Comprehending Isabelle/HOL's Consistency

Ondřej Kunčar[1] and Andrei Popescu[2,3]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Department of Computer Science, Middlesex University London, UK
[3] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania

Abstract. The proof assistant Isabelle/HOL is based on an extension of Higher-Order Logic (HOL) with ad hoc overloading of constants. It turns out that the interaction between the standard HOL type definitions and the Isabelle-specific ad hoc overloading is problematic for the logical consistency. In previous work, we have argued that standard HOL semantics is no longer appropriate for capturing this interaction, and have proved consistency using a nonstandard semantics. The use of an exotic semantics makes that proof hard to digest by the community. In this paper, we prove consistency by proof-theoretic means—following the healthy intuition of definitions as abbreviations, realized in HOLC, a logic that augments HOL with comprehension types. We hope that our new proof settles the Isabelle/HOL consistency problem once and for all. In addition, HOLC offers a framework for justifying the consistency of new deduction schemas that address practical user needs.

1 Introduction

Isabelle/HOL [35, 36] is a popular proof assistant, with hundreds of users world-wide in both academia and industry. It is being used in major verification projects, such as the seL4 operating system kernel [24]. In addition, Isabelle/HOL is a framework for certified programming: functional programming (including lazy (co)programming [9]) is supported natively and imperative programming is supported via a monadic extension [10]. Programs can be written and verified in Isabelle/HOL, and efficient code for them (in Haskell, Standard ML, OCaml and Scala) can be produced using a code generator [19]. This certified programming methodology has yielded a wide range of verified software systems, from a Java compiler [32] to an LTL model checker [14] to a conference management system [23]. The formal guarantees of all such systems, as well as those considered by some formal certification agencies [21], are based on one major assumption: the *correctness/consistency of Isabelle/HOL's inference engine*.

Keeping the underlying logic simple, hence manifestly consistent, along with reducing all the user developments to the logic kernel, has been a major tenet of the LCF/HOL approach to formal verification, originating from Robin Milner and Mike Gordon [17]. Yet, Isabelle/HOL, one of the most successful incarnations of this approach, takes some liberties beyond the well-understood higher-order logic kernel. Namely, its definitional mechanism allows *delayed ad hoc overloading of constant definitions*—in turn, this enables Haskell-style type classes [46] on top of HOL [37].

In standard HOL, a polymorphic constant should either be *only declared* (and forever left uninterpreted), or *fully defined* at its most general type.[4] By contrast, Isabelle/HOL allows first declaring a constant, and at later times overloading it by defining some of its instances, as in the following example:[5]

```
consts 0 : α
...
definition 0 : real ≡ real_zero
...
definition 0 : α list ≡ []
```

Recursive overloading is also supported, as in:

```
definition 0 : α list ≡ [0:α]
```

In between the declaration and the instance definitions, arbitrary commands may occur, including type definitions ("typedef") and (co)datatype definitions (which are derived from typedef [7, 45]). For example, the following definition introduces a type of polynomials over an arbitrary domain α, where $\forall_\infty$ is the "for all but finitely many" quantifier:

```
typedef α poly ≡ {f : nat → α | ∀∞ n. f n = 0}
```

When 0 is defined for concrete types, such as real and α list, the library theorems about arbitrary-domain polynomials are enabled for polynomials over these concrete types.[6]

To avoid inconsistency, this overloading mechanism is regulated by syntactic checks for *orthogonality* and *termination*. Examples like the above should be allowed, whereas examples like the following encoding of Russell's paradox [28, Sect. 1] should be forbidden:

```
consts c : α
typedef T ≡ {True, c}
definition c : bool ≡ ¬ (∀(x:T) y. x = y)
```

The above would lead to a proof of false taking advantage of the circularity $T \rightsquigarrow c_{\mathsf{bool}} \rightsquigarrow T$ in the dependency relation introduced by the definitions: one first defines the type T to contain precisely one element just in case c_{bool} is True, and then defines c_{bool} to be True just in case T contains more than one element.

[4] There are other specification schemes supported by some HOL provers, allowing for more abstract (under)specification of constants—but these schemes are known to be captured or over-approximated by the standard (equational) definition scheme [5].

[5] To improve readability, in the examples we use a simplified Isabelle syntax. To run these examples in Isabelle, one must enclose in overloading blocks the overloaded definitions of constants and add the overloaded attribute to type definitions that depend on overloaded constants; in addition, one must provide nonemptiness proofs for type definitions [47, Sect. 11(3,7)]. Note also that Isabelle uses ⇒ instead of → for function types and :: instead of : for typing.

[6] Isabelle/HOL implements a type-class infrastructure allowing fine control over such instantiations. In this case, α is assumed to be of type class zero; then real, α list etc. are registered as members of zero as soon as 0 is defined for them. Polymorphic properties can also be associated to type classes, and need to be verified upon instantiation. Type classes do not require any logical extension, but are representable as predicates inside the logic— [48, Sect. 5] explains the mechanism in detail.

Because Isabelle/HOL has a large user base and is heavily relied upon, it is important that the consistency of its logic be established with a high degree of *clarity* and a high degree of *rigor*. In 2014, we started an investigation into the foundations of this logic, which has revealed a few consistency problems (including the above "paradox"). These issues have generated quite a lot of discussion in the Isabelle community, during which some philosophical disagreements and misunderstandings among the users and the developers have surfaced [1]. The technical issues have been addressed [26, 28] and are no longer exhibited in Isabelle2016.[7]

In addition to taking care of these issues, one of course needs some guarantees that similar issues are not still present in the logic. To address this, in previous work [28] we have proved that the logic is now consistent, employing a *semantic argument* in terms of a nonstandard semantics for HOL. Our original proof was somewhat sketchy and lacking in rigor—full (pen-and-paper) proofs are now included in an extended report [27]. Of course, a machine-checked proof, perhaps building on a recent formalization of HOL [15, 25], would make further valuable progress on the rigor aspect.

In this paper, we hope to improve on the *clarity* aspect and provide deeper insight into why Isabelle/HOL's logic is consistent. As mentioned, Isabelle/HOL is richer than HOL not in the rules of deduction, but in the definitional mechanism. A natural reluctance that comes to mind concerning our semantic proof of consistency is best expressed by Isabelle's creator's initial reaction to our proof idea [40]:

> It's a bit puzzling, not to say worrying, to want a set-theoretic semantics for plain definitions. The point of definitions (and the origin of the idea that they preserve consistency) is that they are abbreviations.

This paper's first contribution is a new proof of consistency for Isabelle/HOL's logic, easy to digest by the large community of "syntacticists" who (quite legitimately) wish to regard definitions as a form of abbreviations. The problem is that type definitions cannot simply be unfolded (and inlined)—a type definition is an axiom that postulates a new type and an embedding-projection pair between the new type and the original type (from where the new type is carved out by a nonempty predicate). But the syntactic intuition persists: what if we *were* allowed to unfold type definitions? As it turns out, this can be achieved in a gentle extension of HOL featuring comprehension types. This extended logic, called HOL with Comprehension (HOLC), is a syntacticist's paradise, allowing for a consistency proof along their intuition. This proof is systematically developed in Section 3. First, HOLC is introduced (Section 3.1) and shown consistent by a standard argument (Section 3.2). Then, a translation is defined from well-formed Isabelle/HOL definitions to HOLC, which is proved sound, i.e., deduction-preserving (Section 3.3). The key to establishing soundness is the use of a modified deduction system for HOL where type instantiation is restricted—this tames the inherent lack of uniformity brought about by ad hoc overloading. Finally, soundness of the translation together with consistency of HOLC ensures consistency of Isabelle/HOL.

[7] The philosophical dispute about foundations is far from having come to an end [4], and unfortunately tends to obscure what should be a well-defined mathematical problem: the consistency of the Isabelle/HOL *logical system* (which is of course not the same as the overall reliability of the Isabelle/HOL *implementation*).

As a second contribution, we use HOLC to justify some recently proposed extensions of Isabelle/HOL—namely, two new deduction schemas [29]. One enables local type definitions inside proof contexts; the other allows replacing undefined instances of overloaded constants with universally quantified variables. As we argue in [29], both extensions are useful for simplifying proof development by enabling the transition from light type-based theorems to heavier but more flexible set-based theorems. However, proving that these extensions do not introduce inconsistency is surprisingly difficult. In particular, our previously defined (consistency-justifying) semantics [28] has a blind spot on the second extension—it is only from the viewpoint of HOLC that the consistency of both extensions is manifest (Section 4).

More details on our constructions and proofs can be found in a technical report made available online [30].

2 The Isabelle/HOL Logic Recalled

The *logic of Isabelle/HOL* consists of:

- HOL, that is, classical higher-order logic with rank 1 polymorphism, Hilbert choice and the Infinity axiom (recalled in Section 2.1)
- A definitional mechanism for introducing new types and constants *in an overloaded fashion* (recalled in Section 2.2)

2.1 HOL Syntax and Deduction

The syntax and deduction system we present here are minor variations of the standard ones for HOL (as in, e.g., [3, 18, 20]). What we call *HOL axioms* correspond to the theory INIT from [3].

Syntax. Throughout this section, we fix the following:

- an infinite set TVar, of *type variables*, ranged by α, β
- an infinite set VarN, of *(term) variable names*, ranged by x, y, z
- a set K of symbols, ranged by k, called *type constructors*, containing three special symbols: bool, ind and $\to$ (aimed at representing the type of booleans, an infinite type and the function type constructor, respectively)

We fix a function $\mathsf{arOf} : K \to \mathbb{N}$ giving arities to type constructors, such that $\mathsf{arOf}(\mathsf{bool}) = \mathsf{arOf}(\mathsf{ind}) = 0$ and $\mathsf{arOf}(\to) = 2$. Types, ranged by σ, τ, are defined as follows:

$$\sigma = \alpha \mid (\sigma_1, \ldots, \sigma_{\mathsf{arOf}(k)})\, k$$

Thus, a type is either a type variable or an n-ary type constructor k postfix-applied to a number of types corresponding to its arity. If $n = 1$, instead of $(\sigma)\, k$ we write $\sigma\, k$. We write Type for the set of types.

Finally, we fix the following:

- a set Const, ranged over by c, of symbols called *constants*, containing five special symbols: $\longrightarrow$, $=$, ε, zero and 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:

$\mathsf{tpOf}(\longrightarrow) = \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$ $\quad$ $\mathsf{tpOf}(\mathsf{zero}) = \mathsf{ind}$
$\mathsf{tpOf}(=) = \alpha \to \alpha \to \mathsf{bool}$ $\quad$ $\mathsf{tpOf}(\mathsf{suc}) = \mathsf{ind} \to \mathsf{ind}$
$\mathsf{tpOf}(\varepsilon) = (\alpha \to \mathsf{bool}) \to \alpha$

$\mathsf{TV}(\sigma)$ is the set of variables of a type σ. Given a function $\rho : \mathsf{TVar} \to \mathsf{Type}$, its *support* is the set of type variables where ρ is not the identity: $\mathsf{supp}(\rho) = \{\alpha \mid \rho(\alpha) \neq \alpha\}$. A *type substitution* is a function $\rho : \mathsf{TVar} \to \mathsf{Type}$ with finite support. We let TSubst denote the set of type substitutions. Each $\rho \in \mathsf{TSubst}$ extends to a function $\overline{\rho} : \mathsf{Type} \to \mathsf{Type}$ by defining $\overline{\rho}(\alpha) = \rho(\alpha)$ and $\overline{\rho}((\sigma_1, \ldots, \sigma_n)\, k) = (\overline{\rho}(\sigma_1), \ldots, \overline{\rho}(\sigma_n))\, k$. We write $\sigma[\tau/\alpha]$ for $\overline{\rho}(\sigma)$, 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.

We say that σ is an *instance* of τ *via a substitution* of $\rho \in \mathsf{TSubst}$, written $\sigma \leq_\rho \tau$, if $\overline{\rho}(\tau) = \sigma$. We say that σ is an *instance* of τ, written $\sigma \leq \tau$, if there exists $\rho \in \mathsf{TSubst}$ such that $\sigma \leq_\rho \tau$. Two types σ_1 and σ_2 are called *orthogonal*, written $\sigma_1 \,\#\, \sigma_2$, if they have no common instance; i.e., for all τ it holds that $\tau \not\leq \sigma_1$ or $\tau \not\leq \sigma_2$.

A *(typed) variable* is a pair of a variable name x and a type σ, written x_σ. Let Var denote the set of all variables. A *constant instance* is a pair of a constant and a type, written c_σ, such that $\sigma \leq \mathsf{tpOf}(c)$. We let CInst denote the set of constant instances. We extend the notions of being an instance ($\leq$) and being orthogonal (#) from types to constant instances:

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

The tuple $(K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf})$, which will be fixed in what follows, is called a *signature*. This signature's *terms*, ranged over by s, t, are defined by the grammar:

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

Thus, a term is either a typed variable, or a constant instance, or an application, or an abstraction. As usual, we identify terms modulo alpha-equivalence. Typing is defined as a binary relation between terms and types, written $t : \sigma$, inductively as follows:

$$\frac{x \in \mathsf{VarN}}{x_\sigma : \sigma} \qquad \frac{c \in \mathsf{Const} \quad \tau \leq \mathsf{tpOf}(c)}{c_\tau : \tau} \qquad \frac{t_1 : \sigma \to \tau \quad t_2 : \sigma}{t_1\, t_2 : \tau} \qquad \frac{t : \tau}{\lambda x_\sigma.\, t : \sigma \to \tau}$$

A term is a well-typed term if there exists a (necessarily unique) type τ such that $t : \tau$. We write $\mathsf{tpOf}(t)$ for this unique τ. We let Term_w be the set of well-typed terms. We can apply a type substitution ρ to a term t, written $\overline{\rho}(t)$, by applying $\overline{\rho}$ to the types of all variables and constant instances occurring in t. $\mathsf{FV}(t)$ is the set of t's free variables. The term t is called *closed* if it has no free variables: $\mathsf{FV}(t) = \emptyset$. We write $t[s/x_\sigma]$ for the term obtained from t by capture-free substituting s for all free occurrences of x_σ.

A *formula* is a term of type bool. The formula connectives and quantifiers are defined in a standard way, starting from the implication and equality primitives. When writing terms, we sometimes omit the types of variables if they can be inferred.

Deduction. A *theory* is a set of closed formulas. The HOL axioms, forming a set denoted by Ax, are the standard ones, containing axioms for equality, infinity, choice, and excluded middle. (The technical report [30] gives more details.) A *context* Γ is a finite set of formulas. The notation $\alpha \notin \Gamma$ (or $x_\sigma \notin \Gamma$) means that the variable α (or x_σ) is not

free in any of the formulas in Γ. We define *deduction* as a ternary relation $\vdash$ between theories D, contexts Γ and formulas φ, written $D; \Gamma \vdash \varphi$.

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

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

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

$$\frac{D; \Gamma \cup \{\varphi\} \vdash \chi}{D; \Gamma \vdash \varphi \longrightarrow \chi}\ (\textsc{ImpI}) \qquad \frac{D; \Gamma \vdash f\ x_\sigma = g\ x_\sigma}{D; \Gamma \vdash f = g}\ [x_\sigma \notin \Gamma]\ (\textsc{Ext})$$

A theory D is called *consistent* if $D; \emptyset \nvdash \mathsf{False}$.

Built-Ins and Non-Built-Ins. 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 types that are *not* built-in. Note that 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 $\mathsf{bool}\ \mathsf{list}$ and $(\alpha \to \beta)\ \mathsf{list}$ are non-built-in types, whereas $\alpha \to \beta\ \mathsf{list}$ is a built-in type.

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

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

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. E.g., if the type constructors real (nullary) and list (unary) are in the signature and if σ is $(\mathsf{bool} \to \alpha\ \mathsf{list}) \to \mathsf{real} \to (\mathsf{bool} \to \mathsf{ind})\ \mathsf{list}$, then $\mathsf{types}^\bullet(\sigma)$ has three elements: $\alpha\ \mathsf{list}$, real and $(\mathsf{bool} \to \mathsf{ind})\ \mathsf{list}$.

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.

As a general notation rule, the superscript $\bullet$ indicates non-built-in items, where an item can be either a type or a constant instance.

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

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

2.2 The Isabelle/HOL Definitional Mechanisms

Constant(-Instance) Definitions. Given $c_\sigma \in \mathsf{CInst}^\bullet$ and a closed term $t : \sigma$, 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., $\mathsf{TV}(t) \subseteq \mathsf{TV}(\sigma)$).

Type Definitions. Given the types $\tau \in \mathsf{Type}^\bullet$ and $\sigma \in \mathsf{Type}$ and the closed term t whose type is $\sigma \to \mathsf{bool}$, we let $\tau \equiv t$ denote the formula

$$(\exists x_\sigma.\ t\ x) \longrightarrow \exists rep_{\tau\to\sigma}.\ \exists abs_{\sigma\to\tau}.(\tau \approx t)^{abs}_{rep} \tag{1}$$

where $(\tau \approx t)^{abs}_{rep}$ is the formula $(\forall x_\tau.\ t\ (rep\ x)) \wedge (\forall x_\tau.\ abs\ (rep\ x) = x) \wedge (\forall y_\sigma.\ t\ y \longrightarrow rep\ (abs\ y) = y)$. We call $\tau \equiv t$ a *type definition*, provided τ has the form $(\alpha_1, \ldots, \alpha_n)\ k$ such that α_i are all distinct type variables and $\mathsf{TV}(t) \subseteq \{\alpha_1, \ldots, \alpha_n\}$. (Hence, we have $\mathsf{TV}(t) \subseteq \mathsf{TV}(\tau)$, which also implies $\mathsf{TV}(\sigma) \subseteq \mathsf{TV}(\tau)$.)

Thus, $\tau \equiv t$ means: provided t represents a nonempty subset of σ, the new type τ is isomorphic to this subset via *abs* and *rep*. Note that this is a *conditional* type definition, which distinguishes Isabelle/HOL from other HOL-based provers where an *unconditional* version is postulated (but only after the user proves nonemptiness). We shall see that this conditional approach, known among the Isabelle developers as the Makarius Wenzel trick, is useful in the overall scheme of proving consistency.

However, as far as user interaction is concerned, Isabelle/HOL proceeds like the other HOL provers, in particular, it requires nonemptiness proofs. When the user issues a command to define τ via $t : \sigma \to \mathsf{bool}$, the system asks the user to prove $\exists x_\sigma.\ t\ x$, after which the new type τ and the morphisms *abs* and *rep* are produced and $(\tau \approx t)^{abs}_{rep}$ is proved by applying Modus Ponens.

An Isabelle/HOL development proceeds by declaring types and constants, issuing constant-instance and type definitions, and proving theorems about them via HOL deduction.[8] Therefore, at any point in the development, there is a finite set D of registered constant-instance and type definitions (over a HOL signature Σ)—we call such a set a *definitional theory*. We are interested in proving the consistency of definitional theories, under the syntactic well-formedness restrictions imposed by the system.

Well-Formed Definitional Theories. Given any binary relation R on $\mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$, we write $R^\downarrow$ for its (type-)substitutive closure, defined as follows: $p\ R^\downarrow\ q$ iff there exist p', q' and a type substitution ρ such that $p = \overline{\rho}(p')$, $q = \overline{\rho}(q')$ 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. We shall write R^+ and R^* for the transitive and the reflexive-transitive closure of R.

Let us fix a definitional theory D. We say D is *orthogonal* if the following hold for any two distinct definitions $def_1, def_2 \in D$:

- either one of them is a type definition and the other is a constant-instance definition
- or both are type definitions with orthogonal left-hand sides, i.e., def_1 has the form $\tau_1 \equiv \ldots$, def_2 has the form $\tau_2 \equiv \ldots$, and $\tau_1 \,\#\, \tau_2$

[8] Isabelle/HOL is a complex software system, allowing interaction at multiple levels, including by the insertion of ML code. What we care about here is of course an abstract notion of an Isabelle/HOL development—employing the logical mechanisms alone.

– or both are constant-instance definitions with orthogonal left-hand sides, i.e., def_1 has the form $c_{\tau_1} \equiv \ldots$, def_2 has the form $d_{\tau_2} \equiv \ldots$, and $c_{\tau_1} \# d_{\tau_2}$

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 in D a definition of the form $u \equiv t$ such that $v \in \mathsf{consts}^\bullet(t) \cup \mathsf{types}^\bullet(t)$
2. $u \in \mathsf{CInst}^\bullet$ such that u has the form $c_{\mathsf{tpOf}(c)}$, and $v \in \mathsf{types}^\bullet(\mathsf{tpOf}(c))$

We call $\rightsquigarrow$ the *dependency relation* associated to D: it shows how the types and constant instances depend on each other through definitions in D. The fact that built-in items do not participate at this relation (as shown by the bullets which restrict to non-built-in items) is justified by the built-in items having a pre-determined interpretation, which prevents them from both "depending" and "being depended upon" [28].

We call the definitional theory D *well-formed* if it is orthogonal and the substitutive closure of its dependency relation, $\rightsquigarrow^{\downarrow}$, is terminating. Orthogonality prevents inconsistency arising from overlapping left-hand sides of definitions: defining $c_{\alpha\times\mathsf{ind}\to\mathsf{bool}}$ to be $\lambda x_{\alpha\times\mathsf{ind}}.\mathsf{False}$ and $c_{\mathsf{ind}\times\alpha\to\mathsf{bool}}$ to be $\lambda x_{\mathsf{ind}\times\alpha}.\mathsf{True}$ yields $\lambda x_{\mathsf{ind}\times\mathsf{ind}}.\mathsf{False} = c_{\mathsf{ind}\times\mathsf{ind}\to\mathsf{bool}} = \lambda x_{\mathsf{ind}\times\mathsf{ind}}.\mathsf{True}$ and hence $\mathsf{False} = \mathsf{True}$. Termination prevents inconsistency arising from circularity, as in the encoding of Russel's paradox in the introduction.

In previous work [28], we proved that these prevention measures are sufficient:

Theorem 1. If D is well-formed, then D is consistent.

Let us briefly recall the difficulties arising in proving the consistency theorem. A main problem problem rests in the fact that (recursive) overloading does not interact well with set-theoretic semantics. This makes it difficult to give a meaning to the overloaded definitions, in spite of the fact that their syntactic dependency terminates.

Example 2. $\mathtt{consts\ c} : \alpha \to \mathtt{bool} \qquad \mathtt{consts\ d} : \alpha$
$\mathtt{typedef}\ (\alpha,\beta)\ \mathtt{k} \equiv \mathtt{\{(x,y)} : \alpha\times\beta \mid \mathtt{c\ x} \wedge \mathtt{c\ y} \vee \mathtt{(x,y) = (d,d)\}}$
$\mathtt{consts\ l} : (\alpha,\beta)\ \mathtt{k} \to \alpha \qquad \mathtt{consts\ r} : (\alpha,\beta)\ \mathtt{k} \to \beta$
$\mathtt{definition\ c} : \mathtt{bool} \to \mathtt{bool} \equiv \lambda\ \mathtt{x.\ True}$
$\mathtt{definition\ c} : \mathtt{nat} \to \mathtt{bool} \equiv \lambda\ \mathtt{x.\ False}$
$\mathtt{definition\ c} : (\alpha,\beta)\ \mathtt{k} \to \mathtt{bool} \equiv \lambda\ \mathtt{x.\ c\ (l}\ x) \wedge \neg\ \mathtt{c\ (r}\ x)$

Here, c and k are mutually dependent. Hence, since c is overloaded, both c and k behave differently depending on the types they are instantiated with or applied to. Here are some examples. Because $c_{\mathsf{bool}\to\mathsf{bool}}$ is (vacuously) true, $(\mathsf{bool}, \mathsf{bool})\ k$ contains four elements (corresponding to all elements of $\mathsf{bool} \times \mathsf{bool}$). On the other hand, because $c_{\mathsf{nat}\to\mathsf{bool}}$ is (vacuously) false, $(\alpha, \mathsf{nat})\ k$ and $(\mathsf{nat}, \alpha)\ k$ each contain one single element (corresponding to (d, d)). Moreover, $(\mathsf{bool}, (\mathsf{bool}, \mathsf{nat})\ k)\ k$ contains two elements, for the following reason: both $c_{\mathsf{bool}\to\mathsf{bool}}$ and $c_{(\mathsf{bool},\mathsf{nat})\ k\to\mathsf{bool}}$ are true, the latter since $c_{\mathsf{bool}\to\mathsf{bool}}$ is true and $c_{\mathsf{nat}\to\mathsf{bool}}$ is false (as required in the definition of $c_{(\alpha,\beta)\ k\to\mathsf{bool}}$); so $(\mathsf{bool}, (\mathsf{bool}, \mathsf{nat})\ k)\ k$ has as many elements as its host type, $\mathsf{bool} \times (\mathsf{bool}, \mathsf{nat})\ k$; and $(\mathsf{bool}, \mathsf{nat})\ k$ has only one element (corresponding to (d, d)). Finally, $(\mathsf{bool}, (\mathsf{nat}, \mathsf{bool})\ k)\ k$ contains only one element, because $c_{(\mathsf{nat},\mathsf{bool})\ k\to\mathsf{bool}}$ is false (by the definitions of $c_{(\alpha,\beta)\ k\to\mathsf{bool}}$ and $c_{\mathsf{nat}\to\mathsf{bool}}$).

In the standard HOL semantics [41], a type constructor such as k is interpreted compositionally, as an operator $[k]$ on sets (from a suitable universe) obtained from k's type definition—here, as a binary operator taking the sets A and B to the set $\{(a,b) \in A \times B \mid [c]_A(a) \wedge [c]_B(b) \vee (a,b) = ([d]_A, [d]_B)\}$, where $([c]_A)_A$ and $([d]_A)_A$ would be the interpretations of c and d as families of sets, with each $[c]_A$ a predicate on A and each $[d]_A$ an element of A. But defining $[k]$ in one go for any sets A and B is impossible here, since the needed instances of $[c]$ are not yet known, and in fact are mutually dependent with $[k]$. This means that, when defining $[k]$ and $[c]$, the inputs A and B would need to be analyzed in an ad hoc fashion, for the (syntactic!) occurrences of $[k]$ itself. The orthogonality and termination of such semantic definitions would be problematic (and, as far as we see, could only be worked out by a heavy machinery that would constrain semantics to behave like syntax—adding syntactic annotations to the interpreting sets). Using John Reynolds's famous wording [42], we conclude that *ad hoc* polymorphism is not set-theoretic.[9]

In [28], we proposed a workaround based on acknowledging that ad hoc overloading regards different instances of the same non-built-in polymorphic type as *completely unrelated types*. Instead of interpreting type constructors as operators on sets, we interpret each non-built-in ground type and constant instance separately, in the order prescribed by the terminating dependency relation. Here, for example, $c_{\mathsf{bool}\to\mathsf{bool}}$ and $c_{\mathsf{nat}\to\mathsf{bool}}$ are interpreted before $(\mathsf{bool},\mathsf{nat})\ k$, which is interpreted before $c_{(\mathsf{bool},\mathsf{nat})\ k\to\mathsf{bool}}$, which is interpreted before $((\mathsf{bool},\mathsf{nat})\ k,\mathsf{bool})\ k$, etc. (But note that termination does not necessarily come from structural descent on types: definitions such as $e_{\mathsf{nat}} \equiv \mathsf{head}(e_{\mathsf{nat\ list}})$ are also acceptable.) Finally, polymorphic formulas are interpreted as the infinite conjunction of the interpretation of all their ground instances: for example, $c_{\alpha\to\mathsf{bool}}\ d_\alpha$ is true iff $c_{\sigma\to\mathsf{bool}}\ d_\sigma$ is true for all ground types σ. This way, we were able to construct a ground model for the definitional theory. And after showing that the deduction rules for (polymorphic) HOL are sound for ground models, we inferred consistency. Thus, our solution was based on a mixture of syntax and semantics: interpret type variables by universally quantifying over all ground instances, and interpret non-built-in ground types disregarding their structure.

Such a hybrid approach, involving a nonstandard semantics, may seem excessive. There is a more common-sense alternative for accommodating the observation that standard semantics cannot be married with ad hoc overloading: view overloaded definitions as mere textual abbreviations. The "semantics" of an overloaded constant will then be the result of unfolding the definitions—but, as we have seen, types must also be involved in this process. This is the alternative taken by our new proof.

3 New Proof of Consistency

The HOL logical infrastructure allows unfolding constant definitions, but not type definitions. To amend this limitation, we take an approach common in mathematics. The reals were introduced by closing the rationals under Cauchy convergence, the complex numbers were introduced by closing the reals under roots of polynomials. Similarly,

[9] Reynolds's result of course refers to (higher-rank) *parametric* polymorphism.

we introduce a logic, HOL with Comprehension (HOLC), by closing HOL under type comprehension—that is, adding to HOL comprehension types to express subsets of the form $\{x : \sigma \mid t\ x\}$ (Section 3.1). While there is some tension between these subsets being possibly empty and the HOLC types having to be nonempty due to the Hilbert choice operator, this is resolved thanks to the HOLC comprehension axioms being conditioned by nonemptiness. With this proviso, HOLC admits standard set-theoretical models, making it manifestly consistent (Section 3.2). In turn, Isabelle/HOL-style overloaded constants and types can be normalized in HOLC by unfolding their definitions (Section 3.3). The normalization process provides an intuition and a justification for the syntactic checks involving non-built-in types and constants. Finally, consistency of Isabelle/HOL is inferable from consistency of HOLC.

3.1 HOL with Comprehension (HOLC)

Syntax. Just like for HOL, we fix the sets TVar (of type variables) and VarN (of term variable names), as well as the following:

- a set K of type constructors including the built-in ones bool, ind, $\to$
- a function $\mathsf{arOf} : K \to \mathbb{N}$ assigning an arity to each type constructor.
- a set Const of constants, including the built-in ones $\longrightarrow$, $=$, ε, zero and suc

The HOLC types and terms, which we call *ctypes* and *cterms*, are defined as follows:

$$\sigma = \alpha \mid (\sigma_1, \ldots, \sigma_{\mathsf{arOf}(k)})\, k \mid \{\!|t|\!\} \qquad\qquad t = x_\sigma \mid c_\sigma \mid t_1\, t_2 \mid \lambda x_\sigma.\, t$$

Above, we highlight the only difference from the HOL types and terms: the comprehension types, whose presence makes the ctypes and cterms mutually recursive. Indeed, $\{\!|t|\!\}$ contains the term t, whereas a typed variable x_σ and a constant instance c_σ contain the type σ. We think of a comprehension type $\{\!|t|\!\}$ with $t : \sigma \to \mathsf{bool}$ as representing a set of elements which in standard mathematical notation would be written $\{x : \sigma \mid t\ x\}$, that is, the set of all elements of σ satisfying t. Var denotes the set of (typed) variables, x_σ. CType and CTerm denotes the sets of ctypes and cterms.

We also fix a function $\mathsf{tpOf} : \mathsf{Const} \to \mathsf{CType}$, assigning ctypes to constants. Similarly to the case of HOL, we call the tuple $(K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf})$, which shall be fixed in what follows, a *HOLC signature*. Since ctypes contain cterms, we define typing mutually inductively together with the notion of a ctype being well-formed (i.e., only containing well-typed terms):

$$\frac{\alpha \in \mathsf{TVar}}{\mathsf{wf}(\alpha)}\ (\mathrm{W}_1) \qquad \frac{\mathsf{wf}(\sigma_1)\ \ldots\ \mathsf{wf}(\sigma_{\mathsf{arOf}(k)})}{\mathsf{wf}((\sigma_1, \ldots, \sigma_{\mathsf{arOf}(k)})\, k)}\ (\mathrm{W}_2) \qquad \frac{t :: \sigma \to \mathsf{bool}}{\mathsf{wf}(\{\!|t|\!\})}\ (\mathrm{W}_3) \qquad \frac{t :: \tau \quad \mathsf{wf}(\sigma)}{\lambda x_\sigma.\, t :: \sigma \to \tau}\ (\textsc{Abs})$$

$$\frac{x \in \mathsf{VarN} \quad \mathsf{wf}(\sigma)}{x_\sigma :: \sigma}\ (\textsc{Var}) \qquad \frac{c \in \mathsf{Const} \quad \mathsf{wf}(\tau) \quad \tau \leq \mathsf{tpOf}(c)}{c_\tau :: \tau}\ (\textsc{Const}) \qquad \frac{t_1 :: \sigma \to \tau \quad t_2 :: \sigma}{t_1\, t_2 :: \tau}\ (\textsc{App})$$

We let CType_w and CTerm_w be the sets of well-formed ctypes and well-typed cterms. Also, we let Var_w be the set of variables x_σ that are well-typed as cterms, i.e., have their ctype σ well-formed.

The notions of type substitution, a type or a constant instance being an instance of ($\leq$) or being orthogonal with (#) another type or constant instance, are defined similarly

to those for HOL. Note that a type $\{\!|t|\!\}$ is unrelated to another type $\{\!|t'|\!\}$ even when the extent of the predicate t' includes that of t. This is because HOLC, like HOL (and unlike, e.g., PVS [39]), has no subtyping—instead, traveling between smaller and larger types is achieved via embedding-projection pairs.

Since in HOLC types may contain terms, we naturally lift term concepts to types. Thus, the free (cterm) variables of a ctype σ, written $\mathsf{FV}(\sigma)$, are all the free variables occurring in the cterms contained in σ. A type is called closed if it has no free variables.

A note on declaration circularity. In HOLC we allow tpOf to produce declaration cycles—for example, the type of a constant may contain instances of that constant, as in $\mathsf{tpOf}(c) = \{\!|c_{\mathsf{bool}}|\!\}$. However, the typing system will ensure that no such cyclic entity will be well-typed. For example, to type an instance c_σ, we need to apply the rule (CONST), requiring that $\{\!|c_{\mathsf{bool}}|\!\}$ be well-formed. For the latter, we need the rule (W_3), requiring that c_{bool} be well-typed. Finally, to type c_{bool}, we again need the rule (CONST), requiring that $\{\!|c_{\mathsf{bool}}|\!\}$ be well-formed. So c_σ can never be typed. It may seem strange to allow constant declarations whose instances cannot be typed (hence cannot belong to well-typed terms and well-formed types)—however, this is harmless, since HOLC deduction only deals with well-typed and well-formed items. Moreover, all the constants translated from HOL will be shown to be well-typed.

Deduction. The notion of formulas and all the related notions are defined similarly to HOL, so that HOL formulas are particular cases of HOLC formulas. In addition to the axioms of HOL (the set Ax), HOLC shall include the following type comprehension axiom type_comp:

$$\forall t_{\alpha\to\mathsf{bool}}.\ (\exists x_\alpha.\ t\ x) \longrightarrow \exists \mathit{rep}_{\{\!|t|\!\}\to\alpha}.\ \exists \mathit{abs}_{\alpha\to\{\!|t|\!\}}.(\{\!|t|\!\} \approx t)^{\mathit{abs}}_{\mathit{rep}}$$

This axiom is nothing but a generalization of the HOL type definition $\tau \equiv t$, taking advantage of the fact that in HOLC we have a way to write the expression defining τ as the type $\{\!|t|\!\}$. Note that α is a type variable standing for an arbitrary type, previously denoted by σ. Thus, HOLC allows us to express what in HOL used to be a schema (i.e., an infinite set of formulas, one for each type σ) by a single axiom.

HOLC's deduction $\Vdash$ is defined by the same rules as HOL's deduction $\vdash$, but applied to ctypes and cterms instead of types and terms and using the additional axiom type_comp. Another difference from HOL is that HOLC deduction does not factor in a theory D—this is because we do not include any definitional principles in HOLC.

$$\frac{}{\Gamma \Vdash \varphi}\ [\varphi \in \mathsf{Ax} \cup \{\mathsf{type\_comp}\}]\ (\textsc{Fact}) \qquad \frac{}{\Gamma \Vdash \varphi}\ [\varphi \in \Gamma]\ (\textsc{Assum})$$

$$\frac{\Gamma \Vdash \varphi}{\Gamma \Vdash \varphi[\sigma/\alpha]}\ [\alpha \notin \Gamma]\ (\text{T-}\textsc{Inst}) \qquad \frac{\Gamma \Vdash \varphi}{\Gamma \Vdash \varphi[t/x_\sigma]}\ [x_\sigma \notin \Gamma]\ (\textsc{Inst})$$

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

$$\frac{\Gamma \cup \{\varphi\} \Vdash \chi}{\Gamma \Vdash \varphi \longrightarrow \chi}\ (\textsc{ImpI}) \qquad \frac{\Gamma \Vdash f\ x_\sigma = g\ x_\sigma}{\Gamma \Vdash f = g}\ [x \notin \Gamma]\ (\textsc{Ext})$$

3.2 Consistency of HOLC

In a nutshell, HOLC is consistent for a similar reason that HOL (without definitions) is consistent: the types have a straightforward set-theoretic interpretation and the deduction rules are manifestly sound w.r.t. this interpretation. Similar logics, employing mutual dependency between types and terms, have been shown to be consistent for the foundations of Coq [6] and PVS [39].

Compared to these logics, the only twist of HOLC is that all types have to be nonempty. Indeed, HOLC inherits from HOL the polymorphic Hilbert choice operator, $\varepsilon : (\alpha \to \mathsf{bool}) \to \alpha$, which immediately forces all types to be inhabited, e.g., by $\varepsilon\,(\lambda x_\sigma.\ \mathsf{True})$.

From a technical point of view, this nonemptiness requirement is easy to satisfy. The only types that are threatened by emptiness are the comprehension types $\{\!|t|\!\}$. We will interpret them according to their expected semantics, namely, as the subset of σ for which t holds, *only if this subset turns out to be nonempty*; otherwise we will interpret them as a fixed singleton set $\{*\}$. This is consistent with the HOLC comprehension axiom, type_comp, which only requires that $\{\!|t|\!\}$ have the expected semantics if $\exists x_\alpha.\ t\ x$ holds. Notice how the Makarius Wenzel trick of introducing type definitions as conditional statements in Isabelle/HOL (recalled on page 7), which has inspired a similar condition for type_comp, turns out to be very useful in our journey. Of all the HOL-based provers, this "trick" is only used by Isabelle/HOL, as if anticipating the need for a more involved argument for consistency.

A full-frame model for HOLC. We fix a Grothendieck universe $\mathcal{V}$ and let $\mathcal{U} = \mathcal{V} \setminus \{\emptyset\}$ (since all types will have nonempty interpretations). We fix the following items in $\mathcal{U}$ and operators on $\mathcal{U}$:

- a two-element set $\mathbb{B} = \{\mathsf{false}, \mathsf{true}\} \in \mathcal{U}$
- a singleton set $\{*\} \in \mathcal{U}$
- for each $k \in K$, a function $\boxed{k} : \mathcal{U}^{\mathsf{arOf}(k)} \to \mathcal{U}$
- a global choice function, choice, that assigns to each nonempty set $A \in \mathcal{U}$ an element $\mathsf{choice}(A) \in A$

We wish to interpret well-formed ctypes and well-typed cterms, u, as items $[u]$ in $\mathcal{U}$. Since ctypes and cterms are mutually dependent, not only the interpretations, but also their domains need to be defined recursively. Namely, we define the following notions together, by structural recursion on $u \in \mathsf{CType}_w \cup \mathsf{CTerm}_w$:

- the set $\mathsf{Compat}(u)$, of *compatible valuation functions* $\xi : \mathsf{TV}(u) \cup \mathsf{FV}(u) \to \mathcal{U}$
- the interpretation $[u] : \mathsf{Compat}(u) \to \mathcal{U}$

For each u, assuming $[v]$ has been defined for all structurally smaller $v \in \mathsf{CType}_w \cup \mathsf{CTerm}_w$, we take $\mathsf{Compat}(u)$ to consist of all functions $\xi : \mathsf{TV}(u) \cup \mathsf{FV}(u) \to \mathcal{U}$ such that $\xi(x_\sigma) \in [\sigma]\,(\xi\restriction_{\mathsf{TV}(\sigma) \cup \mathsf{FV}(\sigma)})$ for all $x_\sigma \in \mathsf{FV}(u)$. (Here, $\xi\restriction_{\mathsf{TV}(\sigma) \cup \mathsf{FV}(\sigma)}$ denotes the restriction of ξ to the indicated set, which is clearly included in ξ's domain, since $\mathsf{TV}(\sigma) \subseteq \mathsf{TV}(u)$ and $\mathsf{FV}(\sigma) \subseteq \mathsf{FV}(u)$.)

In turn, $[u]$ is defined as shown below. First, the equations for type interpretations:

$$[\alpha](\xi) = \xi(\alpha) \tag{2}$$

$$[\mathsf{bool}](\xi) = \mathbb{B} \tag{3}$$

$$[\mathsf{ind}](\xi) = \mathbb{N} \quad \text{(the set of natural numbers)} \tag{4}$$

$$[\sigma_1 \to \sigma_2](\xi) = [\sigma_1](\xi_1) \to [\sigma_2](\xi_2)$$
$$\text{(the set of functions from } [\sigma_1](\xi_1) \text{ to } [\sigma_2](\xi_2)) \tag{5}$$
$$\text{where } \xi_i \text{ is the restriction of } \xi \text{ to } \mathsf{Compat}(\sigma_i)$$

$$[(\sigma_1, \ldots, \sigma_n)\ k](\xi) = \boxed{k}\ ([\sigma_1](\xi_1), \ldots, [\sigma_n](\xi_n)) \text{ for } \overline{\sigma}\ k \in \mathsf{Type}^\bullet$$
$$\text{where } \xi_i \text{ is the restriction of } \xi \text{ to } \mathsf{Compat}(\sigma_i) \tag{6}$$

$$[\{\!|t|\!\}](\xi) = \begin{cases} \{x \in [\sigma](\xi) \mid [t](\xi)\ x = \mathsf{true}\} & \text{if set nonempty and } t : \sigma \to \mathsf{bool} \\ \{*\} & \text{otherwise} \end{cases} \tag{7}$$

The equation (7) shows how we interpret comprehension types with no inhabitants (e.g., $\{\!|\lambda x_{\mathsf{ind}}.\ \mathsf{False}|\!\}$)—we chose the singleton set $\{*\}$ (in fact, any nonempty set would do the job). As previously discussed, this conforms to the type_comp axiom, which only prescribes the meaning of inhabited comprehension types.

Next, the equations for term interpretations:

$$[\longrightarrow_{\mathsf{bool}\to\mathsf{bool}\to\mathsf{bool}}](\xi) \text{ as the logical implication on } \mathbb{B} \tag{8}$$

$$[=_{\tau\to\tau\to\mathsf{bool}}](\xi) \text{ as the equality predicate in } [\tau](\xi) \to [\tau](\xi) \to \mathbb{B} \tag{9}$$

$$[\varepsilon_{(\tau\to\mathsf{bool})\to\tau}](\xi)(f) = \begin{cases} \mathsf{choice}(A_f) & \text{if } A_f \text{ is nonempty} \\ \mathsf{choice}([\tau](\xi)) & \text{otherwise,} \end{cases} \tag{10}$$
$$\text{where } A_f = \{a \in [\tau](\xi) \mid f(a) = \mathsf{true}\} \text{ for each } f : [\tau](\xi) \to \mathbb{B}$$

$$[\mathsf{zero}_{\mathsf{ind}}](\xi) = 0 \text{ and } [\mathsf{suc}_{\mathsf{ind}\to\mathsf{ind}}](\xi) \text{ as the successor function for } \mathbb{N} \tag{11}$$

$$[c_\sigma](\xi) = \mathsf{choice}\ ([\sigma](\xi)) \tag{12}$$

$$[x_\sigma](\xi) = \begin{cases} \xi(x_\sigma) & \text{if } \xi(x_\sigma) \in [\sigma](\xi') \\ \mathsf{choice}\ ([\sigma](\xi')) & \text{otherwise,} \end{cases} \tag{13}$$
$$\text{where } \xi' \text{ is the restriction of } \xi \text{ to } \mathsf{Compat}(\sigma)$$

$$[t_1\ t_2](\xi) = [t_1](\xi_1)\ [t_2](\xi_2) \text{ where } \xi_i \text{ is the restriction of } \xi \text{ to } \mathsf{Compat}(t_i) \tag{14}$$

$$[\lambda x_\sigma.\ t](\xi) = \Lambda_{a\in[\sigma](\xi')}[t](\xi[x_\sigma \leftarrow a])$$
$$\text{where } \xi' \text{ is the restriction of } \xi \text{ to } \mathsf{Compat}(\sigma) \tag{15}$$

Since the logic has no definitions, we are free to choose any interpretation for non-built-in constant instances—as seen in (12), we do this using the global choice operator choice. In (15), we use Λ for meta-level lambda-abstraction.

We say that a formula φ *is true under the valuation* $\xi \in \mathsf{Compat}(\varphi)$ if $[\varphi](\xi) = \mathsf{true}$. We say that φ *is (unconditionally) true* if it is true under all $\xi \in \mathsf{Compat}(\varphi)$. Given a context Γ and a formula φ, we write $\Gamma \models \varphi$ to mean that $\bigwedge \Gamma \longrightarrow \varphi$ is true, where $\bigwedge \Gamma$ is the conjunction of all formulas in Γ.

Theorem 3. HOLC is consistent, in that $\emptyset \not\Vdash \mathsf{False}$.

Proof. It is routine to verify that HOLC's deduction is sound w.r.t. to its semantics: for every HOLC deduction rule of the form

$$\frac{\Gamma_1 \Vdash \varphi_1 \qquad \ldots \qquad \Gamma_n \Vdash \varphi_n}{\Gamma \Vdash \varphi}$$

it holds that $\Gamma \models \varphi$ if $\Gamma_i \models \varphi_i$ for all $i \leq n$. Then $\emptyset \not\Vdash \mathsf{False}$ follows from $\emptyset \not\models \mathsf{False}$. □

3.3 Translation of Isabelle/HOL to HOLC

We fix a HOL signature $\Sigma = (K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf})$ and an Isabelle/HOL well-formed definitional theory D over Σ. We will produce a translation of the types and well-typed terms of Σ into well-formed ctypes and well-typed cterms of the HOLC signature $\Sigma_D = (K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf}_D)$ (having the same type constructors and constants as Σ). The typing function $\mathsf{tpOf}_D : \mathsf{Const} \to \mathsf{CType}$ will be defined later. For Σ_D, we use all the notations from Section 3.1—we write CType and CTerm for the sets of cterms and ctypes, etc.

The translation will consist of two homonymous "normal form" functions $\mathsf{NF} : \mathsf{Type} \to \mathsf{CType}_w$ and $\mathsf{NF} : \mathsf{Term}_w \to \mathsf{CTerm}_w$. However, since we have not yet defined tpOf_D, the sets CType_w and CTerm_w (of well-formed ctypes and well-typed cterms) are not yet defined either. To bootstrap the definitions, we first define $\mathsf{NF} : \mathsf{Type} \to \mathsf{CType}$ and $\mathsf{NF} : \mathsf{Term}_w \to \mathsf{CTerm}$, then define tpOf_D, and finally show that the images of the NF functions are included in CType_w and CTerm_w.

The NF functions are defined mutually recursively by two kinds of equations. First, there are equations for recursive descent in the structure of terms and types:

$$\mathsf{NF}(t_1\ t_2) = \mathsf{NF}(t_1)\ \mathsf{NF}(t_2) \tag{16}$$
$$\mathsf{NF}(\lambda x_\sigma.\ t) = \lambda x_{\mathsf{NF}(\sigma)}.\ \mathsf{NF}(t) \tag{17}$$
$$\mathsf{NF}(x_\sigma) = x_{\mathsf{NF}(\sigma)} \tag{18}$$
$$\mathsf{NF}(c_\sigma) = c_{\mathsf{NF}(\sigma)} \text{ if } c_\sigma \notin \mathsf{CInst}^\bullet \tag{19}$$
$$\mathsf{NF}(\sigma \to \tau) = \mathsf{NF}(\sigma) \to \mathsf{NF}(\tau) \tag{20}$$
$$\mathsf{NF}(\mathsf{bool}) = \mathsf{bool} \tag{21}$$
$$\mathsf{NF}(\mathsf{ind}) = \mathsf{ind} \tag{22}$$
$$\mathsf{NF}(\alpha) = \alpha \tag{23}$$

Second, there are equations for unfolding the definitions in D. But before listing these, we need some notation. Given $u, v \in \mathsf{Type} \cup \mathsf{Term}_w$, we write $u \equiv^\downarrow v$ to mean that there exists a definition $u' \equiv v'$ in D and a type substitution ρ such that $u = \rho(u')$ and $v = \rho(v')$. This notation is intuitively consistent (although slightly abusively so) with the notation for the substitutive closure of a relation, where we would pretend that $\equiv$ is a relation on $\mathsf{Type} \cup \mathsf{Term}_w$, with $u' \equiv v'$ meaning $(u' \equiv v') \in D$. By Orthogonality, we have that, for all $u \in \mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$, there exists at most one $v \in \mathsf{Type} \cup \mathsf{Term}_w$ such that $u \equiv^\downarrow v$. Here are the equations for unfolding:

$$\mathsf{NF}(c_\sigma) = \begin{cases} c_{\mathsf{NF}(\sigma)} & \text{if there is no matching definition for } c_\sigma \text{ in } D \\ \mathsf{NF}(t) & \text{if there exists } t \text{ such that } c_\sigma \equiv^\downarrow t \end{cases} \tag{24}$$

$$\mathsf{NF}(\sigma) = \begin{cases} \sigma & \text{if there is no matching definition for } \sigma \text{ in } D \\ \{\!|\mathsf{NF}(t)|\!\} & \text{if there exists } t : \tau \to \mathsf{bool} \text{ such that } \sigma \equiv^\downarrow t \end{cases} \tag{25}$$

Thus, the functions NF first traverse the terms and types "vertically," delving into the built-in structure (function types, λ-abstractions, applications, etc.). When a non-built-in item is being reached that is matched by a definition in D, NF proceed "horizontally" by unfolding this definition. Since the right-hand side of the definition can be any term, NF switch again to vertical mode. Hence, NF repeatedly unfold the definitions when a definitional match in a subexpression is found, following a topmost-first strategy (with the exception that proper subexpressions of non-built-in types are not investigated). For example, if a constant c_σ is matched by a definition, as in $c_\sigma \equiv^\downarrow t$, then c_σ is eagerly unfolded to t, as opposed to unfolding the items occurring in σ. This seems to be a

reasonable strategy, given that after unfolding c_σ the possibility to process σ is not lost: since $t : \sigma$, we have that σ occurs in t.

Example 4. $\texttt{consts c} : \alpha \to \texttt{bool} \qquad \texttt{consts d} : \alpha$
$\texttt{typedef}\ \alpha\ \texttt{k} \equiv \texttt{\{x} : \alpha \mid \texttt{c d\}}$
$\texttt{definition c} : \alpha\ \texttt{k} \to \texttt{bool} \equiv \lambda\ \texttt{x} : \alpha\ \texttt{k}.\ \texttt{(c} : \alpha \to \texttt{bool) d}$

Let us show the results of applying NF on some of the constant instances and types in the above example.

$$\mathsf{NF}(\alpha\, k) \;=\; \{\!|\lambda x_\alpha.\, c_{\alpha\to\mathsf{bool}}\, d_\alpha|\!\}$$
$$\mathsf{NF}(c_{\mathsf{bool}\, k\to\mathsf{bool}}) \;=\; \lambda x_{\{\!|\lambda x_{\mathsf{bool}}.\, c_{\mathsf{bool}\to\mathsf{bool}}\, d_{\mathsf{bool}}|\!\}}.\, c_{\mathsf{bool}\to\mathsf{bool}}\, d_{\mathsf{bool}}$$
$$\mathsf{NF}(\mathsf{bool}\, k^2) \;=\; \{\!|\lambda x_{\{\!|\lambda x_{\mathsf{bool}}.\, c_{\mathsf{bool}\to\mathsf{bool}}\, d_{\mathsf{bool}}|\!\}}.\ (\lambda x_{\{\!|\lambda x_{\mathsf{bool}}.\, c_{\mathsf{bool}\to\mathsf{bool}}\, d_{\mathsf{bool}}|\!\}}.\, c_{\mathsf{bool}\to\mathsf{bool}}\, d_{\mathsf{bool}})\, d_{\{\!|\lambda x_{\mathsf{bool}}.\, c_{\mathsf{bool}\to\mathsf{bool}}\, d_{\mathsf{bool}}|\!\}}|\!\}$$

The evaluation of NF on $\mathsf{bool}\, k^n$ terminates in a number of steps depending on n, and the result contains n levels of comprehension-type nesting.

The first fact that we need to show is that NF is well-defined, i.e., its recursive calls terminate. For this, we take the relation $\blacktriangleright$ to be $\equiv^{\downarrow} \cup\ \rhd$, where $\equiv^{\downarrow}$ was defined above and $\rhd$ simply contains the structural recursive calls of NF:

$$t_1 t_2 \rhd t_1 \qquad \lambda x_\sigma.\, t \rhd \sigma \qquad x_\sigma \rhd \sigma \qquad \sigma_1 \to \sigma_2 \rhd \sigma_1$$
$$t_1 t_2 \rhd t_2 \qquad \lambda x_\sigma.\, t \rhd t \qquad c_\sigma \rhd \sigma \qquad \sigma_1 \to \sigma_2 \rhd \sigma_2$$

It is immediate to see that $\blacktriangleright$ captures the recursive calls of NF: the structural calls via $\rhd$ and the unfolding calls via $\equiv^{\downarrow}$. So the well-definedness of NF is reduced to the termination of $\blacktriangleright$.

Lemma 5. The relation $\blacktriangleright$ is terminating (hence the functions NF are well-defined).

Proof. We shall use the following crucial fact, which follows by induction using the definitions of $\rhd$ and $\rightsquigarrow^{\downarrow}$: If $u, v \in \mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$ and $u \equiv^{\downarrow} t \rhd^* v$, then $u \rightsquigarrow^{\downarrow} v$. (*)

Let us assume, for a contradiction, that $\blacktriangleright$ does not terminate. Then there exists an infinite sequence $(w_i)_{i\in\mathbb{N}}$ such that $w_i \blacktriangleright w_{i+1}$ for all i. Since $\blacktriangleright$ is defined as $\equiv^{\downarrow}\cup\rhd$ and $\rhd$ clearly terminates, there must exist an infinite subsequence $(w_{i_j})_{j\in\mathbb{N}}$ such that $w_{i_j} \equiv^{\downarrow} w_{i_j+1} \rhd^* w_{i_{j+1}}$ for all j. Since from the definition of $\equiv$ we have $w_{i_j} \in \mathsf{Type}^\bullet \cup \mathsf{CInst}^\bullet$, we obtain from (*) that $w_{i_j} \rightsquigarrow^{\downarrow} w_{i_{j+1}}$ for all j. This contradicts the termination of $\rightsquigarrow^{\downarrow}$. □

With NF in place, we can define the missing piece of the target HOLC signature: we take tpOf_D to be the normalized version of tpOf, i.e. $\mathsf{tpOf}_D(c) = \mathsf{NF}(\mathsf{tpOf}(c))$.

Lemma 6. NF preserves typing, in the following sense:

- $\mathsf{NF}(\sigma)$ is well-formed in HOLC.
- If $t : \tau$, then $\mathsf{NF}(t) :: \mathsf{NF}(\tau)$.

Our main theorem about the translation will be its soundness:

Theorem 7. If $D; \emptyset \vdash \varphi$ in HOL, then $\emptyset \Vdash \mathsf{NF}(\varphi)$ in HOLC.

Let us focus on proving this theorem. If we define $\mathsf{NF}(\Gamma)$ as $\{\mathsf{NF}(\varphi) \mid \varphi \in \Gamma\}$, the proof that $D; \Gamma \vdash \varphi$ implies $\mathsf{NF}(\Gamma) \Vdash \mathsf{NF}(\varphi)$ should proceed by induction on the definition of $D; \Gamma \vdash \varphi$. Due to the similarity of $\vdash$ and $\Vdash$, most of the cases go smoothly.

For the HOL rule (BETA), we need to prove $\mathsf{NF}(\Gamma) \Vdash \mathsf{NF}((\lambda x_\sigma.\, t)\ s = t[s/x_\sigma])$, that is, $\mathsf{NF}(\Gamma) \Vdash (\lambda x_{\mathsf{NF}(\sigma)}.\, \mathsf{NF}(t))\ \mathsf{NF}(s) = \mathsf{NF}(t[s/x_\sigma])$. Hence, in order to conclude the proof for this case using the HOLC rule (BETA), we need that NF commutes with term substitution—this is not hard to show, since substituting terms for variables does not influence the matching of definitions, i.e., the behavior of NF:

Lemma 8. $\mathsf{NF}(t[s/x_\sigma]) = \mathsf{NF}(t)\ [\mathsf{NF}(s)/x_{\mathsf{NF}(\sigma)}]$

However, our proof (of Theorem 7) gets stuck when handling the (T-INST) case. It is worth looking at this difficulty, since it is revealing about the nature of our encoding. We assume that in HOL we inferred $D; \Gamma \vdash \varphi[\sigma/\alpha]$ from $D; \Gamma \vdash \varphi$, where $\alpha \notin \Gamma$. By the induction hypothesis, we have $\mathsf{NF}(\Gamma) \Vdash \mathsf{NF}(\varphi)$. Hence, by applying (T-INST) in HOLC, we obtain $\mathsf{NF}(\Gamma) \Vdash \mathsf{NF}(\varphi)[\mathsf{NF}(\sigma)/\mathsf{NF}(\alpha)]$. Therefore, to prove the desired fact, we would need that the NF functions commute with type substitutions in formulas, and therefore also in arbitrary terms (which may be contained in formulas):

$$\mathsf{NF}(t[\sigma/\alpha]) = \mathsf{NF}(t)[\mathsf{NF}(\sigma)/\alpha]$$

But this is not true, as seen, e.g., when $\mathsf{tpOf}(c) = \alpha$ and $c_{\mathsf{bool}} \equiv \mathsf{True}$ is in D:

$$\mathsf{NF}(c_\alpha[\mathsf{bool}/\alpha]) = \mathsf{NF}(c_{\mathsf{bool}}) = \mathsf{True} \neq c_{\mathsf{bool}} = c_\alpha[\mathsf{bool}/\alpha] = \mathsf{NF}(c_\alpha)\ [\mathsf{NF}(\mathsf{bool})/\alpha]$$

The problem resides at the very essence of overloading: a constant c is declared at a type σ (α in the above example) and defined at a less general type τ (bool in the example). Our translation reflects this: it leaves c_σ as it is, whereas it compiles away c_τ by unfolding its definition. So then how can such a translation be sound? Essentially, it is sound because in HOL nothing interesting can be deduced about the undefined c_σ that may affect what is being deduced about c_τ—hence it is OK to decouple the two when moving to HOLC.

To capture this notion, of an undefined c_σ not affecting a defined instance c_τ in HOL, we introduce a variant of HOL deduction that restricts type instantiation—in particular, it will not allow arbitrary statements about c_σ to be instantiated to statements about c_τ. Concretely, we define $\vdash'$ by modifying $\vdash$ as follows. We remove (T-INST) and strengthen (FACT) to a rule that combines the use of axioms with type instantiation:

$$\frac{}{D; \Gamma \vdash' \overline{\rho}(\varphi)}\ [\varphi \in \mathsf{Ax} \cup D \text{ and } \forall \alpha \in \mathsf{supp}(\rho).\, \alpha \notin \Gamma]\ \text{(FACT-T-INST)}$$

(where ρ is a type substitution). Note the difference between (FACT-T-INST) and the combination of (FACT) and (T-INST): in the former, only axioms and elements of D are allowed to be type-instantiated, whereas in the latter instantiation can occur at any moment in the proof. It is immediate to see that $\vdash$ is at least as powerful as $\vdash'$, since (FACT-T-INST) can be simulated by (FACT) and (T-INST). Conversely, it is routine to show that $\vdash'$ is closed under type substitution, and a fortiori under (T-INST); and (FACT-T-INST) is stronger than (FACT).

Using $\vdash'$ instead of $\vdash$, we can prove Theorem 7. All the cases that were easy with $\vdash$ are also easy with $\vdash'$. In addition, for the case (FACT-T-INST) where one infers $D; \Gamma \vdash$

$\rho(\varphi)$ with $\varphi \in \mathsf{Ax}$, we need a less general lemma than commutation of NF in an arbitrary term. Namely, *noticing that the HOL axioms do not contain non-built-in constants or types*, we need the following lemma, which can be proved by routine induction over t:

Lemma 9. $\mathsf{NF}(\overline{\rho}(t)) = \overline{\mathsf{NF} \circ \rho}(t)$ whenever $\mathsf{types}^\bullet(t) \cup \mathsf{consts}^\bullet(t) = \emptyset$.

Now, assume (FACT-T-INST) is being used to derive $D; \Gamma \vdash' \overline{\rho}(\varphi)$ for $\varphi \in \mathsf{Ax}$. We need to prove $\Gamma \Vdash \mathsf{NF}(\overline{\rho}(\varphi))$, that is, $\Gamma \Vdash \overline{\mathsf{NF} \circ \rho}(\varphi)$. But this follows from n applications of the (T-INST) rule (in HOLC), where n is the size of $\overline{\mathsf{NF} \circ \rho}$'s support (as any finite-support simultaneous substitution can be reduced to a sequence of unary substitutions).

It remains to handle the case when (FACT-T-INST) is being used to derive $D; \Gamma \vdash' \overline{\rho}(\varphi)$ for $\varphi \in D$. Here, Lemma 9 does not apply, since of course the definitions in D contain non-built-in items. However, we can take advantage of the particular shape of the definitions. The formula φ necessarily has the form $u \equiv t$. By Orthogonality, it follows that $\overline{\rho}(t)$ is the unique term such that $\overline{\rho}(u) \equiv^{\downarrow} \overline{\rho}(t)$. We have two cases:

- If u is a constant instance c_σ, then by the definition of NF we have $\mathsf{NF}(\overline{\rho}(u)) = \mathsf{NF}(\overline{\rho}(t))$. But then $\Gamma \Vdash \mathsf{NF}(\rho(\varphi))$, that is, $\Gamma \Vdash \mathsf{NF}(\overline{\rho}(u)) = \mathsf{NF}(\overline{\rho}(t))$, follows from (FACT) applied with the reflexivity axiom.
- If u is a type σ and $t : \tau \to \mathsf{bool}$, then $\overline{\rho}(\varphi)$ is $\overline{\rho}(\sigma) \equiv^{\downarrow} \overline{\rho(t)}$. In other words, $\overline{\rho}(\varphi)$ has the format of a HOL type definition, just like φ. Hence, $\mathsf{NF}(\overline{\rho}(\varphi))$ is seen to be an instance of type_comp, namely, $\mathsf{type\_comp}[\mathsf{NF}(\overline{\rho}(\sigma))/\alpha]$ together with $\mathsf{NF}(\overline{\rho}(t))$ substituted for the first quantifier. Hence $\Gamma \Vdash \mathsf{NF}(\overline{\rho}(\varphi))$ follows from (FACT) applied with type_comp, followed by $\forall$-instantiation (the latter being the standardly derived rule for $\forall$).

In summary, our HOLC translation of overloading emulates overloading itself in that it treats the defined constant instances c_τ as being disconnected from their "mother" instances $c_{\mathsf{tpOf}(c)}$. The translation is sound thanks to the fact that the considered theory has no axioms about these constants besides the overloaded definitions. This sound translation immediately allows us to reach our overall goal:

Proof of Theorem 1 (Consistency of Isabelle/HOL). By contradiction. Let $D; \emptyset \vdash \mathsf{False}$. Then by Theorem 7, we obtain $\emptyset \Vdash \mathsf{NF}(\mathsf{False})$ and since $\mathsf{NF}(\mathsf{False}) = \mathsf{False}$, we derive contradiction with Theorem 3. □

4 Application: Logical Extensions

We introduced HOLC as an auxiliary for proving the consistency of Isabelle/HOL's logic. However, a natural question that arises is whether HOLC would be itself a practically useful extension. We cannot answer this question yet, beyond noting that it would be a significant implementation effort due to the need to reorganize types as mutually dependent with terms. Over the years, some other proposals to go beyond HOL arose. For example, an interesting early proposal by Melham to extend the terms with explicit type quantifiers [34] was never implemented. Homeier's HOL_ω [22], an implemented and currently maintained extension of HOL with first-class type constructors, is another example.

A strong argument for using HOL in theorem proving is that it constitutes a sweet spot between expressiveness and simplicity. The expressiveness part of the claim is debatable—and has been challenged, as shown by the above examples, as well as by Isabelle/HOL itself which extends HOL in a nontrivial way. In our recent work we joined the debate club and advocated a new sweet spot for HOL (and for Isabelle/HOL, respectively) [29] by introducing local type definitions and an unoverloading rule expressing parametricity. HOLC plays a special role in this proposal because we use it to prove the extensions' consistency.

In the following, we first introduce and motivate the extensions (Section 4.1), and then discuss how we applied HOLC to justify their consistency and why our previous ground semantic [28] is not suitable for this job (Section 4.2).

4.1 Two Extensions for Traveling From Types to Sets

We start with a theorem stating that all compact sets are closed in T2 spaces (a topological space), whose definition uses an overloaded constant $\mathsf{open} : \alpha\ \mathsf{set} \to \mathsf{bool}$:

$$\forall \alpha_{\mathsf{T2\text{-}space}}.\ \forall S_{\alpha\ \mathsf{set}}.\ \mathsf{compact}\ S \longrightarrow \mathsf{closed}\ S \tag{26}$$

Since we quantify over spaces defined on α here, the theorem is not applicable to spaces defined on a proper subset A of α. Let us recall that types and sets are different syntactic categories in HOL. Defining a new ad hoc type isomorphic to A is undesirable or not even allowed since A can be an open term. Thus a more flexible theorem quantifying over all nonempty carriers A and unary predicates *open* forming a T2 space is needed:

$$\begin{array}{c}\forall \alpha.\ \forall A_{\alpha\ \mathsf{set}}.\ A \neq \emptyset \longrightarrow \forall open_{\alpha\ \mathsf{set}\to\mathsf{bool}}.\ \mathsf{T2\text{-}space}^{\mathsf{on}}_{\mathsf{with}}\ A\ open \longrightarrow \\ \forall S_{\alpha\ \mathsf{set}} \subseteq A.\ \mathsf{compact}^{\mathsf{on}}_{\mathsf{with}}\ A\ open\ S \longrightarrow \mathsf{closed}^{\mathsf{on}}_{\mathsf{with}}\ A\ open\ S\end{array} \tag{27}$$

As the proof automation works better with types, ideally one should only prove type-based theorems such as (26) and automatically obtain set-based theorems such as (27). Unfortunately, this is not possible in HOL, which is frustrating given that (26) and (27) are semantically equivalent (in the standard interpretation of HOL types).

To address the discrepancy and achieve the automatic translation, we extended the logic of Isabelle/HOL by two new rules: Local Typedef (LT) and Unoverloading (UO).

$$\frac{\Gamma \vdash A \neq \emptyset \qquad \Gamma \vdash (\exists abs\ rep.\ (\beta \approx A)^{abs}_{rep}) \longrightarrow \varphi}{\Gamma \vdash \varphi}\ [\beta \text{ fresh}]\ (\mathrm{LT})$$

where $(\beta \approx A)^{abs}_{rep}$ means that β is isomorphic to A via morphisms *abs* and *rep*; basically the core of the formula (1) from Section 2.2, where for notation convenience we identify the set A with its characteristic predicate $\lambda x.\ x \in A$. The rule allows us to assume the existence of a type isomorphic to a nonempty set A (which syntactically is a possibly *open* term) inside of a proof.

To formulate (UO), let us recall that $\rightsquigarrow^{\downarrow}$ is the substitutive closure of the constant–type dependency relation $\rightsquigarrow$ from Section 2.2 on page 8 and let us define Δ_c to be the set of all types for which the constant c was overloaded. The notation $\sigma \not\leq S$ means that σ is not an instance of any type in S. We write $\rightsquigarrow^{\downarrow+}$ for the transitive closure of $\rightsquigarrow^{\downarrow}$.

$$\frac{\varphi[c_\sigma/x_\sigma]}{\forall x_\sigma.\,\varphi}\ [\sigma \not\leq \Delta_c;\ \text{and } u \rightsquigarrow^{\downarrow+} c_\sigma \text{ does not hold for any type or constant } u \text{ in } \varphi] \quad \text{(UO)}$$

Thus, (UO) tells us that if a constant c was not overloaded for σ (or a more general type), the meaning of the constant instance c_σ is unrestricted, i.e., it behaves as a free term variable of the same type. That is to say, the truth of a theorem φ containing c_σ *cannot* depend on the definition of c_τ for some $\tau < \sigma$. In summary, (UO) imposes a certain notion of parametricity, which is willing to cohabitate with ad hoc overloading.

We use the two rules in the translation as follows: the (UO) rule allows us to compile out the overloaded constants from (26) (by a dictionary construction) and thus obtain

$$\forall\alpha.\,\forall open_{\alpha\ \mathsf{set}\to\mathsf{bool}}.\ \mathsf{T2\text{-}space}_{\mathsf{with}}\ open \longrightarrow \ldots. \tag{28}$$

Then we fix a nonempty set A, locally "define" a type β isomorphic to A by (LT) and obtain (27) from the β-instance of (28) along the isomorphism between β and A.

The extensions have already been picked up by Isabelle/HOL power users for translating between different representations of matrices [12, 13], for implementing a certified and efficient algorithm for factorization [11], and for tightly integrating invariants in proof rules for a probabilistic programming language [33].

4.2 Consistency of the Extensions

We will fist show that HOL + (LT) is consistent by showing (LT) to be admissible in HOLC (as a straightforward consequence of type_comp).

Theorem 10. The inference system consisting of the deduction rules of Isabelle/HOL and the (LT) rule is consistent (in that it cannot prove False).

Proof sketch. It is enough if we show that for every step

$$\frac{\Gamma \vdash A \neq \emptyset \qquad \Gamma \vdash (\exists abs\ rep.\ (\beta \approx A)^{abs}_{rep}) \longrightarrow \varphi}{\Gamma \vdash \varphi}\ [\beta \text{ is fresh}]$$

in a HOL proof, we can construct a step in a HOLC proof of $\mathsf{NF}(\Gamma) \Vdash \mathsf{NF}(\varphi)$ given

$$\mathsf{NF}(\Gamma) \Vdash \mathsf{NF}(A) \neq \mathsf{NF}(\emptyset) \tag{29}$$

$$\mathsf{NF}(\Gamma) \Vdash (\exists abs\ rep.\ (\beta \approx \mathsf{NF}(A))^{abs}_{rep}) \longrightarrow \mathsf{NF}(\varphi). \tag{30}$$

The side-condition of the (LT) (freshness of β) transfers into HOLC: if β is fresh for some $u \in \mathsf{Type} \cup \mathsf{Term}$, it must also be fresh for $\mathsf{NF}(u)$. This follows from the fact that unfolding a (type or constant) definition $u \equiv t$ cannot introduce new type variables since we require $\mathsf{TV}(t) \subseteq \mathsf{TV}(u)$. Thus we obtain

$$\mathsf{NF}(\Gamma) \Vdash (\exists abs\ rep.\ (\{\!|\mathsf{NF}(A)|\!\} \approx \mathsf{NF}(A))^{abs}_{rep}) \longrightarrow \mathsf{NF}(\varphi), \tag{31}$$

an instance of (30) where we substituted the witness $\{\!|\mathsf{NF}(A)|\!\}$ for β. As a last step, we discharge the antecedent of (31) by using type_comp (with the help of (29)) and obtain the desired $\mathsf{NF}(\Gamma) \Vdash \mathsf{NF}(\varphi)$. □

We were also able to prove Theorem 10 by using our previous ground semantics, as discussed in Kunčar's thesis [31, Sect. 7.2]. The proof is more technically elaborate and the main idea is to prove that the following principle holds in the semantic world of HOL:

$$\forall\alpha.\, \forall A_{\alpha\,\mathsf{set}}.\, A \neq \emptyset \longrightarrow \exists\beta.\, \exists abs_{\alpha\to\beta}\, rep_{\beta\to\alpha}.\, (\beta \approx A)^{abs}_{rep} \qquad (\star)$$

Working in HOLC gives us the advantage to get closer to ($\star$) in the following sense: for every nonempty set $A : \sigma\,\mathsf{set}$, not only we can postulate that there always exists a type isomorphic to A, but we can even directly express such a type in HOLC as the comprehension $\{\!|A|\!\}$. That is basically what the axiom type_comp tells us. Thus, informally speaking, the property ($\star$) is more first-class in HOLC than in HOL.

In contrast to (LT), we could not use the ground semantics for the consistency of (UO) and this is where HOLC shows its power.

Theorem 11. The inference system consisting of the deduction rules of Isabelle/HOL, the (LT) rule and the (UO) rule is consistent.

Proof sketch. We will first argue that HOLC + (UO) (without its side-conditions, since they do not make sense in HOLC) is still a consistent logic. This means, from $\varphi[c_\sigma/x_\sigma]$ we can derive $\forall x_\sigma.\, \varphi$ in HOLC + (UO). W.l.o.g. let us assume that the interpretation of type constructors in the semantics of HOLC from Section 3.2 is nonoverlapping. Since HOLC does not contain any definitions, we interpret c_σ arbitrarily (as long as the value belongs to the interpretation of σ) in the model construction for HOLC. That is to say, the proof of consistency does not rely on the actual value of c_σ's interpretation, hence we can replace c_σ by a term variable x_σ. Therefore the formula φ must be fulfilled for every evaluation of x_σ.

The second step is to show that NF is a sound embedding of Isabelle/HOL + (LT) + (UO) into HOLC + (UO). Since we have shown that the translation of (LT) is admissible in HOLC, we only need to focus on (UO). The first side condition of (UO) guarantees that unfolding by NF does not introduce any new c_σ and the second one guarantees that NF does not unfold any c_σ. Therefore the substitution $[c_\sigma/x_\sigma]$ commutes with NF, i.e., $\mathsf{NF}(\varphi[c_\sigma/x_\sigma]) = (\mathsf{NF}(\varphi))[c_{\mathsf{NF}(\sigma)}/x_{\mathsf{NF}(\sigma)}]$. □

The reason we could not use the ground semantics to prove Theorem 11 is because the semantics is too coarse to align with the meaning of (UO): that the truth of a theorem φ stating a property of c_σ cannot depend on the fact that a proper instance of c_σ, say, c_τ for $\tau < \sigma$, was already overloaded, say, by a definition $c_\tau \equiv t$. In semantic terms, this means that the interpretation of c_σ cannot depend on the interpretation of c_τ. Recall that in the ground semantics we considered a polymorphic HOL formula φ to be true just in case all its ground type instances are true. (See also the discussion on page 9.) This definition of truth cannot validate (UO). To see this, let us assume that φ is polymorphic only in α and τ is ground. We want to assume the truth of $\varphi[\sigma/\alpha][c_\sigma/x_\sigma]$ and infer the truth of $\forall x_\sigma.\, \varphi[\sigma/\alpha]$. In particular, since $\forall x_\tau.\, \varphi[\tau/\alpha]$ is a ground instance of the latter, we would need to infer that $\forall x_\tau.\, \varphi[\tau/\alpha]$ is true, and in particular that $\varphi[\tau/\alpha][c_\tau/x_\tau]$ is true. But this is impossible, since the interpretation of c_τ in $\varphi[\tau/\alpha][c_\tau/x_\tau]$ is fixed and dictated by the definitional theorem $c_\tau \equiv t$.

5 Conclusions and Related Work

It took the Isabelle/HOL community almost twenty years to reach a definitional mechanism that is consistent by construction, w.r.t. both types and constants.[10] This paper, which presents a clean syntactic argument for consistency, is a culmination of previous efforts by Wenzel [48], Obua [38], ourselves [26,28], and many other Isabelle designers and developers.

The key ingredients of our proof are a type-instantiation restricted version of HOL deduction and HOLC, an extension of HOL with comprehension types. HOLC is similar to a restriction of Coq's Calculus of Inductive Constructions (CiC) [8], where: (a) proof irrelevance and excluded middle axioms are enabled; (b) polymorphism is restricted to rank 1; (c) the formation of (truly) dependent product types is suppressed. However, unlike CiC, HOLC stays faithful to the HOL tradition of avoiding empty types. HOLC also bears some similarities to HOL with predicate subtyping [43] as featured by PVS [44]. Yet, HOLC does not have real subtyping: from $t : \sigma \to \mathsf{bool}$ and $s :: \{\!|t|\!\}$ we cannot infer $s :: \sigma$. Instead, HOLC retains HOL's separation between a type defined by comprehension and the original type: the former is not included, but merely embedded in the latter. Comprehension types are also known in the programming language literature as refinement types [16].

Wiedijk defines *stateless HOL* [49], a version of HOL where types and terms carry definitions *in their syntax*. Kumar et al. [25] define a translation from standard (stateful) HOL with definitions to stateless HOL, on the way of proving the consistency of both. Their translation is similar to our HOL to HOLC translation, in that it internalizes HOL definitions as part of "stateless" formulas in a richer logic.

Although a crucial property, consistency is nevertheless rather weak. One should legitimately expect definitions to enjoy a much stronger property: that they can be compiled away without affecting provability *not in a richer logic (like HOLC), but in HOL itself*. Wenzel calls this property "meta-safety" and proves it for Isabelle/HOL *constant* definitions [48]. In particular, meta-safety yields *proof-theoretic conservativity*, itself stronger than consistency: if a formula that contains no defined item is deducible from a definitional theory, then it is deducible in the core (definition-free) logic. Meta-safety and conservativity for arbitrary definitional theories (factoring in not only constant, but also type definitions) are important meta-theoretic problems, which seem to be open not only for Isabelle/HOL, but also for standard HOL [2]. We leave them as future work.

Acknowledgments. We thank Tobias Nipkow, Larry Paulson, Makarius Wenzel, and the members of the Isabelle mailing list for inspiring and occasionally intriguing opinions and suggestions concerning the foundations of Isabelle/HOL. We also thank the reviewers for suggestions on how to improve the presentation and for indicating related work. Kunčar is supported by the German Research Foundation (DFG) grant "Security Type Systems and Deduction" (Ni 491/13-3) in the priority program "RS^3 – Reliably Secure Software Systems" (SPP 1496). Popescu is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) starting grant "VOWS – Verification of Web-based Systems" (EP/N019547/1).

[10] Isabelle is by no means the only prover with longstanding foundational issues [28, Sect. 1].

References

1. Isabelle Foundation & Certification (2015), archived at https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2015-September/thread.html
2. Conservativity of HOL constant and type definitions (2016), archived at https://sourceforge.net/p/hol/mailman/message/35448054/
3. The HOL system logic (2016), https://sourceforge.net/projects/hol/files/hol/kananaskis-10/kananaskis-10-logic.pdf
4. Type definitions in Isabelle; article "A Consistent Foundation for Isabelle/HOL" by Kunçar/Popescu (2016), archived at https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2016-August/thread.html
5. Arthan, R.: On definitions of constants and types in HOL. J. Autom. Reasoning 56(3), 205–219 (2016)
6. Barras, B.: Sets in Coq, Coq in Sets. Journal of Formalized Reasoning 3(1) (2010)
7. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—Lessons learned in formal-logic engineering. In: TPHOLs. pp. 19–36 (1999)
8. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
9. Blanchette, J.C., Popescu, A., Traytel, D.: Foundational Extensible Corecursion. ICFP '15, ACM (2015)
10. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with isabelle/hol. In: TPHOLs. pp. 134–149 (2008)
11. Divasón, J., Joosten, S., Thiemann, R., Yamada, A.: A formalization of the berlekamp-zassenhaus factorization algorithm. In: CPP. pp. 17–29 (2017)
12. Divasón, J., Kunčar, O., Thiemann, R., Yamada, A.: Certifying Exact Complexity Bounds for Matrix Interpretations. LCC (2016)
13. Divasón, J., Kunčar, O., Thiemann, R., Yamada, A.: Perron-Frobenius Theorem for Spectral Radius Analysis. Archive of Formal Proofs (2016), https://www.isa-afp.org/entries/Perron_Frobenius.shtml
14. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: CAV. pp. 463–478 (2013)
15. Fallenstein, B., Kumar, R.: Proof-producing reflection for HOL - with an application to model polymorphism. In: ITP. pp. 170–186 (2015)
16. Freeman, T., Pfenning, F.: Refinement types for ML. In: PLDI. pp. 268–277 (1991)
17. Geuvers, H.: Proof assistants: History, ideas and future. Sadhana 34(1), 3–25 (2009)
18. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)
19. Haftmann, F., Nipkow, T.: Code Generation via Higher-Order Rewrite Systems. In: FLOPS. Springer (2010)
20. Harrison, J.: HOL light: An overview. In: TPHOLs. pp. 60–66 (2009)
21. Holger Blasum, O.H., Tverdyshev, S.: Euro-mils: Secure european virtualisation for trustworthy applications in critical domains – formal methods used, www.euromils.eu/downloads/Deliverables/Y2/2015-EM-UsedFormalMethods-WhitePaper-October2015.pdf
22. Homeier, P.V.: The HOL-Omega logic. In: TPHOLs 2009. LNCS, vol. 5674, pp. 244–259 (2009)
23. Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: CAV. pp. 167–183 (2014)
24. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commun. ACM 53(6), 107–115 (2010)

25. Kumar, R., Arthan, R., Myreen, M., Owens, S.: HOL with Definitions: Semantics, Soundness, and a Verified Implementation. In: ITP. Springer (2014)
26. Kunčar, O.: Correctness of Isabelle's cyclicity checker: Implementability of overloading in proof assistants. In: CPP. pp. 85–94 (2015)
27. Kunčar, O., Popescu, A.: A Consistent Foundation for Isabelle/HOL – Extended Version, http://www21.in.tum.de/~kuncar/kuncar-popescu-isacons2016.pdf
28. Kunčar, O., Popescu, A.: A Consistent Foundation for Isabelle/HOL. In: ITP. pp. 234–252 (2015)
29. Kunčar, O., Popescu, A.: From Types To Sets By Local Type Definitions in Higher-Order Logic. In: ITP. pp. 200–218 (2016)
30. Kunčar, O., Popescu, A.: Comprehending Isabelle/HOL's consistency (2017), technical report. http://andreipopescu.uk/pdf/compr_IsabelleHOL_cons_TR.pdf
31. Kunčar, O.: Types, Abstraction and Parametric Polymorphism in Higher-Order Logic. Ph.D. thesis, Fakultät für Informatik, Technische Universität München (2016), http://www21.in.tum.de/~kuncar/documents/kuncar-phdthesis.pdf
32. Lochbihler, A.: Verifying a compiler for Java threads. In: ESOP. pp. 427–447 (2010)
33. Lochbihler, A.: Probabilistic functions and cryptographic oracles in higher order logic. In: ESOP. pp. 503–531 (2016)
34. Melham, T.F.: The HOL logic extended with quantification over type variables. In: TPHOLs. pp. 3–17 (1992)
35. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. vol. 2283 of LNCS, Springer (2002)
36. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer (2014)
37. Nipkow, T., Snelting, G.: Type Classes and Overloading Resolution via Order-Sorted Unification. In: Functional Programming Languages and Computer Architecture (1991)
38. Obua, S.: Checking Conservativity of Overloaded Definitions in Higher-Order Logic. In: RTA. Springer (2006)
39. Owre, S., Shankar, N.: The formal semantics of PVS (March 1999), SRI technical report. http://www.csl.sri.com/papers/csl-97-2/
40. Paulson, L.: Personal communication (2014)
41. Pitts, A.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, chap. The HOL Logic, pp. 191–232. In: Gordon and Melham [18] (1993)
42. Reynolds, J.C.: Polymorphism is not set-theoretic. In: Semantics of Data Types. vol. 173, pp. 145–156 (1984)
43. Rushby, J.M., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. IEEE Trans. Software Eng. 24(9), 709–720 (1998)
44. Shankar, N., Owre, S., Rushby, J.M.: PVS Tutorial. Computer Science Laboratory, SRI International (1993)
45. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In: LICS, pp. 596–605 (2012)
46. Wadler, P., Blott, S.: How to Make ad-hoc Polymorphism Less ad-hoc. In: POPL (1989)
47. Wenzel, M.: The Isabelle/Isar reference manual (2016), available at http://isabelle.in.tum.de/doc/isar-ref.pdf
48. Wenzel, M.: Type Classes and Overloading in Higher-Order Logic. In: TPHOLS. pp. 307–322 (1997)
49. Wiedijk, F.: Stateless HOL. In: TYPES. pp. 47–61 (2009)