A Consistent Foundation for Isabelle/HOL

Ondřej Kunčar[1] and Andrei Popescu[2]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Department of Computer Science, School of Science and Technology, Middlesex University, UK

Abstract. The interactive theorem prover Isabelle/HOL is based on 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 achieve *Haskell-like type classes* in the user space. 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 in the nineties as a logic for interactive theorem provers by Mike Gordon, who also implemented the seminal HOL theorem prover [9]. This system has produced many successors and emulators known under the umbrella term "HOL-based provers" (e.g., [2,20,11,4,3]), lounching a very successful paradigm in interactive theorem proving.

A main strength of HOL-based provers is a sweet spot in expressiveness versus complexity: HOL on the one hand is sufficient for most mainstream mathematics and computer science applications, and on the other is a well-understood logic. In particular, the 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 $c \equiv t$, where t is a closed term not containing c
- New types τ are introduced by "typedef" equations $\tau \equiv p$, where $p : \sigma \Rightarrow \mathsf{bool}$ is a predicate on an existing type σ (not containing τ anywhere in the types of its subterms)—intuitively, the type τ is carved out as a subset of σ

Again, this mechanism is manifestly consistent by an immediate semantic argument [23]; alternatively, its consistency can be established by regarding definitions as mere abbreviations (which are non-cyclic by construction).

Polymorphic HOL with Ad Hoc Overloading Isabelle/HOL [20,19] 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 [27] 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 pi to an atom a:

Example 1 consts perm : prm $\Rightarrow \alpha \Rightarrow \alpha$
defs perm_atom: perm pi (a : atom) $\equiv$ apply pi a
defs perm_nat: perm pi (n : nat) $\equiv$ n
defs perm_list: perm pi (xs : α list) $\equiv$ map (perm pi) xs

Above, the constant perm is declared using the keyword "consts"—its intended behavior is the application of a permutation to all atoms contained in an element of a type α. Then, using the keyword "defs", several overloaded definitions of perm are performed for different instances of α. For atoms, perm applies the permutation; for numbers (which don't have atoms), perm is the identity function; for α list, the instance of perm is defined in terms of the instances for the component α. All these definitions are non-overlapping and their type-based recursion is terminating, hence Isabelle is fine with them.

Inconsistency Of course, one may not be able to specify all the relevant instances immediately after declaring a constant like perm—at a later point, a user may define their own atom-container type, such as[3]

```
datatype myTree = Atom atom | LNode atom list | FNode nat => atom
```

and instantiate perm to this type. (In fact, the Nominal package automates instantiations for user-requested datatypes, including terms with bindings.) To support such delayed instantiations, which are crucial for the implementation of type classes [29,10], 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 accepts the following definitions[4]

Example 2 consts c : α
typedef T = {True, c} by blast
defs c_bool_def: c:bool $\equiv \neg$ ($\forall$(x:T) y. x = y)

which immediately yield a proof of False:

lemma L: ($\forall$(x:T) y. x = y) $\leftrightarrow$ c
using Rep_T Rep_T_inject Abs_T_inject by (cases c:bool) force+

theorem False
using L unfolding c_bool_def by auto

[3] In Isabelle/HOL, as in any HOL-based prover, the "datatype" command is not primitive, but is compiled into "typedef."

[4] This example works in Isabelle2014; our correction patch [1] based on the results of this paper and in its predecessor [14] is being evaluated at the Isabelle headquarters.

The inconsistency argument takes advantage of the circularity $T \rightsquigarrow c_{\mathsf{bool}} \rightsquigarrow T$ in the dependency relation introduced by the definitions: one first defines T to contain only one element just in case c_{bool} is True, and then defines c to beTrue just in case T contains more than one element.

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

– A definitional dependency relation that factors in both constant and type definitions in a sensible fashion (§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 (§4)
– A new semantics for polymorphic HOL (§4.4) that 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 (§2). Later we also show how this work fits together with previous work by the first author (§5).

2 Related Work

Type Classes and Overloading Type classes were introduced in Haskell by Wadler and Blott [28]—they allow programmers to write functions that operate 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 an α-list. Then the program can be run on any concrete type T that replaces α provided T has this binary operation $+$. Prover-powered type classes were introduced by Nipkow and Snelting [21] in Isabelle/HOL and by Sozeau and Oury [25] in Coq—they additionally feature verifiability of the type-class conditions upon instantiation: a type T is deemed a member of the semigroup class only if associativity can be proved for its $+$ operation.

Whereas Coq implements type classes directly by virtue of its powerful type system, Isabelle/HOL relies on arbitrary ad hoc overloading: to introduce the semigroup class, the system declares a "global" constant $+ : \alpha \Rightarrow \alpha \Rightarrow \alpha$ and defines the associativity predicate; then different instance types T are registered after defining the corresponding overloaded operation $+ : T \Rightarrow T \Rightarrow T$ and verifying the condition. Our current paper focuses on the consistency of the mechanism of ad hoc overloading, which makes type classes in Isabelle/HOL possible.

Previous Consistency Attempts The settling of this consistency problem has been previously attempted by Wenzel [29] and Obua [22]. In 1997, Wenzel defined a notion of 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:

```
consts c : α ⇒ bool
defs c (x : α list × α) ≡ c (snd x # fst x)
defs c (x : α list) ≡ ¬ c (tl x, hd x)

lemma c [x] = ¬ c([], x) = ¬ c[x]
```

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

Triggered by Obua's observations, Wenzel implemented a simpler and more structural solution based on work of Haftmann, Obua and Urban: fewer overloadings are accepted in order to make the consistency/termination check decidable (which Obua's original check is not). Wenzel's solution has been part of the kernel since Isabelle2007 without any important changes—parts of this solution (which still misses out dependencies through types) are described by Haftmann and Wenzel [10].

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 [14] 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. Our current paper complements this result by showing that termination leads to consistency.

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 [24] are folklore. Coq's [5] current stable version[5] 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 [8]. Agda [7] suffers from similar problems [17] The recent Dafny prover [15] proposes an innovative combination of recursion and corecursion whose initial version turned out to be inconsistent [6].

Of course, such "dangerous" experiments are often motivated by better support for the users' formal developments. The Isabelle/HOL type class experiment was practically successful: substantial developments such as the Nominal [27,13] and HOLCF [18] packages and Isabelle's mathematical analysis library [12] rely heavily on type classes. One of Isabelle's power users writes [16]: "Thanks to type classes and refinement during code generation, our light-weight framework is flexible, extensible, and easy to use."

[5] Namely, Coq 8.4pl5; the inconsistency is reported as being fixed in Coq 8.5 beta.

3 Polymorphic HOL with Ad Hoc Overloading

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

3.1 Syntax

All throughout this section, we fix the following countable sets:

- A set TVar, of *type variables*, ranged over by α, β
- A set Var, of *(term) variables*, ranged over by x, y, z
- A set K of symbols, ranged over by k, called *type constructors*, containing three special symbols: bool, ind and $\Rightarrow$

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 $\mathsf{arOf}(\Rightarrow) = 2$. 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 $\mathsf{arOf}(k) = n$

A *typed variable* is a pair of a 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 the special symbols: $\to$, zero, suc, $=$ and some
- A function $\mathsf{tpOf} : \mathsf{Const} \to \mathsf{Type}$ associating a type to every constant, such that:

$$\begin{array}{ll}
\mathsf{tpOf}(\to) = \mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool} & \mathsf{tpOf}(=) = \alpha \Rightarrow \alpha \Rightarrow \mathsf{bool} \\
\mathsf{tpOf}(\mathsf{zero}) = \mathsf{ind} & \mathsf{tpOf}(\mathsf{some}) = (\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha \\
\mathsf{tpOf}(\mathsf{suc}) = \mathsf{ind} \Rightarrow \mathsf{ind} &
\end{array}$$

A *type-variable substitution* is a function $\rho : \mathsf{TVar} \to \mathsf{Type}$; we let TSubst denote the set of type-variable substitutions. Each $\rho \in \mathsf{TSubst}$ is naturally extended to a 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 $\rho \in \mathsf{TSubst}$, 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 ρ such that $\sigma \leq_\rho \tau$. Two types σ and τ are called orthogonal, written $\sigma \;\#\; \tau$, if they have no common instance.

We define the type variables of a type, $\mathsf{TV} : \mathsf{Type} \to \mathcal{P}(\mathsf{TVar})$, as expected. A type σ is called *ground* if $\mathsf{TV}(\sigma) = \emptyset$. We let GType be the set of ground types and $\mathsf{GTSubst}$ the set of all ground type-variable substitutions $\theta : \mathsf{TVar} \to \mathsf{GType}$. We can again naturally extended each $\theta \in \mathsf{GTSubst}$ to a homonymous function $\theta : \mathsf{Type} \to \mathsf{GType}$.

The tuple $(K, \mathsf{tpOf}, C, \mathsf{arOf})$, which will be fixed in what follows, is called a *signature*. The *terms*, ranged over by t, are defined by the following grammar:

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

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

Typing of terms is defined in the expected way (by assigning the most general type possible); we write Term for the set of well-typed terms and, given $t \in \mathsf{Term}$, we write $\mathsf{tpOf}(t)$ for its (unique) type. Given a term t, we write $\mathsf{FV}(t)$ for the set of its free (term) variables. t is *closed* if $\mathsf{FV}(t) = \emptyset$. We let $\mathsf{types}(t)$ denote the set of types of all constants and (free or bound) variables that occur in t. Then $\mathsf{TV}(t)$, the set of type variables of t, is defined as $\bigcup_{\sigma \in \mathsf{types}(t)} \mathsf{TV}(\sigma)$. A term t is called *type-ground* if $\mathsf{TV}(t) = \emptyset$. Note that being type-ground is a stronger condition than having a ground type: $(\lambda x_\alpha.\ x)\ c_{\mathsf{bool}}$ has the ground type bool, but is not type-ground.

A *formula* is a well-typed term of type bool. We let Fmla, ranged over by φ, ψ, χ, denote the set of formulas. The formula connectives and quantifiers are defined in the standard way, starting from implication and equality primitives.

3.2 Built-Ins and Non-Built-Ins

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

A *built-in type* is any type of the form bool, ind, or $\sigma_1 \Rightarrow \sigma_2$ for $\sigma_1, \sigma_2 \in \mathsf{Type}$. We let $\mathsf{Type}^\bullet$ denote the set of types that are *not* built-in types. Note that a non-built-in type can have a built-in type as a subtype, and vice versa; e.g., if list is a type constructor, then $\mathsf{bool\ list}$ and $(\alpha \Rightarrow \beta)\ \mathsf{list}$ are non-built types, whereas $\alpha \Rightarrow \beta\ \mathsf{list}$ is a built-in type. We let $\mathsf{GType}^\bullet = \mathsf{GType} \cap \mathsf{Type}^\bullet$ denote the set of non-built-in ground types.

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

$$\mathsf{types}^\bullet(\mathsf{bool}) = \mathsf{types}^\bullet(\mathsf{ind}) = \emptyset$$
$$\mathsf{types}^\bullet((\sigma_1, \ldots, \sigma_n)\ k) = \{(\sigma_1, \ldots, \sigma_n)\ k\}, \text{ if } k \text{ is different from } \Rightarrow$$
$$\mathsf{types}^\bullet(\sigma_1 \Rightarrow \sigma_2) = \mathsf{types}^\bullet(\sigma_1) \cup \mathsf{types}^\bullet(\sigma_2)$$

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 prm (0-ary) and list (unary) are in the signature and if σ is $(\mathsf{bool} \Rightarrow \alpha\ \mathsf{list}) \Rightarrow \mathsf{prm} \Rightarrow (\mathsf{bool} \Rightarrow \mathsf{ind})\ \mathsf{list}$, then $\mathsf{types}^\bullet(\sigma)$ has three elements: $\alpha\ \mathsf{list}$, prm and $(\mathsf{bool} \Rightarrow \mathsf{ind})\ \mathsf{list}$.

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

Given $T \subseteq \mathsf{Type}^\bullet$, we define $\mathsf{BIC}(T)$, the *built-in constants* of T to be the set of built-in constants whose types are in $\mathsf{Cl}(T)$.

Given $c \in \mathsf{Const}$ such that $\sigma \leq \mathsf{tpOf}(c)$, we call the pair (c, σ), written c_σ, an *instance of* c. We let CInst be the set of all constant instances, and GCInst be the set of constant instances whose type is ground. We extend $\leq$ to constant instances: let $c_\tau, d_\sigma \in \mathsf{CInst}$, $c_\tau \leq d_\sigma$ iff $c = d$ and $\tau \leq \sigma$. We also extend tpOf to constant instances as $\mathsf{tpOf}(c_\sigma) = \sigma$.

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. More precisely:

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

3.3 Deduction

The notion of (proof) context is defined inductively as follows:

- the empty sequence [] is a context
- if Γ is a context and $\mathsf{fix}(x_\sigma)$ does not appear in Γ, then Γ, x_σ is a context
- if Γ is a context and $\mathsf{fix}(x_\sigma)$ appears in Γ for each $x_\sigma \in \mathsf{FV}(\varphi)$, then Γ, $\mathsf{assume}(\varphi)$ is a context

Thus, a context Γ is a sequence $k_1, \ldots, k_n$ where each k_i represents either a "fixed" variable x_σ or an assumption φ, subject to the following requirements:

- the fixed variables are distinct
- the assumptions have their free variables fixed beforehand

Deduction of a formula φ in a context Γ, written $\Gamma \vdash \varphi$, is defined using the standard polymorphic HOL axioms and deduction rules. If Γ contains only assumptions (and no "fixes"), it is easy to see that all its assumptions are closed formulas and that the order of these assumptions does not affect deduction. Hence, if D is a finite set of closed formulas, a.k.a. a *theory*, we write $D \vdash \varphi$ for the deducibility of φ from D. A theory D is called *consistent* if $D \not\vdash \mathsf{False}$, or equivalently if there exists φ such that $D \not\vdash \varphi$. (Isabelle/HOL distinguishes between theory contexts and proof contexts—we ignore this distinction here, since it does not affect our consistency argument.)

3.4 Definitional Theories

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 $t \in \mathsf{Term}_\sigma$, we let $c_\sigma \equiv t$ denote the formula $c_\sigma = t$. We call $c_\sigma \equiv t$ a *constant definition* provided $\mathsf{TV}(t) \subseteq \mathsf{TV}(c_\sigma)$ (i.e., $\mathsf{TV}(t) \subseteq \mathsf{TV}(\sigma)$). Notice that we do not require that c is a fresh symbol as in the traditional constant definition mechanism without overloading.

Given the types $\tau \in \mathsf{Type}^\bullet$ and $\sigma \in \mathsf{Type}$ and the closed well-typed term p whose type is $\sigma \Rightarrow \mathsf{bool}$, we let $\tau \equiv p$ denote the formula

$$\begin{array}{l}
(\exists x_\sigma.\ p\ x) \rightarrow \\
\quad \exists rep_{\tau \Rightarrow \sigma}.\ \exists abs_{\sigma \Rightarrow \tau}. \\
\qquad (\forall x_\tau.\ p\ (rep\ x)) \ \wedge\ (\forall x_\tau.\ abs\ (rep\ x) = x) \ \wedge\ (\forall y_\sigma.\ p\ y \rightarrow rep\ (abs\ y) = y).
\end{array}$$

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

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

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 definitional theory D
3. other proved theorems

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 are not influenced by the others. Indeed, note we defined $\tau \equiv p$ (where $\mathsf{tpOf}(p) = \sigma \Rightarrow \mathsf{bool}$) *not* to mean:

(*): *The type σ is isomorphic, via abs and rep, to the subset of σ given by p*

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

If p gives a nonempty subset of σ, then (*) holds

After this point, Isabelle/HOL's behavior converges with standard HOL behavior since the user is immediately required to prove that p gives a nonempty subset (i.e., that $\exists x_\sigma.\ p\ x$ holds); then the system infers (*). However, this last inference step is normal deduction, having nothing to do with the definition itself—this very useful trick, due to 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 definitional theory D to be consistent (while allowing flexible overloading).

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: defining $c_{\alpha \Rightarrow \mathsf{bool}}$ to be $\neg\, c_{\mathsf{bool} \Rightarrow \alpha}$ yields $c_{\mathsf{bool} \Rightarrow \mathsf{bool}} = \neg\, c_{\mathsf{bool} \Rightarrow \mathsf{bool}}$. Moreover, it is clear that the definitions need to be orthogonal: defining $c_{\alpha \Rightarrow \mathsf{bool}}$ to be False and $c_{\mathsf{bool} \Rightarrow \alpha}$ to be True yields again $c_{\mathsf{bool} \Rightarrow \mathsf{bool}} = \neg\, c_{\mathsf{bool} \Rightarrow \mathsf{bool}}$.

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 definitional theory D. We say D is *orthogonal* if for all distinct definitions $u \equiv t$ and $u' \equiv t'$ in D, $u\ \#\ u'$.

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

1. there exists a definition in D of the form $u \equiv t$ such that $v \in \mathsf{consts}^\bullet(t) \cup \mathsf{types}^\bullet(t)$
2. there exists no definition in D with u as left-hand side, and there exists $c \in \mathsf{Const}^\bullet$ such that $u = c_{\mathsf{tpOf}(c)}$ and $v \in \mathsf{types}^\bullet(\mathsf{tpOf}(c))$

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

We illustrate the choices behind our definition of $\rightsquigarrow$ through the following example, where the definition of $\alpha\ k$ is omitted:

Example 3 `consts c : α   d : α`
```
typedef α k = ...
consts c (x : ind k ⇒ bool) ≡ (d : bool k k ⇒ ind k ⇒ bool) (d : bool k k)
```

It is clear which constant instances $c_{\mathsf{ind}\ k \Rightarrow \mathsf{bool}}$ depends on after its definition, namely $d_{\mathsf{bool}\ k\ k \Rightarrow \mathsf{ind}\ k \Rightarrow \mathsf{bool}}$ and $d_{\mathsf{bool}\ k\ k}$. But what are the types on which $c_{\mathsf{ind}\ k \Rightarrow \mathsf{bool}}$ depends? A conservative answer would be: all the subtypes of the types of the constants and variables used in the definition, namely: $\mathsf{bool}\ k\ k \Rightarrow \mathsf{ind}\ k \Rightarrow \mathsf{bool}$, $\mathsf{bool}\ k\ k$, $\mathsf{ind}\ k \Rightarrow \mathsf{bool}$, $\mathsf{bool}\ k$, $\mathsf{ind}\ k$, bool and ind.[6]

Some of these dependencies are indeed crucial: if we omit to register that $c_{\mathsf{ind}\ k \Rightarrow \mathsf{bool}}$ depends on any of $\mathsf{bool}\ k\ k$, $\mathsf{ind}\ k$ and bool, we can easily reach an inconsistency in the style of Example 2 by suitable choices of k's definition and d's overloaded definitions. However, the others seem harmless:

– the built-in types ind and bool
– the types built from the "crucial" ones by the function type constructor—any inconsistency achievable using these needs to go through the "crucial" components
– the types only appearing below a non-built-in type constructor, such as $\mathsf{bool}\ k$ (covered by $\mathsf{bool}\ k\ k$)—the only way to reach these is through constants that have them uncovered (i.e., only covered by function types)

In other words, it appears that we only need to record the dependency on the non-built-in types reachable from the occurring types, i.e., those in $\mathsf{types}^\bullet(t)$ where t is the right-hand side of the definition. This intuition will receive a rigorous justification in

[6] The precise granularity of the dependency relation is not relevant here; e.g., it does not matter whether we decide to have $c_{\mathsf{ind}\ k \Rightarrow \mathsf{bool}} \rightsquigarrow \mathsf{bool}\ k\ k$ directly or $c_{\mathsf{ind}\ k \Rightarrow \mathsf{bool}} \rightsquigarrow d_{\mathsf{bool}\ k\ k} \rightsquigarrow \mathsf{bool}\ k\ k$; so in this discussion we interpret "depends on" as $\rightsquigarrow^+$.

our new semantics for HOL, where a polymorphic type will be interpreted as the family of all its ground-instance interpretations, which for non-built-in types will be mutually independent; e.g., the interpretations of $\mathsf{bool}\ k\ k$ and $\mathsf{bool}\ k$ will be unrelated.

4.2 The Consistency Theorem

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

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

Previous attempts to prove consistency employed syntactic methods [29,22]. Instead, we will give a semantic proof:

1. We define a new semantics of Polymorphic HOL, suitable for overloading and for which standard HOL deduction is sound (§4.4)
2. We prove that D has a model according to our semantics (§4.5)

Then 1 and 2 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 [23], 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 $\mathcal{U}$
   - $[\Rightarrow] : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$ 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 similarly:
   - a defined type prm or type constructor list as an element $[\mathsf{prm}] \in \mathcal{U}$ or operator $[\mathsf{list}] : \mathcal{U} \to \mathcal{U}$, produced according to their "typedef"
   - a polymorphic constant such as $\mathsf{perm} : \mathsf{prm} \to \alpha \to \alpha$ as a family $[\mathsf{perm}] \in \prod_{A \in \mathcal{U}} [\mathsf{prm}] \to A \to A$

In standard polymorphic HOL, perm would be either completely unspecified, or completely defined in terms of previously existing constants—this has a faithful semantic counterpart in $\mathcal{U}$. But now how to represent the overloaded definitions of perm from Example 1? In $\mathcal{U}$, they would become:

$$[\mathsf{perm}]_{[\mathsf{atom}]}\ pi\ a = [\mathsf{apply}]\ pi\ a$$
$$[\mathsf{perm}]_{[\mathsf{nat}]}\ pi\ n = n$$
$$[\mathsf{perm}]_{[\mathsf{list}](A)}\ pi\ xs = [\mathsf{map}]_A\ (\mathsf{perm}_A\ pi)\ xs$$

There are two problems with these semantic definitions. First, given $B \in \mathcal{U}$, the value of $[\mathsf{perm}]_B$ is different depending on whether B has the form $[\mathsf{atom}]$, or $[\mathsf{nat}]$, or $[\mathsf{list}](A)$ for some $A \in \mathcal{U}$; 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. 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 equation 3) is a terminating process.

Even worse, termination is in general a global property, possibly involving both constants and type constructors, as shown in the following example where c and k are mutually defined (so that a copy of $e_{\mathsf{bool}\ k^n}$ is in bool k^{n+1} iff n is even):

Example 4 `consts c : α ⇒ bool   d : α   e : α`
```
typedef α k = {d:α} ∪ {e : α . c (d : α)}
c (x : α k) ≡ ¬ c (d : α)
c (x : bool) ≡ True
```

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, thinking of the former as "macros" for families of the latter. For example, $\alpha \Rightarrow \alpha$ list represents the family $(\tau \Rightarrow \tau\ \mathsf{list})_{\tau \in \mathsf{GType}}$.

4.4 Ground Semantics of Polymorphic HOL

We fix a singleton set $\{*\}$ and a global choice function choice that assigns to each nonempty set A an element $\mathsf{choice}(a) \in A$.

A *ground type interpretation* is a family $([\tau])_{\tau \in \mathsf{GType}^\bullet}$ where each $[\tau]$ is a nonempty set. We extend $[\tau]$ inductively to ground types by interpreting the built-in types:

$[\mathsf{bool}] = \{0, 1\}$ (where we think of 0 as "false" and of 1 as "true")
$[\mathsf{ind}] = \mathcal{N}$
$[\sigma \Rightarrow \tau] = [\sigma] \to [\tau]$ (the set of functions from $[\sigma]$ to $[\tau]$)

An *interpretation* is a pair $\mathcal{I} = (([\tau])_{\tau \in \mathsf{GType}^\bullet}, ([c_\sigma])_{c_\sigma \in \mathsf{GCInst}^\bullet})$ such that the family $([\tau])_{\tau \in \mathsf{GType}^\bullet}$ is a type interpretation and $[c_\sigma] \in [\sigma]^{\mathcal{I}}$ for every $[c_\sigma]$ from the family $([c_\sigma])_{c_\sigma \in \mathsf{GCInst}^\bullet}$, which we call a *ground constant interpretation*.

We extend $[c_\sigma]$ to ground constant instances by interpreting the built-in constants:

– $[\longrightarrow_{\mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}}]^{\mathcal{I}}$ is the logical implication on $\{0, 1\}$

- $[=_{\sigma\Rightarrow\sigma\Rightarrow\mathsf{bool}}]^{\mathcal{I}}$ is the equality predicate of the type $[\sigma]^{\mathcal{I}} \to [\sigma]^{\mathcal{I}} \to \{0,1\}$
- $[\mathsf{zero}_{\mathsf{ind}}]^{\mathcal{I}}$ is zero for $\mathcal{N}$ and $[\mathsf{suc}_{\mathsf{ind}\Rightarrow\mathsf{ind}}]$ is the successor function for $\mathcal{N}$
- $[\mathsf{some}_{(\sigma\Rightarrow\mathsf{bool})\Rightarrow\sigma}]^{\mathcal{I}}$ is defined as follows, where, for each $f : [\sigma]^{\mathcal{I}} \to \{0,1\}$, $A_f = \{a \in [\sigma]^{\mathcal{I}} \mid f(a) = 1\}$:
  $[\mathsf{some}_{(\sigma\Rightarrow\mathsf{bool})\Rightarrow\sigma}]^{\mathcal{I}}(f) = \begin{cases} \mathsf{choice}(A_f) & \text{, if } A_f \text{ is non-empty} \\ \mathsf{choice}([\sigma]^{\mathcal{I}}) & \text{, otherwise} \end{cases}$

A valuation $\xi : \mathsf{Var}_{\mathsf{GType}} \to \mathcal{S}et$ is called *compatible* if $\xi(x_\sigma) \in [\sigma]^{\mathcal{I}}$ for each $x_\sigma \in \mathsf{Var}_{\mathsf{GType}}$. We write $\mathsf{Comp}^{\mathcal{I}}$ for the set of compatible valuations.

Given an interpretation $\mathcal{I}$, we extend an interpretation of ground constant instances $[c_\sigma]^{\mathcal{I}}$ to an interpretation of ground terms: For each $t \in \mathsf{GTerm}$, $[t]^{\mathcal{I}}$ is a function

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

defined recursively over terms as follows (we write $[.]^{\mathcal{I}}_{\xi}$ for $[.]^{\mathcal{I}}(\xi)$):

$[x_\sigma]^{\mathcal{I}}_{\xi} = \xi(x_\sigma)$
$[c_\sigma]^{\mathcal{I}}_{\xi} = [c_\sigma]^{\mathcal{I}}$
$[t_1\ t_2]^{\mathcal{I}}_{\xi} = [t_1]^{\mathcal{I}}_{\xi}\ [t_2]^{\mathcal{I}}_{\xi}$
$[\lambda x_\sigma.\ t]^{\mathcal{I}}_{\xi}$ is the function sending each $a \in [\sigma]^{\mathcal{I}}$ to $[t]^{\mathcal{I}}_{\xi[x_\sigma\leftarrow a]}$

Lemma 5. For each $t \in \mathsf{GTerm}$, $[t]^{\mathcal{I}}$ is a function that only depends on the restriction of its inputs to $\mathsf{FV}(t)$, which means $\forall \xi, \xi' \in \mathsf{Comp}^{\mathcal{I}}.\ \xi =_{\mathsf{FV}(t)} \xi' \to [t]^{\mathcal{I}}_{\xi} = [t]^{\mathcal{I}}_{\xi'}$.

Given Γ, φ, $\theta \in \mathsf{GTSubst}$ and $\xi \in \mathsf{Comp}^{\mathcal{I}}$, we say that $\mathcal{I}$ *satisfies* (Γ, φ) *under the valuations* θ *and* ξ, written $\mathcal{I} \models_{\theta,\xi} (\Gamma, \varphi)$, if $[\theta(\varphi)]^{\mathcal{I}}_{\xi} = 1$ whenever $[\theta(\psi)]^{\mathcal{I}}_{\xi} = 1$ for all assumptions $\psi \in \Gamma$.

We say that $\mathcal{I}$ *satisfies* (Γ, φ), written $\mathcal{I} \models (\Gamma, \varphi)$, if $\mathcal{I} \models_{\theta,\xi} (\Gamma, \varphi)$ for all $\theta \in \mathsf{GTSubst}$ and $\xi \in \mathsf{Comp}^{\mathcal{I}}$. If $\Gamma = []$, we write simply write $\mathcal{I} \models_{\theta,\xi} \varphi$ and $\mathcal{I} \models \varphi$. Finally, if E is a set of formulas, $\mathcal{I} \models E$ is defined as $\mathcal{I} \models \varphi$ for all $\varphi \in E$.

It is routine to verify that deduction is sound with respect to the semantics:

Theorem 2. (Soundness) If $\Gamma \vdash \varphi$, then $\mathcal{I} \models (\Gamma, \varphi)$ for all interpretations $\mathcal{I}$.

4.5 The Model Construction

We say that a definitional theory D has a *model* if there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models D$. In this section, we fix a well-formed D and construct its model.

The construction will be in principle guided by the dependency relation introduced in Section 4.1. Let us recall that in order to provide an interpretation, we have to define a meaning of all $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, i.e., how to interpret u. In order to do so, we employ a *ground dependency relation* defined as $\rightsquigarrow_G = (\rightsquigarrow^{\downarrow}) \restriction_{\mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet}$. This $\rightsquigarrow_G$ gives us an order in which we should define an interpretation of $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$ because for each u, $u \rightsquigarrow_G v$ identifies the syntactic items v that are needed to define u.

Notice that well-foundedness of D guarantees termination of $\rightsquigarrow_G$.

Lemma 6. $\rightsquigarrow_G$ terminates iff $\rightsquigarrow^{\downarrow}$ terminates.

Proof. Sufficiency: $\rightsquigarrow^{\downarrow} \supseteq (\rightsquigarrow^{\downarrow}) \restriction_{\mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet} = \rightsquigarrow_G$. Necessity: If $\rightsquigarrow^{\downarrow}$ does not terminate, there exists a non-terminating sequence, which we can make ground by substituting bool for all type variables in it. But then $\rightsquigarrow_G$ does not terminate either.

Thus, we can construct a model for D by well-founded recursion on $\rightsquigarrow_G$ when we extend a (partial) interpretation $\mathcal{I}$ for bigger and bigger $U \subseteq \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$. On this account, we need a formal notion of a signature fragment and a partial interpretation. Moreover, we would need to merge partial interpretations from the recursion for all vs that u depends on in order to construct an interpretation for u. Thus, we will work only with strict interpretations that provide such conflict-free merges. We will define the notions of partial and strict interpretations in the next two subsections.

Partial Interpretations Recall that we have a fixed signature $(K, \mathsf{arOf}, \mathsf{Const}, \mathsf{tpOf})$, from which we derive the set of ground non-built-in types $\mathsf{GType}^\bullet$ and the set of ground non-built-in constant instances $\mathsf{GCInst}^\bullet$. For each term t, we will be interested in certain signature fragments that are sufficient for interpreting t semantically.

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 let Type^F denote the set of types generated by this fragment, namely $\mathsf{Cl}(T)$. We let Term^F denote the set of well-typed terms that fall within this fragment, i.e., such that $\mathsf{types}^\bullet(t) \subseteq T$ and $\mathsf{consts}^\bullet(t) \subseteq C$.

lemmafragpropsr If $t \in \mathsf{Term}^F$, then $\mathsf{tpOf}(t) \in \mathsf{Type}^F$.

Note that fragments $F = (T, C)$ are self-contained entities, since the types of all constant instances in C can be constructed from T and the built-in type constructors. Since the interpretation of the built-in type-constructors and constants is fixed, F-interpretation will have all the ingredients to interpret the terms in Term^F.

Formally, an *F-interpretation* is a pair $\mathcal{I} = (([\sigma])_{\sigma \in T}, ([c_\tau])_{c_\tau \in C})$ such that each $[\sigma]$ is a non-empty set and $[c_\tau] \in [\tau]^{\mathcal{I}}$ for all $c_\tau \in C$. Similarly to the case of (total) interpretations, we can recursively extend any F-interpretation $\mathcal{I}$ to the interpretation $[t]^{\mathcal{I}} : (\mathsf{Var}_{\mathsf{Type}^F} \to \mathcal{S}et) \to [\mathsf{tpOf}(t)]$ for all $t \in \mathsf{Term}^F$. Note that a (total) interpretation is a particular case of an F-interpretation, where $F = (\mathsf{GType}^\bullet, \mathsf{GCInst}^\bullet)$.

Given $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, we define $u \in F = (T, C)$ to mean that $u \in T$ or $u \in C$. Union, interjection and set difference for fragments are defined componentwise. Given a family of fragments $(F_j)_{j \in J}$, their union $\bigcup_{j \in J} F_j$ is clearly again a fragment.

Let $\mathcal{I}_1$ be an F_1-interpretation and $\mathcal{I}_2$ an F_2-interpretation $\mathcal{I}_2$. $\mathcal{I}_1$ and $\mathcal{I}_2$ are said to be *compatible* if they coincide on items in $F_1 \cap F_2$, i.e., $[u]^{\mathcal{I}_1} = [u]^{\mathcal{I}_2}$ for all $u \in F_1 \cap F_2$. Given a family $(F_j)_{j \in J}$ of fragments and a family $(\mathcal{I}_j)_{j \in J}$ of mutually compatible interpretations for them, i.e., such that each $\mathcal{I}_j$ is an F_j-interpretation and any two $\mathcal{I}_j$ and $\mathcal{I}_{j'}$ are compatible, we define the union $\mathcal{I} = \bigcup_{j \in J} \mathcal{I}_j$ as expected, by setting $[u]^{\mathcal{I}} = [u]^{\mathcal{I}_j}$ whenever $u \in F_j$.

Lemma 7. $\bigcup_{j \in J} \mathcal{I}_j$ is a $(\bigcup_{j \in J} F_j)$-interpretation.

In order to define the meaning of $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, we need a notion of a definition prescribing the meaning of u (in case there is such a definition). Let D be orthogonal and $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, we define a *$u$-definition* $\mathsf{def}(u)$ as follows:

– If there exits $u' \equiv t$ in D such that $u \leq_\rho u'$, $\mathsf{def}(u) = (u \equiv \rho(t))$.
– Otherwise, $\mathsf{def}(u) = \mathsf{True}$.

$\mathsf{def}(u)$ is well-defined because due to orthogonality, there exists at most one matching definition $u' \equiv t$. Notice that due to definitionality $\rho(t) \in \mathsf{GTerm}$.

Although we defined fragments generally, we will be mainly interested in one specific type of them—in fragments containing vs that u depends on: If $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, we define F^u, the *definitional fragment associated to u*, to be (T^u, C^u), where:

– $T^u = \{\tau \in \mathsf{GType}^\bullet \mid u \rightsquigarrow_G^* \tau\}$
– $C^u = \{c_\sigma \in \mathsf{GCInst}^\bullet \mid u \rightsquigarrow_G^* c_\sigma\}$

The following lemma captures our intuition which syntactic material is necessary in order to define the meaning of a type or a constant instance u. lemmasyntdepthm Let $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, $V = \{v \mid u \rightsquigarrow_G v\}$ and $F = \bigcup_{v \in V} F^v$.

1. If $u = c_\sigma$, then $\sigma \in \mathsf{Type}^F$
2. If $\mathsf{def}(u) = (u \equiv t)$, then $t \in \mathsf{Term}^F$.

Proof. From the definition of $\rightsquigarrow$, well-formedness of D and that F^u is a fragment. □

Strict Interpretations Recall that we have a fixed singleton set $\{*\}$ and a fixed choice function choice on sets. Given a fragment F, an F-interpretation $\mathcal{I}$ is called *strict* if the following hold:

– If $\tau \in F$ and $\mathsf{def}(\tau) = \mathsf{True}$, i.e., there exists no matching definition for τ in D, then $[\tau]^\mathcal{I} = \{*\}$.
– If $c_\sigma \in F$ and $\mathsf{def}(c_\sigma) = \mathsf{True}$, then $[c_\sigma]^\mathcal{I} = \mathsf{choice}([\sigma]^\mathcal{I})$.
– If $u \in F$ and $\mathsf{def}(u) = (u \equiv t)$, then $\mathcal{I} \models \mathsf{def}(u)$

Notice that any (total) strict interpretation is a model. lemmagroundmodel Let D be an orthogonal, definitional theory. $\mathcal{I} \models \{\mathsf{def}(u) \mid u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet\}$ iff $\mathcal{I} \models D$. As we already mentioned, strict interpretations are useful because they will allow us to merge interpretations from different branches in the well-founded recursion:

Lemma 8. Let $\rightsquigarrow_G$ be terminating and $U \subseteq \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$ and let us have families $(\mathcal{I}_u)_{u \in U}$ and $(F^u)_{u \in U}$, where $\mathcal{I}_u$ is a strict interpretation for F^u for all $u \in U$. Then $\bigcup_{u \in U} \mathcal{I}_u$ is a strict interpretation for $\bigcup_{u \in U} F^u$.

Proof. Observe that each F^u has at most one strict F^u-interpretation, from which we obtain that all $\mathcal{I}_u$ are pairwise compatible. Apply Lemma 7 and the fact that the strictness is clearly preserved under union. □

The Model Construction Now we are ready to state the main result of this section:

Lemma 9. If D is well-formed, then D has a model.

Proof. For each $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, we define $\mathcal{I}_u$, a strict F^u-interpretation, by well-founded recursion on $\rightsquigarrow_G$, which terminates by Lemma 6 and well-foundedness of D. Let $V = \{v \mid u \rightsquigarrow_G v\}$ and let us obtain $(\mathcal{I}_v)_{v\in V}$ and $(F^v)_{v\in V}$ from the induction hypothesis, such that $\mathcal{I}_v$ is a strict interpretation for F^v for all $v \in V$. Let us define $\mathcal{I}_F = \bigcup_{v\in V} \mathcal{I}_v$ and $F = \bigcup_{v\in V} F^v$. $\mathcal{I}_F$ is a strict F-interpretation by Lemma 8.

We define $\mathcal{I}_u$, an interpretation of F^u, as an extension of $\mathcal{I}_F$. There are three cases:

1. $\mathsf{def}(u) = \mathsf{True}$, i.e., no matching definition for u in D exists. We set $[u]^{\mathcal{I}_u}$ to
   - the singleton set $\{*\}$ if $u \in \mathsf{GType}^\bullet$,
   - $\mathsf{choice}([\sigma]^{\mathcal{I}_F})$, if $u = c_\sigma \in \mathsf{GCInst}^\bullet$.
2. $u = c_\sigma \in \mathsf{GCInst}^\bullet$ and $\mathsf{def}(c_\sigma) = (c_\sigma \equiv t)$. We set $[c_\sigma]^{\mathcal{I}_u} = [t]^{\mathcal{I}_F}$.
3. $u = \tau \in \mathsf{GType}^\bullet$, $\mathsf{def}(\tau) = (\tau \equiv p)$. Let σ be a type such that $p \in \mathsf{Term}^F_\sigma$:
   - If $[\exists x_\sigma.\ p\ x]^{\mathcal{I}_F} = 0$, we set $[\tau]^{\mathcal{I}_u} = \{*\}$.
   - If $[\exists x_\sigma.\ p\ x]^{\mathcal{I}_F} = 1$, we set $[\tau]^{\mathcal{I}_u} = \{a \in [\sigma]^{\mathcal{I}_F} \mid [p]^{\mathcal{I}_F}\ a = 1\}$.

$\mathcal{I}_u$ is an F^u-interpretation: $u \notin F$, otherwise $\rightsquigarrow_G$ would not terminate. $F^u \setminus F = \{u\}$, therefore we can obtain $\mathcal{I}_u$ as an extension of $\mathcal{I}_F$ by defining only $[u]^{\mathcal{I}_u}$. The key step: $[u]^{\mathcal{I}_u}$ is well-defined because $\sigma \in \mathsf{Type}^F$ (by Lemma 7.1) in the case 1. and $\sigma \in \mathsf{Type}^F$ and $p, t \in \mathsf{Term}^F$ (by Lemma 4.5 and 7.2) in the case 2. and 3. Finally, $[u]^{\mathcal{I}_u}$ is always non-empty if u is a type and $[u]^{\mathcal{I}_u} \in [\sigma]^{\mathcal{I}_u}$ if $u = c_\sigma$. Moreover, $\mathcal{I}_u$ is strict since $\mathcal{I}^F$ is strict and $[u]^{\mathcal{I}_u}$ was defined strictly.

$(\mathcal{I}_u)_{u\in\mathsf{GType}^\bullet\cup\mathsf{GCInst}^\bullet}$ is a family of strict interpretations. Let $\mathcal{I}$ denote the union of this family. By Lemma 8, $\mathcal{I}$ is a strict interpretation for $\mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$ and hence $\mathcal{I} \models \mathsf{def}(u)$ for all $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$. Finally Lemma 4.5 tells us that $\mathcal{I}$ is the desired model for D. □

5 Deciding Well-Formedness

We proved that every well-formed theory is consistent. From the implementation perspective, we can ask ourselves how difficult it is to check that the given theory is well-formed. We can check that D is definitional and orthogonal by simple polynomial algorithms. On the other hand, Obua [22] 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 [14] presented the following approach: let us impose a syntactic restriction on accepted overloaded definitions such that the termination of the dependency relation becomes decidable but the restriction still permits all use cases of overloading in Isabelle. The restriction is composability: Let $\rightsquigarrow\!\!\!\!\rightsquigarrow$ be a substitutive and transitive closure[7] of the dependency relation $\rightsquigarrow$, 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 $u \rightsquigarrow\!\!\!\!\rightsquigarrow v$, u' and v do not have a non-trivial common instance, i.e., there does not exist w such that $w \leq u'$, $w \leq v$ and $u' \not\leq v$ and $v \not\leq u'$.

[7] set $\rightsquigarrow\!\!\!\!\rightsquigarrow$ to $\rightsquigarrow^{\downarrow+}$ and prove that this is a substitutive relation

The paper [14] presents a quadratic algorithm (in the size of $\rightsquigarrow$) CHECK that checks that D is definitional, orthogonal and composable, and that $\twoheadrightarrow$ terminates.[8]

Theorem 3. The property of D of being composable and well-formed is decidable.

Proof. Notice that $\twoheadrightarrow = \rightsquigarrow^{\downarrow+}$ terminates iff $\rightsquigarrow^{\downarrow}$ terminates. Thus, CHECK decides whether D is composable and well-formed. □

For efficiency reasons, we optimize 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 its definition. Since $\rightsquigarrow_1^{\downarrow}$ is terminating iff $\rightsquigarrow^{\downarrow}$ is terminating, it suffices to check whether the transitive and substitutive closure of $\rightsquigarrow_1$ terminates.

6 Conclusion

We have provided a solution to the consistency problem for Isabelle/HOL's logic, namely Polymorphic HOL with Ad Hoc Overloading. Consistency is an important, but rather weak property—a suitable notion of conservativeness (perhaps in the style of Wenzel [29], but covering type definitions as well) is left as future work. Independently of Isabelle/HOL, our results show that Gordon-style type definitions and ad hoc overloading can be soundly combined and naturally interpreted semantically.

References

1. `http://www21.in.tum.de/~kuncar/documents/patch.html`
2. The HOL4 Theorem Prover, http://hol.sourceforge.net/
3. Adams, M.: Introducing HOL Zero (extended abstract). In: ICMS '10. Springer (2010)
4. Arthan, R.D.: Some Mathematical Case Studies in ProofPower–HOL. In: TPHOLs 2004
5. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
6. Blanchette, J.C., Popescu, A., Traytel, D.: Foundational Extensible Corecursion, draft available at `www.eis.mdx.ac.uk/staffpages/andreipopescu/pdf/fouco.pdf`
7. Bove, A., Dybjer, P., Norell, U.: A Brief Overview of Agda—A Functional Language with Dependent Types. In: TPHOLs 2009
8. Dénès, M.: [Coq-Club] Propositional extensionality is inconsistent in Coq, archived at `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html`
9. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)
10. Haftmann, F., Wenzel, M.: Constructive Type Classes in Isabelle. In: TYPES (2006)
11. Harrison, J.: HOL Light: A Tutorial Introduction. In: FMCAD '96. Springer (1996)
12. Hölzl, J., Immler, F., Huffman, B.: Type classes and filters for mathematical analysis in isabelle/hol. In: ITP '13
13. Huffman, B., Urban, C.: Proof pearl: A new foundation for Nominal Isabelle. In: ITP '10
14. Kunčar, O.: Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants. CPP '15, ACM (2015)

[8] In fact, the algorithm checks if $\rightsquigarrow$ is acyclic, which is under composability equivalent to the question whether $\twoheadrightarrow$ terminates.

15. Leino, K.R.M., Moskal, M.: Co-induction simply—automatic co-inductive proofs in a program verifier. In: FM 2014
16. Lochbihler, A.: Light-weight containers for isabelle: Efficient, extensible, nestable. In: ITP '13
17. 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`
18. Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF = HOL + LCF. J. Funct. Program. 9, 191–223 (1999)
19. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer (2014)
20. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, vol. 2283 (2002)
21. Nipkow, T., Snelting, G.: Type classes and overloading resolution via order-sorted unification. In: Functional Programming Languages and Computer Architecture (1991)
22. Obua, S.: Checking Conservativity of Overloaded Definitions in Higher-Order Logic. In: RTA. Springer (2006)
23. Pitts, A.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, chap. The HOL Logic, pp. 191–232. In: Gordon and Melham [9] (1993)
24. Shankar, N., Owre, S., Rushby, J.M.: PVS Tutorial. Computer Science Laboratory, SRI International (1993)
25. Sozeau, M., Oury, N.: First-class type classes. In: TPHOLs 2008
26. Traytel, D.: [Agda] Agda's copatterns incompatible with initial algebras, archived at `https://lists.chalmers.se/pipermail/agda/2014/006759.html`
27. Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reason. 40(4) (2008)
28. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: POPL (1989)
29. Wenzel, M.: Type Classes and Overloading in Higher-Order Logic. In: TPHOLS '97

APPENDIX

This appendix gives more technical details about our constructions and proofs. It is only included for the reviewers' convenience. In case the paper is accepted, the appendix will be made part of a technical report available online and cited from the paper.

A Convention and Notations

Given $f, f' : A \to B$ and $C \subseteq A$, we write $f =_C f'$ to indicate that f and f' coincide on C, namely $\forall a \in C.\ f(a) = f'(a)$. $f[a \to b]$ is the function that sends each $a' \neq a$ to $f(a')$ and a to b.

The image of a function under a set is defined as $f[A] = \{f(x) \mid x \in A\}$. If $f : A \to B$ and $C \subseteq A$, the restriction of f to C is a function $f \upharpoonright_C : C \to B$ defined as $f \upharpoonright_C (x) = f(x)$ for all $x \in C$.

A restriction of a relation R to a set X, written $R \upharpoonright_X$, is defined as $R \upharpoonright_X = \{(x, y) \in R \mid x \in X \wedge y \in X\}$.

B More Details on Polymorphic HOL

B.1 Typing of Terms in Polymorphic HOL

We define the sets Term_σ, the set of terms of type σ, inductively as follows:

$x_\sigma \in \mathsf{Term}_\sigma$
$c_\sigma \in \mathsf{Term}_\sigma$
if $t_1 \in \mathsf{Term}_{\sigma_1 \Rightarrow \sigma_2}$ and $t_2 \in \mathsf{Term}_{\sigma_1}$, then $t_1\ t_2 \in \mathsf{Term}_{\sigma_2}$
if $t \in \mathsf{Term}_{\sigma_2}$, then $\lambda x_{\sigma_1}.\ t \in \mathsf{Term}_{\sigma_1 \Rightarrow \sigma_2}$

We let $\mathsf{Term} = \bigcup_{\sigma \in \mathsf{Type}} \mathsf{Term}_\sigma$ be the set of all well-typed terms. We once more extend $\theta \in \mathsf{Type} \to \mathsf{Type}$ naturally to a homonymous function $\theta : \mathsf{Term} \to \mathsf{Term}$ by applying θ to types in the term 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.

Lemma 10. *1. If $\rho \in \mathsf{TSubst}$, $t \in \mathsf{Term}_\sigma$, then $\rho(t) \in \mathsf{Term}_{\rho(\sigma)}$.*
2. $t \in \mathsf{Term}_\sigma$ and $t \in \mathsf{Term}_{\sigma'}$, then $\sigma = \sigma'$.
3. If $\rho, \rho' \in \mathsf{TSubst}$, $t \in \mathsf{Term}$ and $\rho =_{\mathsf{TV}(t)} \rho'$, then $\rho(t) = \rho'(t)$.

In light of Lemma 10.2, we extend tpOf to $\mathsf{tpOf} : \mathsf{Term} \to \mathsf{Type}$ such that $\mathsf{tpOf}(t)$ gives the unique type σ of t.

Lemma 11. Cl is monotone, i.e., if $A \subseteq B$, then $\mathsf{Cl}(A) \subseteq \mathsf{Cl}(B)$.

Proof. By induction on the inductive definition of Cl.

B.2 Other definitions

The operator giving the types of a term is defined recursively as follows:

$$\begin{aligned}
&\mathsf{types}(x_\sigma) = \mathsf{types}(c_\sigma) = \{\sigma\}\\
&\mathsf{types}(t_1\ t_2) = \mathsf{types}(t_1) \cup \mathsf{types}(t_2)\\
&\mathsf{types}(\lambda x_\sigma.\ t) = \{\sigma\} \cup \mathsf{types}(t)
\end{aligned}$$

B.3 More On Non-Built-In Constants and Types

All the next lemmas follow easily by structural induction on types or terms:

Lemma 12. 1. If $c_\sigma \in \mathsf{consts}^\bullet(t)$, then $\sigma \in \mathsf{Cl}(\mathsf{types}^\bullet(t))$
2. If $t \in \mathsf{Term}_\sigma$, then $\sigma \in \mathsf{Cl}(\mathsf{types}^\bullet(t))$

Lemma 13. Let $\rho \in \mathsf{TSubst}$.

1. Let $\sigma \in \mathsf{Type}$. If $\mathsf{types}^\bullet(\sigma) = T$, then $\mathsf{types}^\bullet(\rho(\sigma)) = \rho[T]$.
2. Let $t \in \mathsf{Term}$. If $\mathsf{types}^\bullet(t) = T$, then $\mathsf{types}^\bullet(\rho(t)) = \rho[T]$.

B.4 Deduction System for Polymorphic HOL

When spelling out concrete terms, we take the following conventions:

– we apply the constants $\rightarrow$ and $=$ in an infix manner, e.g., we shall write $t_1 \rightarrow t_2$ instead of $\rightarrow\ t_1\ t_2$.
– we omit redundantly indicating the types of the variables, e.g., we shall write $\lambda x_\sigma.\ x$ instead of $\lambda x_\sigma.\ x_\sigma$ and $x_\sigma\ (y_\tau\ x)$ instead of $x_\sigma\ (y_\tau\ x_\sigma)$.
– We write $\lambda x_\sigma\ y_\tau.\ t$ instead of $\lambda x_\sigma.\ \lambda y_\tau.\ t$

We define the following terms:

$$\begin{aligned}
\mathsf{True} &= (\lambda x_{\mathsf{bool}}.\ x) = (\lambda x_{\mathsf{bool}}.\ x)\\
\mathsf{All} &= \lambda p_{\alpha\Rightarrow\mathsf{bool}}.\ (p = (\lambda x_\alpha.\ \mathsf{True}))\\
\mathsf{Ex} &= \lambda p_{\alpha\Rightarrow\mathsf{bool}}.\ \mathsf{All}(\lambda q_{\alpha\Rightarrow\mathsf{bool}}.\ (\lambda x_\alpha.\ p\ x \rightarrow q) \rightarrow q)\\
\mathsf{False} &= \mathsf{All}(\lambda p_{\mathsf{bool}}.\ p)\\
\mathsf{not} &= \lambda p_{\mathsf{bool}}.\ p \rightarrow \mathsf{False}\\
\mathsf{and} &= \lambda p_{\mathsf{bool}}\ q_{\mathsf{bool}}.\ \mathsf{All}(\lambda r_{\mathsf{bool}}.\ (p \rightarrow q \rightarrow r) \rightarrow r)\\
\mathsf{or} &= \lambda p_{\mathsf{bool}}\ q_{\mathsf{bool}}.\ \mathsf{All}(\lambda r_{\mathsf{bool}}.\ (p \rightarrow r) \rightarrow (q \rightarrow r) \rightarrow r)\\
\mathsf{Ex1} &= \lambda p_{\alpha\Rightarrow\mathsf{bool}}.\ \mathsf{Ex}(\lambda x_\alpha.\ \mathsf{and}\ (p\ x)\ (\mathsf{All}\ (\lambda y_\alpha.\ p\ y \rightarrow y = x)))
\end{aligned}$$

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

– $\mathsf{tpOf}(\mathsf{True}) = \mathsf{tpOf}(\mathsf{False}) = \mathsf{bool}$
– $\mathsf{tpOf}(\mathsf{not}) = \mathsf{bool} \Rightarrow \mathsf{bool}$
– $\mathsf{tpOf}(\mathsf{and}) = \mathsf{tpOf}(\mathsf{or}) = \mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}$
– $\mathsf{tpOf}(\mathsf{All}) = \mathsf{tpOf}(\mathsf{Ex}) = \mathsf{tpOf}(\mathsf{Ex1}) = \alpha \Rightarrow \mathsf{bool}$

As customary, we shall write:

$\mathsf{the}\ x_\alpha.\ t$ instead of $\mathsf{the}(\lambda x_\alpha.\ t)$	$\mathsf{some}\ x_\alpha.\ t$ instead of $\mathsf{some}(\lambda x_\alpha.\ t)$
$\neg\ \varphi$ instead of $\mathsf{not}\ \varphi$	$\forall x_\alpha.\ t$ instead of $\mathsf{All}(\lambda x_\alpha.\ t)$
$\varphi \wedge \chi$ instead of $\mathsf{and}\ \varphi\ \chi$	$\exists x_\alpha.\ t$ instead of $\mathsf{Ex}(\lambda x_\alpha.\ t)$
$\varphi \vee \chi$ instead of $\mathsf{or}\ \varphi\ \chi$	$\exists! x_\alpha.\ t$ instead of $\mathsf{Ex1}(\lambda x_\alpha.\ t)$

We consider the following formulas, where $t_1 \in \mathsf{Type}_{\alpha_1}$ and $t_2 \in \mathsf{Type}_{\alpha_2}$:

$$\begin{array}{lcl}
\mathsf{beta} & = & (\lambda x_{\alpha_2}.\ t_1)\ t_2 = t_1[t_2/x_{\alpha_2}] \\
\mathsf{refl} & = & x_\alpha = x \\
\mathsf{subst} & = & x_\alpha = y_\alpha \to P_{\alpha\to\mathsf{bool}}\ x \to P_{\alpha\to\mathsf{bool}}\ y \\
\mathsf{iff} & = & (p \to q) \to (q \to p) \to (p = q) \\
\mathsf{True\_or\_False} & = & \mathsf{True} \vee \mathsf{False} \\
\mathsf{the\_eq} & = & (\mathsf{the}\ x_\alpha.\ x = a) = a \\
\mathsf{some\_intro} & = & x_\alpha.\ p_{\alpha\Rightarrow\mathsf{bool}}\ x \to p\ (\mathsf{some}\ p) \\
\mathsf{suc\_inj} & = & \mathsf{suc}\ x_{\mathsf{ind}} = \mathsf{suc}\ y_{\mathsf{ind}} \to x = y \\
\mathsf{suc\_not\_zero} & = & \neg\ (\mathsf{suc}\ x_{\mathsf{ind}} = \mathsf{zero})
\end{array}$$

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

Since boolean-typed variables are themselves formula, it is important to formally distinguish between fixed variables and formulas, hence the labels fix and assume. However, to ease notation, in what follows we omit the labels when the intended meaning is clear from our notation, e.g., writing x_σ, φ, y_τ for $\mathsf{fix}(x_\sigma)$, $\mathsf{assume}(\varphi)$, $\mathsf{fix}(y_\tau)$. Moreover, even though technically Γ is a sequence (i.e., a list), we shall use set theoretic notations, e.g., writing $x_\sigma \in \Gamma$ for "$\mathsf{fix}(x_\sigma)$ appears in Γ", etc.

We define *deduction* as a relation $\vdash$ between contexts and formulas as follows:

$$\frac{\cdot}{\Gamma \vdash \varphi}\ (\mathsf{Axiom})\ [\varphi \in \mathsf{Ax}, \mathsf{FV}(\varphi) \subseteq \Gamma] \qquad \frac{\cdot}{\Gamma \vdash \varphi}\ (\mathsf{Assum})\ [\varphi \in \Gamma] \qquad \frac{\Gamma, \varphi \vdash \chi}{\Gamma \vdash \varphi \to \chi}(\mathsf{Impl})$$

$$\frac{\Gamma \vdash \varphi \to \chi \quad \Gamma \vdash \varphi}{\Gamma \vdash \chi}(\mathsf{MP}) \qquad \frac{\Gamma, x_\sigma \vdash f_{\sigma\to\tau}\ x_\sigma = g_{\sigma\to\tau}\ x_\sigma}{\Gamma \vdash f_{\sigma\to\tau} = g_{\sigma\to\tau}}\ (\mathsf{Ext})\ [x_\sigma \notin \Gamma]$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash (\varphi[\overline{\sigma}/\overline{\alpha}])[\overline{t}/\overline{x_\tau}]}\ (\mathsf{INST})\ [\overline{\alpha} \notin \Gamma; \overline{x_\tau} \notin \Gamma; t_i \in \mathsf{Term}_{\tau_i}]$$

The substitution operator _[_/_] for types (and terms) is the usual simultaneous substitution with renaming of bound variables if they get identified (or captured).[9]

C More Details on the Construction of the Model

We fix D, a well-formed theory, in this section.

We prove Lemma 4.5. *

Proof. Let $F = (T, C)$. If $t \in \mathsf{Term}^F$, then $\mathsf{types}^\bullet(t) \subseteq T$. Recall that $\mathsf{Type}^F = \mathsf{Cl}(T)$ and apply Lemmas 11 and 12.2.

[9] Isabelle uses De Bruijn indices to represent bound variables therefore no renaming is needed. But this is just an implementation detail.

Lemma 14. Let $u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$ and $V = \{v \mid u \rightsquigarrow_G v\}$.

1. If $u = c_\sigma$ then $\mathsf{types}^\bullet(c_\sigma) \subseteq V$.
2. If $\mathsf{def}(u) = (u \equiv t)$, then $\mathsf{types}^\bullet(t) \subseteq V$ and $\mathsf{consts}^\bullet(t) \subseteq V$.

Proof. 1. Let $\sigma' = \mathsf{tpOf}(c)$. The equality $\mathsf{types}^\bullet(\sigma') = \{v \mid c_{\sigma'} \rightsquigarrow v\}$ follows from the definition of $\rightsquigarrow$. Let $\sigma \leq_\rho \sigma'$. Since $c_\sigma \in \mathsf{GCInst}^\bullet$, we get $\sigma \in \mathsf{GType}$ and by Lemma 13 also $\mathsf{types}^\bullet(\sigma) = \mathsf{types}^\bullet(\rho(\sigma')) = \{\rho(v) \mid c_{\sigma'} \rightsquigarrow v\} = \{u \mid c_{\rho(\sigma')} \rightsquigarrow_G u\} = V$.
2. Let $(u' \equiv t') \in D$ such that $u \leq_\rho u'$. Since D is well-formed and therefore $\mathsf{TV}(u') \supseteq \mathsf{TV}(t')$, we obtain $\rho(t') \in \mathsf{GTerm}$. Again, The equality $\mathsf{types}^\bullet(t') \cup \mathsf{consts}^\bullet(t') = \{v \mid t' \rightsquigarrow v\}$ follows from the definition of $\rightsquigarrow$. By Lemma 13 and similar reasoning as in 1., we finally derive $\mathsf{types}^\bullet(t) \cup \mathsf{consts}^\bullet(t) = V$. □

We present a more detailed proof of Lemma 4.5. *

Proof. 1. By Lemmas 12.1,11 and 14.1, we get $\sigma \in \mathsf{Cl}(\mathsf{types}^\bullet(c_\sigma)) \subseteq \mathsf{Cl}(V) \subseteq \mathsf{Type}^F$.
2. Let $F = (T, C)$. From definition of Term^F, we have to show that $\mathsf{types}^\bullet(t) \subseteq T$ and $\mathsf{consts}^\bullet(t) \subseteq C$: Let $V_T = V \restriction_{\mathsf{GType}}$ and $V_C = V \restriction_{\mathsf{GCInst}}$. Since $v \in F^v$, we obtain $V_T \subseteq T$ and $V_C \subseteq C$. Finally by Lemma 14.2, we get $\mathsf{types}^\bullet(t) \subseteq V_T$ and $\mathsf{consts}^\bullet(t) \subseteq V_C$. □

We prove Lemma 4.5. *

Proof. From the definition of satisfaction $\mathcal{I} \models D$ iff for all $\theta \in \mathsf{GTSubst}$ and all $\psi \in D$ it holds $[\theta(\psi)]^{\mathcal{I}} = 1$. We omitted $\xi \in \mathsf{Comp}^{\mathcal{I}}$ since all $\psi \in D$ are closed formulas and therefore the evaluation ξ is irrelevant by Lemma 5. But the family $(\theta(\psi))_{\theta \in \mathsf{GTSubst}, \psi \in D}$ equals to $\{\mathsf{def}(u) \mid u \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet\}$.

The following facts are used in the proof of Lemma8.

Lemma 15. Let $\rightsquigarrow_G$ be terminating.

1. There exists at most one strict F^u-interpretation $\mathcal{I}$.
2. Let $u_1, u_2 \in \mathsf{GType}^\bullet \cup \mathsf{GCInst}^\bullet$, $\mathcal{I}_1$ a strict F^{u_1}-interpretation and $\mathcal{I}_2$ a strict F^{u_2}-interpretation. Then $\mathcal{I}_1$ and $\mathcal{I}_2$ are compatible.

Proof. 1. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two strict F^u-interpretations. It follows by well-founded induction on $\rightsquigarrow_G$ that $[v]^{\mathcal{I}_1} = [v]^{\mathcal{I}_2}$ for all $v \in F^u$.
2. If they are not compatible, there exists $v \in F^{u_1} \cap F^{u_2}$ such that $[v]^{\mathcal{I}_1} \neq [v]^{\mathcal{I}_2}$. But this is a contradiction with 1. □