \mathcal{U}_\Sigma \to \mathcal{U}_\Sigma \to \mathsf{bool}$, $\mathsf{App} : (\mathsf{Type} \to \mathsf{Type}) \to \mathcal{U}_\Sigma \to \mathcal{U}_\Sigma$ and $\mathsf{size} : \mathcal{U}_\Sigma \to \mathbb{N}$, indicating how to compare for equality, type-substitute and measure the elements of $\mathcal{U}_\Sigma$. In this paper, we set $\Sigma = (K, \mathsf{arOf}, C, \mathsf{tpOf})$ and $\mathcal{U}_\Sigma = \mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$. The definition of $=$, App and size is then straightforward: two elements of $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$ are equal iff they are both constant instances and they are equal or they are both types and they are equal; $\mathsf{App}\,\rho\,\tau = \rho(\tau)$ and $\mathsf{App}\,\rho\,c_\tau = c_{\rho(\tau)}$; finally, $\mathsf{size}(\tau)$ counts the number of type constructors in τ and $\mathsf{size}(c_\tau) = \mathsf{size}(\tau)$. Notice that $\rightsquigarrow\!\!\!\!\rightarrow = \rightsquigarrow^{\downarrow+}$ terminates iff $\rightsquigarrow^{\downarrow}$ terminates. Thus, CHECK decides whether D is composable and is a definitional theory. □

For efficiency reasons, we optimize the size of the relation that the quadratic algorithm works with. Let $\rightsquigarrow_1$ be the relation defined like $\rightsquigarrow$, but only retaining clause 1 in the definition of $\rightsquigarrow$. Since $\rightsquigarrow^{\downarrow}$ is terminating iff $\rightsquigarrow_1^{\downarrow}$ is terminating, it suffices to check termination of the latter.

6 Conclusion

We have provided a solution to the consistency problem for Isabelle/HOL's logic, namely, polymorphic HOL with ad hoc overloading. The solution has been incorporated in Isa-

[9] Note that the property "D is a definitional theory" is not decidable; it is the conjunction with the composability property that ensures decidability.

belle2016. Independently of Isabelle/HOL, our results show that Gordon-style type definitions and ad hoc overloading can be soundly combined and naturally interpreted semantically.

Acknowledgments. We thank the reviewers for their useful comments and suggestions, and for catching some errors in the proofs. The anonymous ITP 2015 and Makarius Wenzel also made useful comments on the conference version of the paper. We thank Tobias Nipkow, Larry Paulson and Makarius Wenzel for inspiring discussions. This paper was partially supported by the DFG grant Ni 491/13-3 and by the EPSRC grant EP/N019547/1.

References

1. A Consistent Foundation for Isabelle/HOL - A Correction Patch. URL `http://www21.in.tum.de/~kuncar/documents/patch.html`
2. The HOL4 Theorem Prover. URL `http://hol.sourceforge.net/`
3. Adams, M.: Introducing HOL Zero - (Extended Abstract). In: K. Fukuda, J. van der Hoeven, M. Joswig, N. Takayama (eds.) ICMS 2010, *LNCS*, vol. 6327, pp. 142–143. Springer (2010)
4. Anand, A., Rahli, V.: Towards a Formally Verified Proof Assistant. In: G. Klein, R. Gamboa (eds.) ITP 2014, *LNCS*, vol. 8558, pp. 27–44. Springer (2014)
5. Arthan, R.D.: Some Mathematical Case Studies in ProofPower–HOL. In: K. Slind (ed.) TPHOLs 2004 (Emerging Trends), School of Computing, pp. 1–16. University of Utah (2010)
6. Barras, B.: Coq en Coq. Tech. Rep. 3026, INRIA (1996)
7. Barras, B.: Sets in Coq, Coq in Sets. J. Formalized Reasoning **3**(1), 29–48 (2010)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
9. Blanchette, J.C., Popescu, A., Traytel, D.: Foundational extensible corecursion: a proof assistant perspective. In: K. Fisher, J.H. Reppy (eds.) ICFP 2015, pp. 192–204. ACM (2015)
10. Bove, A., Dybjer, P., Norell, U.: A Brief Overview of Agda - A Functional Language with Dependent Types. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) TPHOLs 2009, *LNCS*, vol. 5674, pp. 73–78. Springer (2009)
11. Dénès, M.: [Coq-Club] Propositional extensionality is inconsistent in Coq. Archived at `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html`
12. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)
13. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a Nutshell. J. Formalized Reasoning **3**(2), 153–245 (2010)
14. Haftmann, F., Wenzel, M.: Constructive Type Classes in Isabelle. In: T. Altenkirch, C. McBride (eds.) TYPES 2006, *LNCS*, vol. 4502, pp. 160–174. Springer (2006)
15. Harrison, J.: HOL Light: A Tutorial Introduction. In: M. K. Srivas, A.J. Camilleri (eds.) FMCAD '96, *LNCS*, vol. 1166, pp. 265–269. Springer (1996)
16. Harrison, J.: Towards Self-verification of HOL Light. In: U. Furbach, N. Shankar (eds.) IJCAR 2006, *LNCS*, vol. 4130, pp. 177–191. Springer (2006)
17. Hölzl, J., Immler, F., Huffman, B.: Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) ITP 2013, *LNCS*, vol. 7998, pp. 279–294. Springer (2013)
18. Huffman, B., Urban, C.: A New Foundation for Nominal Isabelle. In: M. Kaufmann, L.C. Paulson (eds.) ITP 2010, *LNCS*, vol. 6172, pp. 35–50. Springer (2010)
19. Krauss, A., Schropp, A.: A Mechanized Translation from Higher-Order Logic to Set Theory. In: M. Kaufmann, L.C. Paulson (eds.) ITP 2010, *LNCS*, vol. 6172, pp. 323–338. Springer (2010)
20. Kumar, R., Arthan, R., Myreen, M.O., Owens, S.: HOL with Definitions: Semantics, Soundness, and a Verified Implementation. In: G. Klein, R. Gamboa (eds.) ITP 2014, *LNCS*, vol. 8558, pp. 308–324. Springer (2014)
21. Kunčar, O.: Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants. In: X. Leroy, A. Tiu (eds.) CPP 2015, pp. 85–94. ACM (2015)
22. Kunčar, O., Popescu, A.: A Consistent Foundation for Isabelle/HOL. In: C. Urban, X. Zhang (eds.) ITP 2015, *LNCS*, vol. 9236, pp. 234–252. Springer (2015)
23. Kunčar, O., Popescu, A.: Comprehending Isabelle/HOL's consistency. In: H. Yang (ed.) ESOP 2017, *LNCS*, vol. 10201, pp. 724–749. Springer (2017)

24. Kunčar, O., Popescu, A.: Safety and conservativity of definitions in hol and isabelle/hol (2018). Conditionally accepted at POPL 2018. Draft available at `http://andreipopescu.uk/pdf/conserv_HOL_IsabelleHOL.pdf`
25. Leino, K.R.M., Moskal, M.: Co-induction Simply - Automatic Co-inductive Proofs in a Program Verifier. In: C.B. Jones, P. Pihlajasaari, J. Sun (eds.) FM 2014, *LNCS*, vol. 8442, pp. 382–398. Springer (2014)
26. Lochbihler, A.: Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) ITP 2013, *LNCS*, vol. 7998, pp. 116–132. Springer (2013)
27. McBride, C., et al.: [HoTT] Newbie questions about homotopy theory and advantage of UF/Coq. Archived at `http://article.gmane.org/gmane.comp.lang.agda/6106`
28. Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF= HOL+LCF. J. Funct. Program. **9**(2), 191–223 (1999)
29. Myreen, M.O., Davis, J.: The Reflective Milawa Theorem Prover Is Sound - (Down to the Machine Code That Runs It). In: G. Klein, R. Gamboa (eds.) ITP 2014, *LNCS*, vol. 8558, pp. 421–436. Springer (2014)
30. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)
31. Nipkow, T., Snelting, G.: Type Classes and Overloading Resolution via Order-Sorted Unification. In: J. Hughes (ed.) Functional Programming Languages and Computer Architecture, *LNCS*, vol. 523, pp. 1–14. Springer (1991)
32. Obua, S.: Checking Conservativity of Overloaded Definitions in Higher-Order Logic. In: F. Pfenning (ed.) RTA 2006, *LNCS*, vol. 4098, pp. 212–226. Springer (2006)
33. Pitts, A.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, chap. The HOL Logic, pp. 191–232. In: Gordon and Melham [12] (1993)
34. Shankar, N., Owre, S., Rushby, J.M.: PVS Tutorial. Computer Science Laboratory, SRI International (1993)
35. Sozeau, M., Oury, N.: First-Class Type Classes. In: O.A. Mohamed, C.A. Muñoz, S. Tahar (eds.) TPHOLs 2008, *LNCS*, vol. 5170, pp. 278–293. Springer (2008)
36. Urban, C.: Nominal Techniques in Isabelle/HOL. J. Autom. Reasoning **40**(4), 327–356 (2008)
37. Wadler, P., Blott, S.: How to Make Ad-hoc Polymorphism Less Ad Hoc. In: POPL '89, pp. 60–76. ACM (1989)
38. Wenzel, M.: Type Classes and Overloading in Higher-Order Logic. In: E.L. Gunter, A.P. Felty (eds.) TPHOLs '97, *LNCS*, vol. 1275, pp. 307–322. Springer (1997)
39. Wenzel, M.: The Isabelle/Isar Reference Manual (2016). Available at `http://isabelle.in.tum.de/doc/isar-ref.pdf`