Type Classes and Overloading in Higher-Order Logic

Markus Wenzel*

Technische Universität München
Institut für Informatik, Arcisstraße 21, 80290 München, Germany
http://www4.informatik.tu-muenchen.de/~wenzelm/
wenzelm@informatik.tu-muenchen.de

Abstract Type classes and overloading are shown to be independent concepts that can both be added to simple higher-order logics in the tradition of Church and Gordon, without demanding more logical expressiveness. In particular, model-theoretic issues are not affected. Our meta-logical results may serve as a foundation of systems like Isabelle/Pure that offer the user Haskell-style order-sorted polymorphism as an extended syntactic feature. The latter can be used to describe simple abstract theories with a single carrier type and a fixed signature of operations.

1 Introduction

Higher-order logic (HOL) dates back to Church's 1940 formulation of the "simple theory of types" [2], originally intended as foundation of mathematics.

Gordon later extended the system by an object-level first-order language of types (by including type variables and type constructors), and — most importantly — definitional mechanisms that guarantee safe theory extensions. Various implementations of theorem provers based on Gordon's HOL [4] proved to be very successful for many applications in computer science and mathematics.

Paulson's generic theorem proving environment Isabelle is based on an (intuitionistic) version of HOL since Isabelle-89 [11]. In Isabelle-91, a Haskell-like type system with ordered type classes has been added [9], though without investigating logical foundation issues very much.

Somewhat later, a conceptual bug concerning the handling of empty classes was discovered that actually made Isabelle's meta-logic implementation inconsistent. Embarrassing slips of this kind illustrate why mechanized proof assistants should be based on well-understood logical frameworks only, lest the "formal" proofs conducted by users inherit any uncertainty.

The present paper aims to close this foundational gap of Isabelle. Our main contributions are:

* Research supported by DFG SPP "Deduktion".

– An interpretation of type classes in higher-order logic.
– A definitional mechanism for axiomatic type classes.
– A generalization of constant definitions admitting overloading and recursion over types.

In particular, we will see that type classes have already been implicitly present in Gordon-like HOL systems all the time. So the seeming extensions of Isabelle/Pure over basic HOL can be explained just as additional syntactic features offered for the user's convenience. What really goes beyond Gordon's HOL (extra-logically, though) are overloaded constant definitions.

While the concepts of type classes and overloading can be explained independently in HOL, they are closely related in practice: Without type classes as a syntactic device, overloading tends to become undisciplined. Without overloaded definitions, type classes could be defined but not instantiated in useful manners.

Although the initial motivation arose in the Isabelle setting, the subsequent presentation is more general. Our results can be easily applied to similar HOL systems.

A note on terminology: HOL shall refer to the abstract logical system used to explain the concepts in this paper. The concrete incarnations are Isabelle/Pure (Isabelle's meta-logic), Isabelle/HOL (an object-logic within Isabelle/Pure), and Gordon/HOL. As a quite harmless simplification, HOL can also be identified directly with Isabelle/Pure.

The paper is structured as follows: Section 2 starts with some examples of using type classes, without giving any formal background. Section 3 sketches the syntax and deductive system of the HOL logic. Section 4 discusses the issue of safe theory extension in general and concludes with generalized constant definitions including overloading and recursion over types. Section 5 introduces type classes and their interpretation in HOL. Section 6 concludes with safe mechanisms for definition and instantiation of axiomatic type classes.

2 Examples of Using Type Classes

2.1 Type Classes in Programming Languages

We quickly review some aspects of type classes in languages like Haskell [6].

Within a setting of this kind, classes are supposed to describe collections of types that provide (or *implement*) operations of certain names and types. For example, consider the following **class** definition (modulo concrete Haskell-syntax):

class *ord*
  $\leq :: \alpha_{ord} \to \alpha_{ord} \to bool$

Class *ord* requires of its instances τ to provide some relation $\leq :: \tau \to \tau \to \mathit{bool}$. This is witnessed in the **instance** construct by a suitable definition. For example:

instance *nat* :: *ord*
  $x_{nat} \leq y_{nat} \equiv \mathit{nat\_le}$

Nothing more specific is required of $\leq :: \mathit{nat} \to \mathit{nat} \to \mathit{bool}$ than its type. From the chosen name $\leq$, everyone will think of it as some order, of course. This shall be implemented appropriately by above program text *nat_le*.

So we observe that Haskell type classes can be viewed as the *signature* part of simple algebraic *structures* consisting of one carrier type and associated operations (or *member functions*). Additional semantic properties (or *class axioms*) may come in as mere convention.

Speaking in terms of the example above, the concrete instance can be understood as a poset structure $(\mathit{nat}, \leq^{\mathit{nat}})$.

2.2 Type Classes in HOL

The Haskell notion that instances of type classes provide operations of certain names and types is *not* amenable to logical systems like HOL. One just cannot express within the logic if objects are *declared* or *meaningful*.

Even from an extra-logical point of view, such notions are not very appropriate. The HOL world is total in the sense that everything of any type is always meaningful. Even constants of arbitrary type can be safely declared at any time, without changing very much. In the worst case it may happen that no useful theorems can be derived about some objects. Consider the latter just as a boundary case of loose specification.

We argue that a straightforward interpretation of classes should be simply as set-theoretic predicates: type classes denote classes of types. A view of classes as abstract algebras can be still recovered from this frugal interpretation. As an example consider the following class of orders in HOL:

consts $\leq :: \alpha \to \alpha \to \mathit{prop}$
class *ord*
  reflexive $x_\alpha \leq x_\alpha$
  transitive $x_\alpha \leq y_\alpha \ \wedge \ y_\alpha \leq z_\alpha \Rightarrow x_\alpha \leq z_\alpha$
  antisymmetric $x_\alpha \leq y_\alpha \ \wedge \ y_\alpha \leq x_\alpha \Rightarrow x_\alpha = y_\alpha$

Note that **consts** above is not actually part of the class definition. The declaration of $\leq$ just ensures that the class axioms are syntactically well-formed.

The meaning of *ord* is a type predicate stating that $\leq :: \alpha \to \alpha \to \mathit{prop}$ is an order relation. It does not express anything like "$\leq$ is available on a type" — this would be trivially true in HOL anyway.

Concrete instances $\tau :: \mathit{ord}$ are required to have the corresponding $\leq :: \tau \to \tau \to \mathit{prop}$ specified in such a way, that the order properties are derivable.

This is typically achieved by means of a constant definition[1] prior to the actual instantiation. As an example consider:

defs $x_{nat} \leq y_{nat} \equiv nat\_le$
instance $nat :: ord$

Again observe that **defs** is not part of our **instance** construct, just as **consts** had been independent of **class**. Assuming that the term *nat_le* expresses a suitable relation, we are able to derive *reflexive*, *transitive*, *antisymmetric* for type *nat* in the theory. Thus the instantiation $nat :: ord$ is justified within the logic.

Note that in concrete system implementations the user will have to provide the witness theorems for **instance** explicitly.

Our version of HOL definitions not only admit overloading, but also primitive recursion over types. The latter can be used to mimic lifting of polymorphic operations. For example, consider the following definition:

defs $x_{\alpha\times\beta} \leq y_{\alpha\times\beta} \equiv fst\ x_{\alpha\times\beta} \leq fst\ y_{\alpha\times\beta} \wedge snd\ x_{\alpha\times\beta} \leq snd\ y_{\alpha\times\beta}$

enabling us to derive the order properties of $\leq$ on $\alpha \times \beta$, under the assumption that these already hold on α and β. This justifies an instantiation of the form:

instance $\times :: (ord, ord)\ ord$

Thus the type operator $\times$ can be understood as a functor for direct binary products of order structures.

Note that overloaded definitions must not overlap. In particular, there may be at most one equation for the same type scheme. For example, having already defined $\leq$ on $\alpha \times \beta$ component-wise rules out to redefine it later as lexicographic order.

Thus the signature part of the abstract theories that can be described is *fixed*. Type classes only have the carrier type as a parameter, but not the operations.

This drawback is not specific to HOL, though. Type classes may be only instantiated once in current Haskell-like languages, too.

More examples and applications of type classes as a light-weight mechanism of simple abstract theories can be found in the Isabelle library [7], especially in the `HOL` and `HOL/AxClasses` directories. There is also a tutorial on axiomatic type classes available as part of the Isabelle documentation [15].

Above examples should have illustrated to some extend how the two concepts of overloaded definitions and type classes can be joined into a practically useful mechanism. Both can be understood independently in HOL, though. The logical foundations of **defs** will be explained in §4, especially §4.5. The exact meaning of **class** and **instance** will be given in §6.

[1] Which is overloaded in general, because there may be many different instantiations.

3 The HOL Logic

We briefly sketch the syntax and deductive system of our version of HOL. The presentation is somewhat reminiscent of [13], but differs in many details.

3.1 HOL Syntax

Types and Terms The syntax of HOL is just that of simply-typed λ-calculus with a first order language of types.

Types are either variables α, or applications $(\tau_1, \ldots, \tau_n)\, t$ of an n-place constructor t applied to types τ_i. We drop the parentheses for $n \in \{0, 1\}$. Binary constructors are often written infix, e. g. function types as $\tau_1 \to \tau_2$ (associate to the right).

Terms are built up from explicitly typed atomic terms (constants c_τ or variables x_τ) through application $t\, u$ (of type τ_2, provided that $t : \tau_1 \to \tau_2$ and $u : \tau_1$) and abstraction $\lambda\, x_{\tau_1}.t$ (of type $\tau_1 \to \tau_2$, provided that $t : \tau_2$). As usual, application associates to the left and binds most tightly. An abstraction body ranges from the dot as far to the right as possible. Nested abstractions like $\lambda\, x.\, \lambda\, y.t$ are abbreviated to $\lambda\, x\ y.t$.

Note that atomic terms a_τ actually consist of two components: name a and type τ. In particular, variables x_{τ_1} and x_{τ_2} with the same name but different types are treated as different.

Furthermore we assume suitable functions TV (on types or terms) and FV (on terms), yielding the type variables and free term variables of their respective arguments.

Type Substitutions and Instances Type substitutions $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ shall be defined as usual. Their application (to types or terms) is written postfix.

For types or terms T, U, we call T a *type instance* of U (written $T \leq U$) iff there is some substitution δ such that $T = U\delta$. Given any set A of types or terms, let $A{\downarrow}$ denote the downwardly closed set of all of its type instances.

Theories consist of a *signature* part (constants and types) together with *axioms*. We use a notation like:

$$\Theta_2 = \Theta_1 \ \cup\ (\alpha_1, \ldots, \alpha_n)\, t \ \cup\ c :: \sigma \ \cup\ \vdash \Phi$$

meaning that theory Θ_2 is the extension of Θ_1 by declaring type constructor t of arity n, constants $c :: \sigma$ (representing the set $c_\sigma{\downarrow}$), and asserting all axioms of the set Φ.

We always assume that theories contain at least the following signature:

$prop$	propositions
$\alpha \to \beta$	functions
$\Rightarrow :: prop \to prop \to prop$	implication
$\equiv :: \alpha \to \alpha \to prop$	equality
$\forall :: (\alpha \to prop) \to prop$	universal quantification

As usual, $\forall(\lambda\, x_\tau.t)$ is written as $\forall x_\tau.\ t$. Nested $\forall$'s are abbreviated like nested λ's. Other logical operators ($True$, $False$, $\wedge$, etc.) could be introduced in an obvious way. They sometimes simplify our presentation, without being really necessary.

Any term of type $prop$ is a *formula*. Our type of propositions is sometimes called o in the literature, in Gordon/HOL the analogous type is $bool$.

3.2 The HOL Deductive System

Due to space limitations, we do not give a full calculus for HOL here. It suffices to say that given some theory Θ, we have some inductively defined relation $\Gamma \vdash_\Theta \varphi$ of *derivable sequents* (where antecedents Γ are *finite* sets of formulas).

We use the usual abbreviations: $\Gamma \vdash \varphi$ for $\Gamma \vdash_\Theta \varphi$ if Θ is clear from context, $\vdash \varphi$ for $\{\} \vdash \varphi$, and $\Gamma_1, \Gamma_2 \vdash \varphi$ for $\Gamma_1 \cup \Gamma_2 \vdash \varphi$, and so on. The full set of inference rules for $\vdash_\Theta$ consists of about 15 schemas. As an example we present only two:

$$\frac{\Gamma \vdash \psi}{\Gamma \setminus \{\varphi\} \vdash \varphi \Rightarrow \psi}\ (\Rightarrow\text{I}) \qquad \frac{\Gamma_1 \vdash \varphi \Rightarrow \psi \quad \Gamma_2 \vdash \varphi}{\Gamma_1, \Gamma_2 \vdash \psi}\ (\text{MP})$$

Thus we get a single-conclusion sequent calculus, similar to the one presented in [13] for Gordon/HOL. If the rules are chosen suitably, the system may also be read as natural deduction (which is preferred in the Isabelle literature [12]). This and other details (e. g. classical vs. intuitionistic HOL) do not matter here. Subsequently, some general idea of what theorems are derivable in higher-order logic will be sufficient for the level of abstraction of this paper.

4 Meta-level Definitions

The most important contribution of Gordon/HOL [4] over the original formulation of Church [2] are disciplined mechanisms of theory extension. Using only these instead of unrestricted axiomatizations guarantees that certain nice properties of theories are preserved.

Such extensions are usually called *conservative*, *definitional*, *sound* etc., often with some confusion about the exact meaning of these phrases. So before introducing our generalized constant definitions (cf. §4.5), we set out to discuss what qualifies extension mechanisms as safe in our HOL setting.

4.1 Consistency Preservation

A theory is called *(syntactically) consistent* iff not all formulas are derivable. An inconsistent theory certainly does not have any models, since every formula is a theorem (including $\vdash \mathit{False}$). A theory extension mechanism is called *consistency preserving* iff any extension of consistent theories is also consistent.

Although being a nice concept, consistency preservation is certainly not the key property that qualifies theory extensions as safe in the HOL setting:

Syntactic consistency of theories is not a very strong property. In particular it does not necessarily imply existence of suitable models. This would require *completeness* of the deductive system wrt. the underlying model theory, which does not generally hold in higher-order logic.

More surprisingly, some kinds of safe extensions do not necessarily preserve consistency in general — notably Gordon/HOL type definitions, see below.

4.2 Syntactic conservativity

Definition 1. *An extension Θ_2 of some theory Θ_1 is called* (syntactically) conservative *iff for any formula φ of signature Θ_1 it holds that $\vdash_{\Theta_2} \varphi \Rightarrow \vdash_{\Theta_1} \varphi$.*

Syntactic conservativity is traditional [1]. It ensures that extensions do not change derivability of formulas that do not contain any of the newly introduced syntactic objects (constants and types). It is also very easy to see that syntactic conservativity implies consistency preservation.

We consider syntactic conservativity as a minimum requirement for well-behaved extension mechanisms within purely deductive logical frameworks.

4.3 Model Preservation

We briefly review Gordon/HOL's extension mechanisms and the way they are justified as *conservative* [13]. Basically, the system features two kinds of theory extensions:[2]

Constant definition $\Theta_2 = \Theta_1 \cup c :: \sigma \cup \vdash c_\sigma \equiv t$ provided that c is new and does not occur in t, also $\mathrm{FV}(t) = \{\}$ and $\mathrm{TV}(t) = \mathrm{TV}(\sigma)$.

Type definition $\Theta_2 = \Theta_1 \cup (\alpha_1, \ldots, \alpha_n)\, t \cup \vdash (\alpha_1, \ldots, \alpha_n)\, t \approx A$, where t is an n-ary type constructor and A is a term representing some set, and the notation $\tau \approx A$ shall abbreviate some suitable formula stating that τ is isomorphic to A. The definition shall be well-formed, provided t is new and does not occur in A, also $\mathrm{FV}(A) = \{\}$, $\mathrm{TV}(A) \subseteq \{\alpha_1, \ldots, \alpha_n\}$, and non-emptiness of A is derivable in Θ_1.

[2] Actually, Gordon/HOL admits more general forms of (loose) specifications than presented here. We can ignore this without loss of generality, at the level of abstraction of this paper.

Both these mechanisms are justified as being safe extensions because they preserve Gordon/HOL standard models: If Θ_1 has such a model, so does Θ_2. These models are specifically tailored for the Gordon/HOL logic [13] and are quite special in at least the following ways:

- They are classical.
- They are standard[3].
- Types are interpreted from subset-closed universes.

In particular, the last property is the crucial one for type definitions being safe. We quickly sketch a counterexample where Gordon/HOL type definitions do not preserve syntactic consistency.

Consider some base theory Θ_0 and some formula $three^\alpha$ expressing that α has cardinality 3. Now let $\Theta_1 = \Theta_0 \cup \vdash(three^\alpha \Rightarrow False)$. It is easy to see that Θ_1 is consistent: For example one can give a very simple model (non-standard in the Gordon/HOL-sense) where all interpretations of types are sets of some cardinality 2^k, for $k \in \mathbb{N}_0$. Finally let $\Theta_2 = \Theta_1 \cup thr \cup \vdash thr \approx \{0, 1, 2\}$, and observe that $\vdash_{\Theta_2} three^{thr}$ is derivable, and thus $\vdash False$. So this theory is inconsistent!

Furthermore, type definitions are not necessarily syntactically conservative, even if the theories involved have a Gordon/HOL standard model.

The counterexample is a simple modification of the previous one: Basically, just substitute some proper constant definition $c \equiv t$ for $False$. Then it is relatively easy to see that $\Theta_0, \Theta_1, \Theta_2$ all have standard models. An argument similar to the one above shows that $\vdash c \equiv t$ is not derivable in Θ_1, but is so in Θ_2. That is, the definition of type thr changed derivability on existing formulas — it is not syntactically conservative!

Of course nothing is wrong with Gordon/HOL type definitions, as long as one does not leave the dedicated model theory. The above examples should illustrate, though, why we cannot justify our extensions in this setting:

Our HOL should serve as a *meta-logical framework* for expressing many different kinds of deductive systems (or *object-logics* in Isabelle parlance). In other words, results about safeness of extensions should be applicable to Isabelle/Pure, not just to particular object-logics like Isabelle/HOL.

Focusing solely on the Gordon/HOL standard model theory here would basically restrict object-logics to what is known as *shallow embeddings* in the HOL community. Then justifying for example full Zermelo-Fränkel set theory in this framework [3] would be much more difficult than if encoded as a purely deductive system the Isabelle way.

[3] *Standard* in the sense of [1] which also treats a certain kind of *non-standard* models. The latter may interpret $\tau_1 \to \tau_2$ as proper subsets of the full function space $[\![\tau_1]\!] \to [\![\tau_2]\!]$. Interestingly, the deductive system of classical HOL is *complete* wrt. this class of *general* models [5].

4.4 Meta-safe Extensions

We now introduce a notion of safe theory extensions that is more appropriate for our meta-logical HOL setting:

Definition 2. *Any extension Θ_2 of some theory Θ_1 is called* meta-safe *iff:*

- *It is syntactically conservative.*
- *Introduced objects are* (syntactically) realizable*:*
  For all new c there is some function r from types to terms, such that $r(\tau) : \tau$ and $\vdash_{\Theta_2} \varphi \Rightarrow \vdash_{\Theta_1} \varphi[r/c]$, for all φ. Here the notation $[r/c]$ shall denote replacement of any c_τ by $r(\tau)$.

First observe that because of syntactic conservativity, object-logics are free to *ignore* meta-safe extensions, by just not referring to them syntactically.

Syntactic realizability can be seen as a generalized counterpart of model preservation, always staying within the deductive system of HOL, though. Newly introduced names can be seen as just an abbreviation for pre-existent syntactic objects that have the same properties (because the same theorems are derivable).

So there are two ways for object-logics to *cooperate* with meta-safe extensions[4]: Either just consider all object-level formulations modulo expansion of all meta definitions, without changing the semantics, or adjust your model theory to interpret defined objects according to $[r/c]$, utilizing the realization function.

4.5 Overloaded Constant Definitions

Having provided enough preliminaries, we can now present our generalization of constant definitions:

Overloaded constant definition $\Theta_2 = \Theta_1 \cup c :: \sigma \cup \vdash \Delta_c$, where Δ_c is some set of equations $c_\tau \equiv t$. The definition shall be well-formed, provided that c is new, all $\mathrm{FV}(t) = \{\}$ and $\mathrm{TV}(\tau) = \mathrm{TV}(t)$; furthermore all c_τ have to be instances of c_σ, no two different c_{τ_1}, c_{τ_2} may have common instances, and recursive occurrences of any $c_{\tau'}$ in some t may be only at such types τ' that are strictly simpler than τ in a well-founded sense.

In practice, the strictly simpler notion above will be just structural containment. Thus we get constant definitions with general *primitive recursion over types*. As an example, consider:

$$0 :: \alpha$$
$$\vdash 0_{nat} \equiv \mathit{zero}$$
$$\vdash 0_{\alpha\times\beta} \equiv (0_\alpha, 0_\beta)$$
$$\vdash 0_{\alpha\to\beta} \equiv \lambda\, x_\alpha.0_\beta$$

[4] Think of Gordon/HOL type definitions, where the representing sets may contain meta-safely introduced constants, for example.

which defines 0 on *nat*, also lifting it to binary products and function spaces. 0_τ is still unspecified on types τ that are not instances of *nat* or $\alpha \times \beta$ or $\alpha \to \beta$.

Note that this extension mechanism requires that all defining equations are given at the same time (right after the constant declaration). One might want to relax this in concrete system implementations, allowing the user to augment theories by additional equations for constants on new type instances in an incremental way. The system will then have to keep track of all the partial definitions, ensuring that the resulting jumble of equations can be sorted out into proper overloaded constant definitions at any time.

We now briefly sketch why overloaded constant definitions are indeed meta-safe. Due to space limitations, we have to gloss over various technical lemmas about HOL deductions (most importantly freeness of unspecified constants and the deduction theorem).

The key property of our generalized constant definitions is:

Lemma 1. *Given any overloaded constant definition* $\Theta_2 = \Theta_1 \cup c :: \sigma \cup \vdash \Delta_c$. *Then there is some partial function* f *from types to terms of* $\Theta_1 \cup c :: \sigma$, *such that* $f(\tau) : \tau$, *and* f *establishes all type instances of* Δ_c*:*

$$\vdash_{(\Theta_1 \cup c::\sigma)} \Delta_c{\downarrow}[f/c]$$

The proof exploits the well-formedness restrictions on the set of equations Δ_c in straightforward ways: Some canonical f^{Δ_c} is constructed by well-founded recursion over types such that the given equations hold. Mainly this works, because no two different c_{τ_1}, c_{τ_2} on the l.h.s. have common instances, and recursive occurrences on the r.h.s. are well-foundedly simpler. Also $\mathrm{TV}(\tau) = \mathrm{TV}(t)$ plays an important rôle.

Note that f^{Δ_c} is really partial in general, i.e. $[f^{\Delta_c}/c]$ does not necessarily eliminate all type instances of $c :: \sigma$. If one views Δ_c as a convergent term rewriting system, it leaves exactly those c_τ unchanged that are normal wrt. Δ_c.

One can easily extend f^{Δ_c} to some total F^{Δ_c} that also eliminates leftover c_τ (replacing them by any term of type τ), such that $\vdash \Delta_c{\downarrow}[F^{\Delta_c}/c]$ in Θ_1.

Our main result on the issue of meta-level definitions is:

Theorem 1. *Overloaded constant definitions are meta-safe.*

The proof exploits f^{Δ_c} and F^{Δ_c} as constructed above. Then both syntactic conservativity and realizability are relatively simple consequences of lemma 1. Unfortunately, we cannot give more details (which are rather technical) at the level of abstraction of this paper.

5 Type Classes

5.1 An Order-sorted Type System

The HOL language as presented in §3 provides two syntactic layers: higher-order *terms* that are annotated by first-order *types*. We now conceptually add a third

level of ordered *type classes* (or *sorts*) that qualify types. Thus the algebra of types becomes an order-sorted structure that is amenable to well-known techniques like order-sorted unification [14]. In particular, ML-style type inference can be easily generalized to the order-sorted system [9,10].

Order-sorted Type Signatures consist of three basic components: a finite set C of *type classes*, a *class inclusion* relation $\preceq$, and a set of *type arities*.

The initial class structure $(C, \preceq)$ is canonically extended to a quasi-ordered sort structure $(S, \sqsubseteq)$ such that sorts are finite sets of classes: Any sort $s = \{c_1, \ldots, c_n\}$ is supposed to represent the intersection $c_1 \sqcap \ldots \sqcap c_n$. Inclusion is extended from classes to sorts accordingly:

$$s_1 \sqsubseteq s_2 \;:\Longleftrightarrow\; \forall c_2 \in s_2.\, \exists c_1 \in s_1.\, c_1 \preceq c_2$$

Note that there is always a greatest sort, namely the empty intersection $\{\}$, which shall be subsequently written as $\top$.

Type arities are declarations of the form $t :: (s_1, \ldots, s_n)\, s$, where t is an n-place type constructor, and $s_1, \ldots, s_n, s$ are sorts. This is supposed to be a partial specification of how t acts on certain subsets of the universe of types.

Sort Assignment We assume that type variables α_s carry globally fixed sorting information. One can think of variables as actually consisting of two components: base name α and sort s.

Now given some order-sorted type signature, sorts are assigned to types via the following set of rules:

$$\frac{}{\alpha_s : s} \qquad \frac{\tau_1 : s_1 \quad \cdots \quad \tau_n : s_n \quad t :: (s_1, \ldots, s_n)\, s}{(\tau_1, \ldots, \tau_n)\, t : s} \qquad \frac{\tau : s_1 \quad s_1 \sqsubseteq s_2}{\tau : s_2}$$

While there may be many type arities for the same constructor, this introduces neither overloading nor partiality to the level of types. In fact, type arity declarations do not change the well-formedness of types (as defined in §3) at all. They only influence sort assignment — via the second rule above. Even having no arities for some constructor is no problem, then one just cannot derive interesting sort assignments.

In general, there may be many sorts assigned to any given type. The literature [14] calls a type signature *regular* iff for all types, the set of assigned sorts has some least element (modulo sort equivalence). This always holds in our setting, because sort structures are closed wrt. intersection. Another nice property is *co-regularity* which guarantees unitary order-sorted unification of types [14] and principal type schemes for arbitrary terms [9].

Such technical issues do not matter here. We will be more interested in the logical content of order-sorted type signatures (see §5.3).

5.2 Representing Type Classes in HOL

Expressing type predicates in HOL might seem difficult at first sight: We cannot have objects c: "*type*" $\to$ *prop*, as for good reasons there is no type of all types.

Type predicates are not needed as first class objects, though. A kind of *propositional language of types* that is capable to express class membership "$\tau \in c$" will be sufficient. Now HOL obviously provides this sort of thing: Any formula $\varphi[\alpha]$ that (potentially) contains some type variable α may be viewed as a proposition about types. As an example consider $\forall x_\alpha\ y_\alpha.\ x_\alpha \equiv y_\alpha$ that describes the class of all singleton types.

Remains the problem to encode *class constants* (in a way that admits some meta-safe mechanism for *class definitions*). There are probably many ways to accomplish this, the one presented now seems to be particularly easy to motivate.

The Encoding First, we augment our basis theory by simply adding unspecified types $\alpha\ itself$ and constants $TYPE :: \alpha\ itself$.

Now for any type class c introduced by the user, we declare a polymorphic constant $c :: \alpha\ itself \to prop$. Applications of the form $(c_{\tau\ itself \to prop}\ TYPE_{\tau\ itself})$, which are of type *prop*, shall be considered to represent the proposition "τ is member of c". Subsequently, the telling notation $\langle\!|\tau : c|\!\rangle$ will be used to abbreviate these terms.

This encoding seems to be an elaboration of a folklore technique from the LCF community, used to express flatness of domains.

A Motivation So far, we have just introduced abbreviations $\langle\!|\tau : c|\!\rangle$ for some terms $(c_{\tau\ itself \to prop}\ TYPE_{\tau\ itself})$. How can we understand this as a representation of "τ is a member of c"?

The following motivation is based on a simple set-theoretic semantics of HOL, where types denote sets and type constructors functions that operate on sets.

We choose to interpret $[\![itself]\!]$ as the function $A \mapsto \{A\}$, then $[\![\tau\ itself]\!] = \{[\![\tau]\!]\}$ for all types τ. In other words, type constructor *itself* builds singleton sets containing the argument itself only. The sole element of any $[\![\tau\ itself]\!]$ will be $[\![TYPE_{\tau\ itself}]\!]$, so we see also that $TYPE_{\tau\ itself}$ has to represent type[5] τ.

Next consider $[\![\tau\ itself \to prop]\!]$. This is interpreted as $\{[\![\tau]\!]\} \to \{0, 1\}$, assuming that $[\![\to]\!]$ is set-theoretic function space, and $[\![prop]\!]$ just the boolean values. Observe that in general, function spaces $\{a\} \to B$ with singleton domain set $\{a\}$ may be viewed as just an isomorphic copy of B marked (or parameterized) by a. So $[\![\tau\ itself \to prop]\!]$ are propositions parameterized by types and objects $c_{\tau\ itself \to prop}$ can already be understood as expressing type membership. Their formal application to the canonical elements $TYPE_{\tau\ itself}$ is strictly speaking redundant, but then $\langle\!|\tau : c|\!\rangle$ also has type *prop* syntactically.

[5] There is nothing wrong with some terms ("objects") representing types ("collections of objects") in higher-order logic.

Note that above interpretation of $\alpha\,itself$ and $TYPE :: \alpha\,itself$ could have been *enforced* by means of Gordon/HOL-style definitions:

$$\vdash \alpha\,itself \approx \{\lambda x_\alpha.\, True\} \qquad \text{and} \qquad \vdash TYPE_{\alpha\,itself} \equiv \lambda x_\alpha.\, True$$

Unfortunately, the first one is an object-level type definition, which is unavailable at our more abstract meta-level of HOL. This is why we prefer to leave $\alpha\,itself$ and $TYPE_{\alpha\,itself}$ unspecified in the first place.

It is important to note that there are no HOL terms representing type classes *per se*. For that we would have to leave HOL and conceptually abstract over the first position of $\langle\!|\tau : c|\!\rangle$. Thus contexts of the form $\langle\!|\_ : c|\!\rangle$ could be viewed as extra-logical representations of actual type predicates.

5.3 Interpreting the Order-sorted Type System

We are now ready to explain the order-sorted concepts of §5.1 in terms of HOL: The meaning of primitive type signature components will be defined in a quite obvious manner. Derived notions that depend on these (e. g. sort assignment) are shown to be consistent with appropriate logical counterparts.

Order-sorted Type Signatures have the following logical content:

Classes c appear as polymorphic constant declarations $c :: \alpha\,itself \to prop$ in the theory's signature (cf. §5.2). Recall that class membership is encoded via some terms written $\langle\!|\tau : c|\!\rangle$.

Class inclusion $c_1 \preceq c_2$ is simply expressed point-wise using logical implication as formula $\langle\!|\alpha : c_1|\!\rangle \Rightarrow \langle\!|\alpha : c_2|\!\rangle$.

Sorts $s = \{c_1, \ldots, c_n\}$ are supposed to represent intersections of finitely many classes. Thus sort membership $\tau : \{c_1, \ldots, c_n\}$ can be expressed using conjunction as $\langle\!|\tau : c_1|\!\rangle \wedge \cdots \wedge \langle\!|\tau : c_n|\!\rangle$. The latter term shall be abbreviated as $\langle\!|\tau : \{c_1, \ldots, c_n\}|\!\rangle$. Note that this interpretation is well-defined, independently of order (or repetition) of $c_1, \ldots, c_n$.

Sort inclusion $s_1 \sqsubseteq s_2$ has been defined in terms of class inclusion in §5.1. To show that this is compatible with $\langle\!|\alpha : s_1|\!\rangle \Rightarrow \langle\!|\alpha : s_2|\!\rangle$ in the logic one has to demonstrate that this formula can be derived in HOL under the assumption of the class inclusions taken from the corresponding relation $\preceq$ of the type signature. The proof of this fact just relies on some basic deductive properties of $\wedge$.

Type arities $t :: (s_1, \ldots, s_n)\, s$ are simple schematic statements about the image of type constructors. We express this point-wise as follows:

$$\langle\!|\alpha_1 : s_1|\!\rangle \wedge \cdots \wedge \langle\!|\alpha_n : s_n|\!\rangle \Rightarrow \langle\!|(\alpha_1, \ldots, \alpha_n)\, t : s|\!\rangle$$

So in ordinary mathematical notation, arity declarations would be something like $f(A_1, \ldots, A_n) \subseteq A$ and not $f\colon A_1 \times \cdots \times A_n \to A$.

Sort Contexts Sorted type variables α_s are supposed to express some implicit restriction to types of certain sorts. Thus formulas $\varphi[\alpha_{s_1}, \beta_{s_2}, \ldots]$ have to be interpreted actually under additional assumptions $\langle\!|\alpha_{s_1} : s_1|\!\rangle, \langle\!|\beta_{s_2} : s_2|\!\rangle, \ldots$.

In general, given any term or type T, let $\mathcal{C}(T)$ denote its set of implicit sort constraints, which shall be also called *sort context* of T.

Sort Assignment $\tau : s$ has been defined §5.1 relatively to a given type signature via a certain set of inference rules. We show compatibility with a corresponding logical notion: If $\tau : s$ holds syntactically, then $\mathcal{C}(\tau) \vdash \langle\!|\tau : s|\!\rangle$ is derivable in HOL (having the implicit sort constraints appear as explicit assumptions).

In order to prove this, simply mimic the syntactic sort assignment rules of §5.1 by suitable logical counterparts. For example, the last rule would become:

$$\frac{\mathcal{C}(\tau) \vdash \langle\!|\tau : s_1|\!\rangle \quad s_1 \sqsubseteq s_1}{\mathcal{C}(\tau) \vdash \langle\!|\tau : s_2|\!\rangle}$$

These rules are either logical trivialities or just variants of modus-ponens combined with instantiation, recalling from above the meaning of $s_1 \sqsubseteq s_2$ and $t :: (s_1, \ldots, s_n)\, s$ as certain implications.

Putting all these results together, we see that syntactic operations performed at the type signature level (e. g. during order-sorted unification or type inference) can be understood as a *correct approximation* of logical reasoning.

Seen the other way round, a simple fragment of the propositional logic of types within HOL is reflected at the type signature level, thus automating some portions of logical reasoning behind the scenes, to the user's benefit.

6 Class Definitions and Instantiations

We finally give the logical meanings of **class** and **instance** that have already been sketched in §2.2.

First the basic mechanism that introduces type classes in a disciplined way:

Class definition $\Theta_2 = \Theta_1 \ \cup \ c :: \alpha\, itself \to prop \ \cup \ \vdash \langle\!|\alpha : c|\!\rangle \equiv \varphi$ provided that c is new and does not occur in φ, also $\mathrm{FV}(\varphi) = \{\}$ and $\mathrm{TV}(\varphi) \subseteq \{\alpha\}$.

Theorem 2. *Class definitions are meta-safe.*

The proof is very simple: Class definitions are already *almost* well-formed definitions of constants $c :: \alpha\, itself \to prop$. Just the equation $\vdash \langle\!|\alpha : c|\!\rangle \equiv \varphi$ looks odd at first sight, but is actually equivalent to a proper definition $\vdash c_{\alpha\ itself \to prop} \equiv \lambda\, x_{\alpha\ itself}.\varphi$.

We can now explain the **class** construct, which has the general form:

class $c \preceq c_1, \ldots, c_n$
$\varphi_1 \ \ldots \ \varphi_m$

where $c_1, \ldots, c_n$ are the superclasses of c and $\varphi_1, \ldots, \varphi_m$ the class axioms (with $\mathrm{TV}(\varphi_j) \subseteq \{\alpha\}$ for all $j = 1, \ldots, m$). This shall be just considered concrete user interface syntax for the following proper class definition:

$$c :: \alpha\ \mathit{itself} \to \mathit{prop}$$
$$\vdash \langle\!|\alpha : c|\!\rangle \equiv \langle\!|\alpha : c_1|\!\rangle \wedge \cdots \wedge \langle\!|\alpha : c_n|\!\rangle \wedge \varphi'_1 \wedge \cdots \wedge \varphi'_m$$

where the φ'_i are the $\forall$-closures of φ_i (thus ensuring $\mathrm{FV}(\varphi'_i) = \{\}$).

The following theorems are derivable from this definition (simply by taking the equivalence apart and stripping some $\forall$'s): The class inclusions $c \preceq c_i$ (or $\vdash \langle\!|\alpha : c|\!\rangle \Rightarrow \langle\!|\alpha : c_i|\!\rangle$), the *abstract class axioms* $\varphi_j[\alpha_c]$ (or $\vdash \langle\!|\alpha : c|\!\rangle \Rightarrow \varphi_j$), and the *class instantiation rule* $\vdash (\ldots \Rightarrow \langle\!|\alpha : c|\!\rangle)$.

Next is the **instance** construct which comes in two variants:

instance $c_1 \preceq c_2$	called *abstract instantiation*
instance $t :: (s_1, \ldots, s_n)\, s$	called *concrete instantiation*

provided that the class inclusion, or type arity is derivable in the corresponding theories: $\vdash \langle\!|\alpha : c_1|\!\rangle \Rightarrow \langle\!|\alpha : c_2|\!\rangle$, or $\vdash \langle\!|\alpha_1 : s_1|\!\rangle \wedge \cdots \wedge \langle\!|\alpha_n : s_n|\!\rangle \Rightarrow \langle\!|(\alpha_1, \ldots, \alpha_n)\, t : s|\!\rangle$.

The effect of instantiations is to augment the current order-sorted type signature by the stated inclusion $c_1 \preceq c_2$ or type arity $t :: (s_1, \ldots, s_n)\, s$.

Theorem 3. *Class instantiations are meta-safe.*

For a proof just note that **instance** is logically almost vacuous: The (axiomatic) additions to the type signature have already been derivable beforehand.

7 Conclusion

We have seen that simple traditional HOL systems (providing object-level type variables) implicitly contain some propositional language of types that may serve as an interpretation of type classes, type arities and related notions from Haskell-like type systems. We could even have supported more general *qualified types* [8], notably n-ary type relations, as does the programming language Gofer and recently proposed extensions of Haskell. Thus the whole order-sorted type system turns out to be just an addition to user convenience, without really changing expressiveness of the logic.

We have also introduced three new safe theory extension mechanisms: overloaded constant definitions with possible recursion over types, class definitions and class instantiations. These have been justified at the purely deductive meta-logical level, without referring to model theory.

One of the most surprising results of this work is *simplicity*. We did not have to leave the seemingly old-fashioned HOL in favour of full-blown theories of dependent types. The sort of abstract theories that type classes are capable of can be offered in HOL at no additional cost, apart from implementation efforts.

Acknowledgments I would like to thank Tobias Nipkow for many controversial and encouraging discussions about this topic. Olaf Müller, Wolfgang Naraschewski, David von Oheimb and Oscar Slotosch commented draft versions of the paper.

References

1. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.
2. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, 1940.
3. M. J. C. Gordon. Set theory, higher order logic or both? In *Proc. 9th TPHOLs*, volume 1125 of *Lecture Notes in Computer Science*, pages 191–201. Springer-Verlag, 1996.
4. M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
5. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.
6. P. Hudak, S. L. P. Jones, and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.2). *SIGPLAN Notices*, March, 1992.
7. The Isabelle library. http://www4.informatik.tu-muenchen.de/~nipkow/isabelle/.
8. M. P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, University of Oxford, 1992.
9. T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.
10. T. Nipkow and C. Prehofer. Type checking type classes. In *20th ACM Symp. Principles of Programming Languages*, 1993.
11. L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
12. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
13. A. Pitts. The HOL logic. In Gordon and Melham [4], pages 191–232.
14. M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1989.
15. M. Wenzel. *Using axiomatic type classes in Isabelle — a tutorial*. Available at http://www4.informatik.tu-muenchen.de/~nipkow/isadist/axclass.dvi.gz.