A Mechanized Translation from Higher-Order Logic to Set Theory

Alexander Krauss and Andreas Schropp

Technische Universität München, Institut für Informatik
{krauss,schropp}@in.tum.de

Abstract. In order to make existing formalizations available for set-theoretic developments, we present an automated translation of theories from Isabelle/HOL to Isabelle/ZF. This covers all fundamental primitives, particularly type classes. The translation produces LCF-style theorems that are checked by Isabelle's inference kernel. Type checking is replaced by explicit reasoning about set membership.

1 Introduction

Compared to the type theories underlying most widely-known proof assistants today, set theory has received less attention in the field of interactive theorem proving. This is unfortunate, since set theory is arguably the formalism that comes closest to a "standard foundation of mathematics" and since it provides a rich and well-understood foundation.

This paper describes an automated translation of theories from Isabelle/HOL to Isabelle/ZF (which implements ZFC). We interpret recorded proof terms, and the resulting derivations are again checked by Isabelle's LCF-style inference kernel, which ensures soundness of the approach and implementation.

While the general idea of a mapping to set theory is not new—in fact, the standard semantics of HOL [17] is defined in ZFC—, translating entire theories of realistic proof developments is a highly non-trivial task: In addition to the bare proofs, one must cope with theory extension mechanisms like constant and type definitions. Moreover, Isabelle's type classes and overloading as well as interactions between the object-logic Isabelle/HOL and the logical framework Isabelle/Pure complicate this task. To our knowledge, this is the first complete translation scheme to set theory (it is complete except for a fine point, discussed in §5.2). It is also the first proof-producing one.

1.1 Motivation

The motivation for this work comes from several directions:

Experimenting with theorem proving based on set theory. The simple type theory of HOL is sometimes a limitation that makes certain concepts (e.g., monads, which would require parametrization over type constructors) hard or impossible to formalize. One way of improving this is to move to a stronger type theory, like the calculus of constructions [2] or the recently proposed HOL Omega [8]. An alternative path is to abandon types as an integral part of the logic and to work in a logic that is untyped, but expresses the equivalent of typing judgements explicitly as propositions. A type discipline could then be reintroduced as an extra layer of "soft types" built on top of an untyped LCF kernel. Such a system could make the notion of type checking more open to experimentation, since it is easier to change something that is not part of the foundation.

This idea has been mentioned in the literature several times [5, 9, 20, 6], and, in principle, Isabelle/ZF could be a starting point to explore these possibilities. This work intends to explore how HOL-style reasoning can be turned into set-theoretic reasoning, mechanized in Isabelle/ZF.

Exchange format between proof assistants. Although there already exist translation facilities for proofs between different theorem provers, combining developments from different systems in practice is still an open problem (Gordon calls it a "Grand Challenge" [6]). Set theory has sometimes been mentioned as a candidate for an exchange format between different logics, mainly because the semantics of many logical systems can be defined set-theoretically.

Reviving Isabelle/ZF. Our translation makes the large body of theories developed in Isabelle/HOL available in ZF. We believe that this may facilitate the development of proof tools (e.g., arithmetic decision procedures) in set theory, which typically require a certain amount of established theory.

None of the goals that we take as a long-term motivation can be solved by this work alone. However, we aim to take a small step towards them with the following concrete contributions:

- By carefully revisiting the foundations of the Isabelle/HOL system, we show how all its primitives can be translated to a purely definitional theory in Isabelle/ZF (with global choice; see §3). In particular, type classes and overloading are eliminated.
- We provide a prototype implementation of this mapping that produces machine-checked proofs in Isabelle/ZF.

1.2 Related Work

The standard set-theoretic semantics of HOL is described by Pitts [17]. Gordon [5] experimented with combinations of HOL and ZFC by axiomatizing a type of sets in HOL. He describes a transfer principle between the two worlds which is very similar to our translation of propositions. However, Gordon's translation is not proof-producing, and one must trust the correctness of its implementation. Moreover, the semantics of the axiomatic combination of HOL and ZFC are still slightly unclear.

A number of proof transformation tools have been developed to replay proofs of one theorem prover in another, mostly within the HOL family of provers [10, 14]. Similarly, the AWE extension [1] interprets theories within the Isabelle/HOL system, replacing types, constants, and axioms with concrete models. Our translation is closely related to these tools but slightly more complex, since our target language is untyped and type reasoning has to be made explicit.

2 Formal Preliminaries

Isabelle [15] is a generic theorem prover, which supports reasoning in different object-logics embedded in a logical framework (often referred to as the "meta-logic"). While many Isabelle applications use Isabelle/HOL exclusively, this particular work critically relies on the generic nature of the system.

In this section, we recall the logical foundations of Isabelle, including how the object-logics HOL and ZF are embedded in Isabelle/Pure.

The meta-syntactic abbreviation $\overline{t_m}$ always denotes the sequence $t_1 \dots t_m$. Binding a sequence of variables $(\lambda\, \overline{x_m : \tau_m}.\ t)$ always means iteration of binders $(\overline{\lambda\, x_m : \tau_m.}\ t)$. We omit type and sort annotations when they are clear from the context. Substitution of t_2 for x in t_1 is written as $t_1[x := t_2]$.

2.1 Pure

Isabelle/Pure is a simply-typed intuitionistic higher-order logic featuring just implication, universal quantification, equality and schematic polymorphism with type classes [19, 7]. Unlike many dependently-typed systems, it retains the stratification into sorts, types, terms, and proofs, which we discuss in this order.

Sorts and Types. Syntactically, (type) classes c are formal names, and sorts s are finite symbolic intersections of classes, written as finite sets $\{c_1, \dots, c_n\}$, where the empty intersection is denoted by $\top$. Types are either type constructor applications, or type variables annotated with their sort.

$$\tau \quad ::= \quad \kappa\, \overline{\tau_m} \quad | \quad \alpha^s$$

Special type constructors are *prop* (propositions) and $\Rightarrow$ (function space).

A set of type classes together with an acyclic subclass relation $\prec$ and a set A of arities of the form $\kappa :: (\overline{s_m})c$ is called an order-sorted algebra [18]. It induces the type-in-class relation $\tau : c$ defined by the following rules, where $\tau : \{c_1, \dots, c_n\}$ abbreviates $\tau : c_1, \dots, \tau : c_n$.

$$\frac{\tau : c_1 \qquad c_1 \prec c_2}{\tau : c_2} \qquad \frac{\overline{\tau_m : s_m} \qquad (\kappa :: (\overline{s_m})c) \in A}{\kappa\, \overline{\tau_m} : c} \qquad \frac{c \in s}{\alpha^s : c}$$

For now, we regard classes and sorts as purely syntactic. The details, including the interaction of type classes and derivations, are deferred to §5.

Terms. The language of terms is a conventional simply-typed lambda calculus extended with constants (also denoted by c), whose types can be instantiated at each occurrence. This yields schematic polymorphism, where type inference is still decidable, since arbitrary type abstractions are not allowed.

$$t \quad ::= \quad x \quad | \quad c[\overline{\tau_m}] \quad | \quad t_1\, t_2 \quad | \quad \lambda\, x : \tau.\, t$$

We also write c for $c[]$. Primitive constants include universal quantification $\bigwedge : (\alpha^\top \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop}$, implication $\Longrightarrow : \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop}$, and equality $\equiv : \alpha^\top \Rightarrow \alpha^\top \Rightarrow \mathit{prop}$. A term is called *closed* if it contains no free term variables.

The typing rules for terms are standard. We assume a function Σ that maps constants to their types with canonical type variables $\overline{\alpha_m^\top}$.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash c[\overline{\tau_m}] : \Sigma(c)[\overline{\alpha_m^\top := \tau_m}]}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda\, x : \tau_1.\, t) : \tau_1 \Rightarrow \tau_2}$$

Terms of type *prop* are called propositions and are denoted by ϕ.

Proofs. The language of proofs constitutes another level of lambda calculus on top of terms, in the spirit of the Curry-Howard correspondence. The two versions of abstraction and application correspond to the introduction and elimination of $\bigwedge$ and $\Longrightarrow$, respectively. Proof variables h stand for hypotheses. Axioms and previously-proved theorems are modelled as proof constants *thm*, whose types can be instantiated in a manner similar to term constants.

$$p \quad ::= \quad \lambda\, x : \tau.\, p \quad | \quad \lambda\, h : \phi.\, p \quad | \quad p \odot t \quad | \quad p_1 \bullet p_2 \quad | \quad h \quad | \quad \mathit{thm}[\overline{\tau_m}]$$

Note that *thm* is a meta variable for proof constants. We also write *thm* for *thm*[].

We give the main propositions-as-types typing rules for proofs, which are to be read modulo $\alpha\beta\eta$-equivalence. Like for term constants, the function Σ yields the proposition proved by a proof constant, which may contain free type variables $\overline{\alpha_m^{s_m}}$.

$$\frac{\Gamma, x : \tau \vdash p : \phi}{\Gamma \vdash (\lambda\, x : \tau.\, p) : (\bigwedge x : \tau.\, \phi)} \qquad \frac{\Gamma, h : \phi_1 \vdash p : \phi_2}{\Gamma \vdash (\lambda\, h : \phi_1.\, p) : \phi_1 \Longrightarrow \phi_2}$$

$$\frac{\Gamma \vdash p : (\bigwedge x : \tau.\, \phi) \qquad \Gamma \vdash t : \tau}{\Gamma \vdash (p \odot t) : \phi[x := t]} \qquad \frac{\Gamma \vdash p_1 : \phi_1 \Longrightarrow \phi_2 \qquad \Gamma \vdash p_2 : \phi_1}{\Gamma \vdash (p_1 \bullet p_2) : \phi_2}$$

$$\frac{h : \phi \in \Gamma}{\Gamma \vdash h : \phi} \qquad \frac{\overline{\tau_m : s_m}}{\Gamma \vdash \mathit{thm}[\overline{\tau_m}] : \Sigma(\mathit{thm})[\overline{\alpha_m^{s_m} := \tau_m}]}$$

Pure has further rules and axioms, ensuring that $\equiv$ is an extensional and congruent equivalence relation, which equates $\alpha\beta\eta$-convertible lambda terms and is the bi-implication on propositions.

Sometimes it is convenient to imagine that all type variables $\overline{\alpha_m^{s_m}}$ occurring in a closed proposition ϕ are explicitly quantified on the outermost level. We take the freedom to write this as $\forall \overline{\alpha_m : s_m}.\ \phi$. Similarly, on the level of proofs, the dependency on type variables is expressed using an abstraction-like notation $\lambda\, \overline{\alpha_m : s_m}.\ p$. A proof is called *closed* if all occuring term and type variables are bound in this way. However, these notations are not part of the formal system!

Constant definitions. Pure allows constant definitions, i.e., the introduction of a new constant $c : \tau$ and the definitional axiom $c[\overline{\alpha_m}] \equiv t$, if τ contains exactly the type variables $\overline{\alpha_m}$, t is closed, $t : \tau$, and t contains only type variables $\overline{\alpha_m}$. Since defining equations can always be unfolded, this is a conservative theory extension. Defining equations are written with $:\equiv$ instead of $\equiv$.

2.2 HOL

Higher-order logic (HOL) is embedded as an object-logic in Isabelle/Pure by introducing a type *bool* of classical truth values. A constant *Trueprop* : *bool* ⇒ *prop* embeds booleans into propositions. We write *Trueprop* t as $[\,t\,]$. Object-level quantifiers and connectives are introduced as constants $\forall : (\alpha \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$; $\neg : \mathit{bool} \Rightarrow \mathit{bool}$; $\longrightarrow, \vee, \wedge : \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$; $= : \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$, etc.

Natural-deduction rules $\dfrac{A \quad B}{C}$ can then be expressed as meta-level propositions $A \Longrightarrow B \Longrightarrow C$. For example, these are the introduction and elimination rules for $\forall$ and $\longrightarrow$, and the law of the excluded middle:

$$\begin{aligned}
\text{allI} :&\ \textstyle\bigwedge P : \alpha \Rightarrow \mathit{bool}.\ (\bigwedge x : \alpha.\ [\,P\ x\,]) \Longrightarrow [\,\forall x.\ P\ x\,]\\
\text{spec} :&\ \textstyle\bigwedge (P : \alpha \Rightarrow \mathit{bool})\ (a : \alpha).\ [\,\forall x.\ P\ x\,] \Longrightarrow [\,P\ a\,]\\
\text{impI} :&\ \textstyle\bigwedge P\,Q : \mathit{bool}.\ ([\,P\,] \Longrightarrow [\,Q\,]) \Longrightarrow [\,P \longrightarrow Q\,]\\
\text{mp} :&\ \textstyle\bigwedge P\,Q : \mathit{bool}.\ [\,P \longrightarrow Q\,] \Longrightarrow [\,P\,] \Longrightarrow [\,Q\,]\\
\text{True\_or\_False} :&\ \textstyle\bigwedge P : \mathit{bool}.\ [\,P = \mathit{True} \vee P = \mathit{False}\,]
\end{aligned}$$

In Isabelle, outermost quantifiers and the $[\,\cdot\,]$-embedding are not printed, such that the first rule reads $(\bigwedge x : \alpha.\ P\ x) \Longrightarrow \forall x.\ P\ x$, but we will keep them explicit in this paper, to visualize the division between meta- and object-logic.

HOL has some more rules and axioms, which we omit for brevity. However, two primitive constants are notable: the definite description operator *THE* : $(\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha$, axiomatized by $\bigwedge a.[\,(\mathit{THE}\ x.\ x = a) = a\,]$, and the constant *undefined* : α, which is unspecified as it comes with no axiom.

The approach of modelling the inference rules of the object-logic as theorems in the meta-logic is common to all of Isabelle's object logics. What is special in HOL is that the function space of the object-logic coincides with that of the meta-logic. This works because HOL and Pure are so similar. To make interactions with the framework more explicit, a type class *hol* is used to characterize HOL types.

This class contains types such as *bool*, natural numbers, lists, and it is closed under ⇒. Types such as *prop* or *bool* ⇒ *prop* are not in this class. The HOL axioms are restricted to types which are in the *hol* class.

Type definitions. A speciality of HOL is the ability to define new type constructors from non-empty subsets of existing types. A type definition $\kappa\ \overline{\alpha_m} \cong P$, where $P : \tau \Rightarrow \mathit{bool}$ is a closed term (called the *representing set*) with type variables $\overline{\alpha_m}$, together with a proof of $[\,\exists x.\ P\ x\,]$, gives rise to a bijection between $\kappa\ \overline{\alpha_m}$ and τ, in the form of new constants $\mathit{Rep}_\kappa : \kappa\ \overline{\alpha_m} \Rightarrow \tau$, $\mathit{Abs}_\kappa : \tau \Rightarrow \kappa\ \overline{\alpha_m}$ and an axiom

$$\begin{aligned}\big[\ &(\forall z : \kappa\ \overline{\alpha_m}.\ P\ (\mathit{Rep}_\kappa[\overline{\alpha_m}]\ z)) \wedge \\ &(\forall z : \kappa\ \overline{\alpha_m}.\ \mathit{Abs}_\kappa[\overline{\alpha_m}]\ (\mathit{Rep}_\kappa[\overline{\alpha_m}]\ z) = z) \wedge \\ &(\forall y : \tau.\ P\ y \longrightarrow \mathit{Rep}_\kappa[\overline{\alpha_m}]\ (\mathit{Abs}_\kappa[\overline{\alpha_m}]\ y) = y)\ \big]\,,\end{aligned}$$

which we abbreviate as $\mathrm{typedef}_\kappa$. Notice that this axiom does not specify the value of $\mathit{Abs}_\kappa[\overline{\alpha_m}]\ y$ when $[\,P\ y\,]$ does not hold.

2.3 ZF

First-order logic (FOL), the basis of ZFC, is modelled in Isabelle by two types ι and o, representing individuals and truth values, respectively. Again, an embedding $[\,\cdot\,] : o \Rightarrow \mathit{prop}$ is introduced, along with first-order connectives $\forall : (\iota \Rightarrow o) \Rightarrow o$; $\neg : o \Rightarrow o$; $\longrightarrow, \vee, \wedge : o \Rightarrow o \Rightarrow o$; $= : \iota \Rightarrow \iota \Rightarrow o$, etc., and the usual natural deduction rules for them. ZFC is then added using the standard collection of FOL axioms (see [16] for details) about the constant $\in : \iota \Rightarrow \iota \Rightarrow o$. Note that the axiom schema of replacement is represented as Pure quantification over a predicate. As ι is the type of all sets, the meta level can be used to reason about proper classes ($\iota \Rightarrow o$), operators ($\iota \Rightarrow \iota$), binding structures ($(\iota \Rightarrow \tau) \Rightarrow \tau$), etc.

Function spaces can be constructed using $\rightarrow : \iota \Rightarrow \iota \Rightarrow \iota$, and elements of $A \rightarrow B$ can be constructed using an operator $\mathit{Lambda} : \iota \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota$, which restricts the domain of an operator $f : \iota \Rightarrow \iota$ to a set A. We write $(\lambda\, x \in A.\ f\ x)$ for $\mathit{Lambda}\ A\ f$. The application of such a function to an argument is written using an explicit apply operator $`\ : \iota \Rightarrow \iota \Rightarrow \iota$. While α-conversion is inherited from the framework, β- and η-reduction are conditional rewrite rules.

$$\begin{aligned}\textstyle\bigwedge A\ y\ f.\ [\,y \in A\,] &\Longrightarrow (\lambda\, x \in A.\ f\ x)\,`\,y \equiv f\ y \\ \textstyle\bigwedge A\ B\ f.\ [\,f \in A \rightarrow B\,] &\Longrightarrow (\lambda\, x \in A.\ f\,`\,x) \equiv f\end{aligned} \qquad (\mathsf{ZF}\text{-}\beta\eta)$$

As opposed to HOL, where formulas are just special terms, in FOL the languages of formulas (o) and terms (ι) are syntactically separated. For the sake of uniformity, our translation will map everything to type ι, using the set $\mathbb{B} :\equiv \{0, 1\}$ for truth values, with the interpretation function $\langle\,\cdot\,\rangle :\equiv (\lambda\, x.\ x = 1) : \iota \Rightarrow o$. We must thus define appropriate versions of logical connectives, such as $\widehat{\mathit{True}}$, $\widehat{\mathit{False}}$, $\widehat{\longrightarrow} : \iota$ and $\widehat{=}$, $\widehat{\forall} : \iota \Rightarrow \iota$.

$$\begin{aligned}\widehat{\mathit{True}} &:\equiv 1, \quad \widehat{\mathit{False}} :\equiv 0 \\ \widehat{\longrightarrow} &:\equiv (\lambda\, A, B \in \mathbb{B}.\ \text{if } A = \widehat{\mathit{False}} \text{ then } \widehat{\mathit{True}} \text{ else } B), \\ \widehat{=} &:\equiv (\lambda\, A : \iota.\ \lambda\, x\ y \in A.\ \text{if } x = y \text{ then } \widehat{\mathit{True}} \text{ else } \widehat{\mathit{False}}), \\ \widehat{\forall} &:\equiv (\lambda\, A : \iota.\ \lambda\, P \in A \rightarrow \mathbb{B}.\ P =_{A \rightarrow \mathbb{B}} (\lambda\, x \in A.\ \widehat{\mathit{True}}))\end{aligned}$$

$(\mathbin{\widehat{=}})A\ x\ y$ is written $x =_A y$, and $(\widehat{\forall}\, A\ (\lambda\, x \in A.\ P\ x))$ is written $(\widehat{\forall}\, x \in A.\ P\ x)$.

3 The Basic Translation

The standard set-theoretic model of HOL [17] is based on a set $\mathcal{U}$, which serves as the universe of types. Among other requirements, $\mathcal{U}$ must be closed under function spaces. For example, the set $V_{\omega+\omega} \setminus \{\emptyset\}$, with $V_{\omega+\omega}$ of the cumulative hierarchy (see, e.g., [11]), could be used as the set of all types. While such a relatively small model may be desirable from a foundational point of view, it would make the results of our translation weaker than necessary, and not very intuitive. In fact, it is not necessary that the universe of types forms a set, and so we prefer to use the proper class of all non-empty sets.

The idea underlying the translation is as follows.

- Types τ are mapped to terms denoting non-empty sets $[\![\,\tau\,]\!] : \iota$.
- Type constructors are mapped to operations on sets.
- Terms $t : \tau$ are mapped to terms $[\![\,t\,]\!] : \iota$, such that $[\![\,t\,]\!] \in [\![\,\tau\,]\!]$ holds.
- Application and abstraction are translated to $(\lambda\, x \in A.\ \ldots)$ and ‘.
- Quantification over types (which may only occur on the outermost level) is mapped to Pure quantification over sets.
- Type annotations on variables are mapped to set membership assumptions.
- Proofs are instrumented with non-emptiness and membership derivations, following the typing rules.

Example 1. The statement $\forall \alpha : \{hol\}.\ \bigwedge (x : \alpha)\, (P : \alpha \Rightarrow bool).\ [\, P\ x\,]$ is translated to

$$\bigwedge A : \iota.\ [\, A \neq \emptyset\,] \Longrightarrow (\bigwedge x : \iota.\ [\, x \in A\,] \Longrightarrow (\bigwedge P : \iota.\ [\, P \in A \rightarrow \mathbb{B}\,] \Longrightarrow [\langle\, P\,‘\,x\,\rangle])) \ ,$$

which, after moving quantifiers out, becomes

$$\bigwedge A\, x\, P : \iota.\ [\, A \neq \emptyset\,] \Longrightarrow [\, x \in A\,] \Longrightarrow [\, P \in A \rightarrow \mathbb{B}\,] \Longrightarrow [\langle\, P\,‘\,x\,\rangle] \ .$$

Example 2. The transitivity rule for HOL equality,

$$\forall \alpha : \{hol\}.\ \bigwedge r\, s\, t : \alpha.\ [\, r = s\,] \Longrightarrow [\, s = t\,] \Longrightarrow [\, r = t\,]$$

is translated to

$$\begin{aligned}&\bigwedge A : \iota.\ [\, A \neq \emptyset\,] \Longrightarrow (\bigwedge r : \iota.\ [\, r \in A\,] \Longrightarrow (\bigwedge s : \iota.\ [\, s \in A\,] \Longrightarrow \\ &\qquad (\bigwedge t : \iota.\ [\, t \in A\,] \Longrightarrow [\langle\, r =_A s\,\rangle] \Longrightarrow [\langle\, s =_A t\,\rangle] \Longrightarrow [\langle\, r =_A t\,\rangle]))) \ ,\end{aligned}$$

which, after moving quantifiers out, becomes

$$\begin{aligned}&\bigwedge A\, r\, s\, t : \iota.\ [\, A \neq \emptyset\,] \Longrightarrow [\, r \in A\,] \Longrightarrow [\, s \in A\,] \Longrightarrow [\, t \in A\,] \\ &\qquad \Longrightarrow [\langle\, r =_A s\,\rangle] \Longrightarrow [\langle\, s =_A t\,\rangle] \Longrightarrow [\langle\, r =_A t\,\rangle] \ .\end{aligned}$$

One consequence of our choice of translation is that we need an axiom of global choice, which postulates an operation *choose* satisfying

$$\bigwedge A : \iota.\ [\, A \neq \emptyset \,] \Longrightarrow [\, \mathit{choose}\ A \in A \,]\ .$$

This is a conservative extension of ZFC [4]. The need for it arises not only from the fact that HOL includes a choice operator by itself, but already from the presence of underspecification. While the constant *undefined* has no particular properties in HOL, its translation to ZF must satisfy at least the formula $\bigwedge A : \iota.[\, A \neq \emptyset \,] \Longrightarrow [\ \widehat{\mathit{undefined}}\ A \in A\,]$, which arises from its type. This is exactly the global choice axiom.

In the following presentation of the basic translation scheme, we make a few simplifying assumptions: First, we assume that the theorems we translate do not mix HOL and Pure arbitrarily, but use essentially plain HOL reasoning, except for an outermost layer of $\bigwedge$ and $\Longrightarrow$. Theorems relevant in practice typically have this form (see §6.2 for exceptions). Second, all type variables must be of sort $\{\mathit{hol}\}$ (see §5 for the treatment of other classes). Third, we assume that there are no type or term variables in proofs that do not occur in the corresponding proposition (which is easy to achieve by substituting any ground type or term) and that proofs are closed.

Type constructors κ, term constants c, and proof constants *thm* occurring in proofs must already have translations $\widehat{\kappa}$, $\widehat{c}$ and $\widehat{\mathit{thm}}$. These are either set up manually, as it must be done for the primitives and axioms, or come from a recursive invocation of the translation scheme.

Types and terms are translated as follows:

Translation of types:

$$\begin{aligned} [\![\, \kappa\ \overline{\tau_m} \,]\!] &:= \widehat{\kappa}\ \overline{[\![\, \tau_m \,]\!]} \\ [\![\, \alpha \,]\!] &:= x_\alpha \end{aligned}$$

Translation of terms:

$$\begin{aligned} [\![\, c[\overline{\tau_m}] \,]\!] &:= \widehat{c}\ \overline{[\![\, \tau_m \,]\!]} \\ [\![\, \lambda\, x : \tau.\ t \,]\!] &:= \lambda\, x \in [\![\, \tau \,]\!]\,.\ [\![\, t \,]\!] \\ [\![\, x \,]\!] &:= x \\ [\![\, t_1\ t_2 \,]\!] &:= [\![\, t_1 \,]\!]\ \text{‘}\ [\![\, t_2 \,]\!] \end{aligned}$$

Note that for a type τ, $[\![\, \tau \,]\!]$ is a term (of type ι), not a type. Type instantiations of constants are translated to applications.

In the outer structure of propositions, the domain of universal quantifiers is restricted to the respective sets, and the *Trueprop* embedding is replaced with $[\,\langle \cdot \rangle\,]$. On the outermost level, non-emptiness conditions are added for the sets arising from type variables.

Translation of outer proposition structure:

$$\begin{aligned} [\![\, \textstyle\bigwedge x : \tau.\ \phi \,]\!] &:= \textstyle\bigwedge x : \iota.\ [\, x \in [\![\, \tau \,]\!] \,] \Longrightarrow [\![\, \phi \,]\!] \\ [\![\, \phi_1 \Longrightarrow \phi_2 \,]\!] &:= [\![\, \phi_1 \,]\!] \Longrightarrow [\![\, \phi_2 \,]\!] \\ [\![\, [\, t \,] \,]\!] &:= [\langle\, [\![\, t \,]\!] \,\rangle] \\ [\![\, \forall \overline{\alpha_m} : \{\mathit{hol}\}.\ \phi \,]\!] &:= \textstyle\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\, x_{\alpha_m} \neq \emptyset \,]} \Longrightarrow [\![\, \phi \,]\!] \end{aligned}$$

In the proof transformation given below, the proofs corresponding to typing judgements and non-emptiness of types must be filled in explicitly. We mark the positions where a proof of $[\,P\,]$ must be inserted by placeholders $\{P\}$. This proof, which may refer to hypotheses available in the respective context, is generated by means of a tactic. Moreover, since the original proof is modulo $\alpha\beta\eta$-equivalence and abstraction and application have been translated to their ZF counterparts, we must explicitly normalize them by rewriting with the rules (ZF-$\beta\eta$). For a proposition ϕ, let $\mathsf{norm}\,(\phi)$ denote the normalized proposition. For a proof $p : \phi$, $\mathsf{norm}\,(p)$ denotes a proof of $\mathsf{norm}\,(\phi)$, and $p_1 \bullet_{\mathsf{n}} p_2$ abbreviates $\mathsf{norm}\,(p_1) \bullet \mathsf{norm}\,(p_2)$. We generate the proof of $\mathsf{norm}\,(p)$ from p using Isabelle's simplifier.

Translation of proofs:

$$
\begin{array}{rcl}
[\![\, \lambda\, x : \tau.\, p \,]\!] & := & \lambda\, x : \iota.\ \lambda\, h : [\, x \in [\![\, \tau \,]\!]\,].\ [\![\, p \,]\!] \\
[\![\, \lambda\, h : \phi.\, p \,]\!] & := & \lambda\, h : [\![\, \phi \,]\!]\,.\ [\![\, p \,]\!] \\
[\![\, h \,]\!] & := & h \\
[\![\, p_1 \bullet p_2 \,]\!] & := & [\![\, p_1 \,]\!] \bullet_{\mathsf{n}} [\![\, p_2 \,]\!] \\
[\![\, p \odot t \,]\!] & := & [\![\, p \,]\!] \odot [\![\, t \,]\!] \bullet_{\mathsf{n}} \{[\![\, t \,]\!] \in [\![\, \tau \,]\!]\} \qquad \text{where } \ t : \tau \\
[\![\, \lambda\, \overline{\alpha_m} : \{hol\}.\, p \,]\!] & := & \lambda\, \overline{x_{\alpha_m}} : \iota.\ \lambda\, \overline{h_m : [\, x_{\alpha_m} \neq \emptyset\,]}.\ [\![\, p \,]\!] \\
[\![\, thm[\overline{\tau_m}] \,]\!] & := & \widehat{thm} \odot \overline{[\![\, \tau_m \,]\!]} \bullet_{\mathsf{n}} \overline{\{[\![\, \tau_m \,]\!] \neq \emptyset\}}
\end{array}
$$

Example 3. The proof of the transitivity rule shown previously is based on a substitution rule, one of HOL's axioms:

$$
\begin{array}{l}
\mathrm{subst} : (\forall \alpha : \{hol\}.\ \bigwedge (s\, t : \alpha)\, (P : \alpha \Rightarrow bool).\ [\, s = t\,] \Longrightarrow [\, P\ s\,] \Longrightarrow [\, P\ t\,]) \\
\widehat{\mathrm{subst}} : (\bigwedge x_\alpha : \iota.\ [\, x_\alpha \neq \emptyset\,] \Longrightarrow (\bigwedge s : \iota.\ [\, s \in x_\alpha\,] \Longrightarrow (\bigwedge t : \iota.\ [\, t \in x_\alpha\,] \Longrightarrow \\
\qquad\qquad (\bigwedge P : \iota.\ [\, P \in x_\alpha \to \mathbb{B}\,] \Longrightarrow [\langle\, s =_{x_\alpha} t\,\rangle] \Longrightarrow [\langle\, P\,`\, s\,\rangle] \Longrightarrow [\langle\, P\,`\, t\,\rangle])))) \\
\mathrm{trans} = (\lambda\, (\alpha : \{hol\})\, (r\, s\, t : \alpha)\, (h : [\, r = s\,])\, (h' : [\, s = t\,]). \\
\qquad\qquad \mathrm{subst}[\alpha] \odot s \odot t \odot (\lambda\, x : \alpha.\ r = x) \bullet h' \bullet h) \\
\qquad : (\forall \alpha : \{hol\}.\ \bigwedge (r\, s\, t : \alpha).\ [\, r = s\,] \Longrightarrow [\, s = t\,] \Longrightarrow [\, r = t\,]) \\
\widehat{\mathrm{trans}} = (\lambda\, (x_\alpha : \iota)\, (h_\alpha : [\, x_\alpha \neq \emptyset\,])\, (r : \iota)\, (h_r : [\, r \in x_\alpha\,])\, (s : \iota)\, (h_s : [\, s \in x_\alpha\,]). \\
\qquad\qquad (\lambda\, (t : \iota)\, (h_t : [\, t \in x_\alpha\,])\, (h : [\langle\, r =_{x_\alpha} s\,\rangle])\, (h' : [\langle\, s =_{x_\alpha} t\,\rangle]). \\
\qquad\qquad\quad \widehat{\mathrm{subst}} \odot x_\alpha \bullet_{\mathsf{n}} \{x_\alpha \neq \emptyset\} \odot s \bullet_{\mathsf{n}} \{s \in x_\alpha\} \odot t \bullet_{\mathsf{n}} \{t \in x_\alpha\} \\
\qquad\qquad\qquad \odot (\lambda\, x \in x_\alpha.\ r =_{x_\alpha} x) \bullet_{\mathsf{n}} \{(\lambda\, x \in x_\alpha.\ r =_{x_\alpha} x) \in x_\alpha \to \mathbb{B}\} \\
\qquad\qquad\qquad \bullet_{\mathsf{n}} h' \bullet_{\mathsf{n}} h))
\end{array}
$$

4 Translating Constant and Type Definitions

When translating constant definitions $c[\overline{\alpha_m}] :\equiv t$ where $t : \tau$, the dependency on types $\overline{\alpha_m}$ is made explicit. We introduce a new constant $\widehat{c} : \iota^m \Rightarrow \iota$.

$$\widehat{c} :\equiv (\lambda\, \overline{x_{\alpha_m}} : \iota.\ [\![\, t \,]\!])$$

The translation of the original definition $\llbracket c[\overline{\alpha_m}] \equiv t \rrbracket$ (that is, $\widehat{c}\ \overline{x_{\alpha_m}} \equiv \llbracket t \rrbracket$) is a simple consequence of the above definition and proved automatically. Moreover, the following theorem is deduced, which is needed by the type checking tactic to derive the translation of typing judgements involving c.

$$\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\, x_{\alpha_m} \neq \emptyset \,]} \Longrightarrow [\, \widehat{c}\ \overline{x_{\alpha_m}} \in \llbracket \tau \rrbracket \,]$$

Type definitions $\kappa\ \overline{\alpha_m} \cong P$ with $P : \tau \Rightarrow \mathit{bool}$ are translated to constant definitions of a set $\widehat{\kappa}$ (parameterized by the sets $\overline{x_{\alpha_m}}$ arising from type parameters) and two functions $\widehat{\mathit{Rep}_\kappa}$ and $\widehat{\mathit{Abs}_\kappa}$.

$$\begin{aligned}
\widehat{\kappa} &:\equiv \lambda\, \overline{x_{\alpha_m}} : \iota.\ \{ z \in \llbracket \tau \rrbracket \mid \langle \llbracket P \rrbracket \text{ ` } z \rangle \} \\
\widehat{\mathit{Rep}_\kappa} &:\equiv \lambda\, \overline{x_{\alpha_m}} : \iota.\ \lambda\, z \in \widehat{\kappa}\ \overline{x_{\alpha_m}}.\ z \\
\widehat{\mathit{Abs}_\kappa} &:\equiv \lambda\, \overline{x_{\alpha_m}} : \iota.\ \lambda\, y \in \llbracket \tau \rrbracket.\ \text{if } y \in \widehat{\kappa}\ \overline{x_{\alpha_m}} \text{ then } y \text{ else } \widehat{\mathit{undefined}}\ (\widehat{\kappa}\ \overline{x_{\alpha_m}})
\end{aligned}$$

Since the new type is simply mapped to a subset of the original type, the functions $\widehat{\mathit{Rep}_\kappa}$ and $\widehat{\mathit{Abs}_\kappa}$ become identity mappings. Since $\widehat{\mathit{Abs}_\kappa}$ must be total and always return an element of $\widehat{\kappa}$ to satisfy its type, we use $\widehat{\mathit{undefined}}$.

From these definitions we derive the characteristic property $\llbracket \mathrm{typedef}_\kappa \rrbracket$ for the type definition, as well as the typing lemmas for $\widehat{\mathit{Rep}_\kappa}$ and $\widehat{\mathit{Abs}_\kappa}$ and the fact that the new type constructor preserves non-emptiness. The proof of the latter theorem makes use of the non-emptiness proof provided for the original HOL definition.

$$\begin{aligned}
&\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\, x_{\alpha_m} \neq \emptyset \,]} \Longrightarrow [\, \widehat{\mathit{Rep}_\kappa}\ \overline{x_{\alpha_m}} \in \widehat{\kappa}\ \overline{x_{\alpha_m}} \to \llbracket \tau \rrbracket \,] \\
&\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\, x_{\alpha_m} \neq \emptyset \,]} \Longrightarrow [\, \widehat{\mathit{Abs}_\kappa}\ \overline{x_{\alpha_m}} \in \llbracket \tau \rrbracket \to \widehat{\kappa}\ \overline{x_{\alpha_m}} \,] \\
&\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\, x_{\alpha_m} \neq \emptyset \,]} \Longrightarrow [\, \widehat{\kappa}\ \overline{x_{\alpha_m}} \neq \emptyset \,]
\end{aligned}$$

5 Translating Type Classes and Overloaded Definitions

The translation described so far covers standard HOL. However, in the Isabelle/HOL libraries, even the most basic theories make heavy use of type classes and overloading, which means that our translation must support them to be practically useful.

The basic solution is to employ a preprocessing step which eliminates classes and overloading from theories, producing a theory in plain HOL. Then, the translation from the previous section can be applied to obtain the set-theoretic version.

In this section, we describe the basics of type classes and overloading, which were neglected in §2. Then we show how to compile them away. Although the two mechanisms are typically used together (cf. [7]), we can treat them separately, removing first type classes and then overloading. The outline of this transformation was already pointed out by Haftmann and Wenzel [7], but not considering proof terms and without implementation.

5.1 Type Classes

In a nutshell, classes assigned to a type τ express extra properties of τ that are propagated by the type system. In particular, a sort annotation s on a type variable α^s corresponds to an implicit hypothesis about α.

It is possible to embed types into the term language using a unary type constructor *itself* and a constant $\mathit{TYPE} : \mathit{itself}\ \alpha$, such that the type τ can be represented by the term $\mathit{TYPE}[\tau]$.

To describe the logical properties of types in a class c, a constant $\mathrm{Cl}_c : \mathit{itself}\ \alpha \Rightarrow \mathit{prop}$ is defined. For example, the class of all finite types can be defined as $\mathrm{Cl}_{\mathit{finite}}[\alpha] :\equiv (\lambda\, x : \mathit{itself}\ \alpha.\ [\nexists f : \mathit{nat} \Rightarrow \alpha.\ \mathit{injective}[\mathit{nat}, \alpha]\ f\,])$. The proposition $\mathrm{Cl}_c[\tau]\ (\mathit{TYPE}[\tau])$ serves as the logical interpretation of the type-in-class statement $\tau : c$ as defined in §2.1, and we abbreviate it by $(\!|\, \tau : c\,|\!)$.

When subclass relationships $c_1 \prec c_2$ and arities $\kappa :: (\overline{s_m})c$ are backed up by proofs of $\forall \alpha : \top.\ (\!|\, \alpha : c_1 \,|\!) \Longrightarrow (\!|\, \alpha : c_2 \,|\!)$ and $\forall \overline{\alpha_m} : \top.\ \overline{(\!|\, \alpha_m : s_m \,|\!)} \Longrightarrow (\!|\, \kappa\ \overline{\alpha_m} : c \,|\!)$, respectively, then $(\!|\, \tau : c\,|\!)$ is provable in Pure when $\tau : c$ holds. More precisely,

$$\tau : c \quad \text{implies} \quad \{(\!|\, \alpha^\top : c' \,|\!) \mid \alpha^s \text{ occurs in } \tau,\ c' \in s\} \vdash (\!|\, \tau^\top : c \,|\!)\ ,$$

where $\tau^\top$ denotes the type τ with all sort annotations replaced by $\top$.

To reflect this connection between the type system and the inference system, Pure provides a special proof constructor $\mathbf{ofclass}\,\tau\, c$ and the rule

$$\frac{\tau : c}{\Gamma \vdash \mathbf{ofclass}\,\tau\,c : (\!|\, \tau : c \,|\!)}\ .$$

The **ofclass** constructor serves as a placeholder for an explicit proof of $(\!|\, \tau : c\,|\!)$, which can always be constructed in a straightforward manner, following the rules for $\tau : c$.

Elimination of classes. To eliminate classes from propositions, we remove all sort annotations from type variables and replace them by explicit assumptions. Thus, a proposition $\forall \overline{\alpha_m : s_m}.\ \phi$ becomes $\forall \overline{\alpha_m} : \top.\ \overline{(\!|\, \alpha_m : s_m \,|\!)} \Longrightarrow \phi$.

Example 4. The proposition

$$\forall \alpha : \{\mathit{finite}\}.\ \bigwedge f : \alpha \Rightarrow \alpha.\ [\,\mathit{injective}[\alpha, \alpha]\ f \longleftrightarrow \mathit{surjective}[\alpha, \alpha]\ f\,]$$

is converted to

$$\forall \alpha : \top.\ (\!|\, \alpha : \mathit{finite} \,|\!) \Longrightarrow \bigwedge f : \alpha \Rightarrow \alpha.\ [\,\mathit{injective}[\alpha, \alpha]\ f \longleftrightarrow \mathit{surjective}[\alpha, \alpha]\ f\,]\ .$$

Eliminating classes from proofs amounts to this explication of sort constraints on type variables and replacing the proof constructors $\mathbf{ofclass}\,\tau\,c$ with derivations for $(\!|\, \tau : c\,|\!)$, using the newly-introduced assumptions on type variables.

A subtlety arises when the proof of a proposition contains type variables that do not occur in the proposition itself. Since the presence of such a type variable β^s constitutes an implicit assumption that the sort s is inhabited (i.e., $(\!|\, \tau : s \,|\!)$ holds for some ground type τ), we must introduce an extra hypothesis $(\!|\, \alpha : s \,|\!)$ for some canonical α. These sort inhabitedness hypotheses are tracked by Isabelle's inference kernel, ensuring soundness even when proof term recording is disabled. Term variables do not need a corresponding treatment, since types are always inhabited.

5.2 Overloading

Overloaded constants can have multiple defining equations on different types. To simplify the presentation, we assume that types of constants have exactly one parameter α, which allows us to write $c[\tau]$ instead of $c[\overline{\tau_m}]$. The treatment below is easily extended to the general case.

An overloaded constant c is specified by giving its type τ and multiple defining equations $c[\tau_i] :\equiv t_i : \tau[\alpha := \tau_i]$ for different type instances τ_i, where all type variables in the closed t_i have to occur in $\tau[\alpha := \tau_i]$, or equivalently in τ_i.

We write $c[\tau] \rhd d[\sigma]$ iff there exists a type τ_i, a substitution θ, and a defining equation $c[\tau_i] :\equiv t_i$, such that $\tau = \theta(\tau_i)$ and t_i contains a constant occurrence $d[\sigma']$ where $\sigma = \theta(\sigma')$. This relation on constants with types is called the *dependency relation.*

A system of overloaded definitions is well-formed, if the defining equations for any constant do not overlap (i.e., different τ_i and τ_j are not unifiable after renaming variables apart) and the dependency relation $\rhd$ is terminating. The latter property ensures that unfolding definitions cannot lead to non-termination and is undecidable [13], but Isabelle approximates this by a simpler criterion [7].

Note that this notion of overloading is more than just the use of a single name for multiple logical constants: The definition of another constant $d[\alpha]$ may refer to an overloaded constant $c[\tau]$, with the effect that the meaning of d also depends on instantiations of α. Then d is called *indirectly overloaded.* We call a constant $c[\tau]$ *overloading-free* iff it is primitive or has exactly one defining equation, whose right-hand side mentions only overloading-free constants.

A constant occurrence $c[\tau]$ in a term is called *resolvable* iff $\tau = \theta(\tau_i)$ for some substitution θ and some τ_i from a defining equation $c[\tau_i] :\equiv t_i$. Since defining equations do not overlap, τ_i and θ are then uniquely defined and we say $c[\tau]$ is *resolvable via* θ on $c[\tau_i]$.

Eliminating Overloading. The idea behind the elimination of overloading resembles the dictionary construction used to eliminate type classes from Haskell programs. For overloaded constants c, an overloading-free *dictionary constant* c_i is defined for each of the equations $c[\tau_i] :\equiv t_i$, abstracting out unresolvable overloading in t_i.

Concrete occurrences $c[\tau]$ can then be replaced by so-called dictionary terms: If $c[\tau]$ is resolvable, the corresponding dictionary constant is used, possibly passing through other dictionaries. If $c[\tau]$ is not resolvable, a dictionary variable $D_{c[\tau]}$ is inserted.

Formally, for a constant c and type τ, we define the set

$$\mathsf{dicts}(c[\tau]) := \begin{cases} \bigcup_{c[\tau] \rhd d[\tau']} \mathsf{dicts}(d[\tau']) & \text{if } c[\tau] \text{ is resolvable.} \\ \{c[\tau]\} & \text{otherwise.} \end{cases}$$

This set is well-defined and finite, since the dependency relation $\rhd$ is terminating and finitely branching. Lifting this to arbitrary terms, we define $\mathsf{dicts}(t) := \bigcup_{i\in\{1,\ldots,k\}} \mathsf{dicts}(d_i[\sigma_i])$, where $\overline{d_k[\sigma_k]}$ are the occurrences of constants in t. We assume some canonical order on $\mathsf{dicts}(t)$.

To eliminate overloading from a term, the mapping $[\![\,\cdot\,]\!]_{\mathsf{ov}}$ replaces (indirectly) overloaded constant occurrences $c[\tau]$ with dictionaries. Overloading-free constants and the rest of the term structure is preserved. If $c[\tau]$ is not resolvable, a dictionary variable is inserted.

$$[\![\, c[\tau] \,]\!]_{\mathsf{ov}} := D_{c[\tau]}$$

If $c[\tau]$ is resolvable via θ on $c[\tau_i]$, the corresponding dictionary constant c_i is used, passing dictionaries through:

$$[\![\, c[\theta\, \tau_i] \,]\!]_{\mathsf{ov}} := c_i[\theta\, \overline{\alpha_m}]\ \overline{[\![\, d_k[\theta\, \sigma_k] \,]\!]_{\mathsf{ov}}}\ ,$$

where $\{\overline{d_k[\sigma_k]}\} = \mathsf{dicts}(c[\tau_i])$ and $\overline{\alpha_m}$ are the type variables in τ_i.

The definitions of the dictionary constants c_i arise from the defining equations $c[\tau_i] :\equiv t_i$:

$$c_i[\overline{\alpha_m}] :\equiv (\lambda\, \overline{D_{d_k[\sigma_k]}}.\ [\![\, t_i \,]\!]_{\mathsf{ov}})$$

where $\overline{\alpha_m}$ are the type variables in τ_i and $\{\overline{d_k[\sigma_k]}\} = \mathsf{dicts}(c[\tau_i])$.

At the outermost level of propositions, we abstract over the generated dictionary variables.

$$[\![\, \forall \overline{\alpha_m}.\ \phi \,]\!]_{\mathsf{ov}} \ := \ \forall \overline{\alpha_m}.\ \bigwedge \overline{D_{d_k[\sigma_k]}}.\ [\![\, \phi \,]\!]_{\mathsf{ov}} \qquad \text{where } \{\overline{d_k[\sigma_k]}\} = \mathsf{dicts}(\phi)\ .$$

Proofs are structurally unchanged, but constant definitions of overloaded constants are replaced by theorems about the resulting dictionary term.

Example 5. Assume an infix type constructor $\times$ with pair syntax $(\cdot,\cdot)$ and projections fst and snd, and a type nat where $\mathit{nat\text{-}plus} : \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$ defines addition. An overloaded addition function $\mathit{plus} : \alpha \Rightarrow \alpha \Rightarrow \alpha$ could be defined by the following equations.

$$\begin{aligned} \mathit{plus}[\mathit{nat}] &:\equiv \mathit{nat\text{-}plus} \\ \mathit{plus}[\alpha \times \beta] &:\equiv \lambda\, x\, y : \alpha \times \beta.\ (\mathit{plus}[\alpha]\ (\mathit{fst}\ x)\ (\mathit{fst}\ y), \mathit{plus}[\beta]\ (\mathit{snd}\ x)\ (\mathit{snd}\ y)) \end{aligned}$$

The overloading elimination introduces constants

$$\begin{aligned} \mathit{plus}_1 &:\equiv \mathit{nat\text{-}plus} \\ \mathit{plus}_2[\alpha, \beta] &:\equiv (\lambda\, (D_{\mathit{plus}[\alpha]} : \alpha \Rightarrow \alpha \Rightarrow \alpha)\, (D_{\mathit{plus}[\beta]} : \beta \Rightarrow \beta \Rightarrow \beta)\, (x\, y : \alpha \times \beta). \\ &\qquad (D_{\mathit{plus}[\alpha]}\ (\mathit{fst}\ x)\ (\mathit{fst}\ y), D_{\mathit{plus}[\beta]}\ (\mathit{snd}\ x)\ (\mathit{snd}\ y))) \end{aligned}$$

from which dictionaries for plus on arbitrarily nested tuples can be built, e.g.,

$$[\![\, \mathit{plus}[(\mathit{nat} \times \mathit{nat}) \times \mathit{nat}] \,]\!]_{\mathsf{ov}} = \mathit{plus}_2[\mathit{nat} \times \mathit{nat}, \mathit{nat}]\ (\mathit{plus}_2[\mathit{nat}, \mathit{nat}]\ \mathit{plus}_1\ \mathit{plus}_1)\ \mathit{plus}_1\ .$$

Notice the dictionary variables in the following translation of a simple theorem.

$$
\begin{aligned}
&[\![\forall \alpha\, \beta : \top.\ [\, \mathit{comm}[\alpha]\ \mathit{plus}[\alpha]\,] \Longrightarrow [\, \mathit{comm}[\beta]\ \mathit{plus}[\beta]\,] \Longrightarrow [\, \mathit{comm}[\alpha \times \beta]\ \mathit{plus}[\alpha \times \beta]\,]\,]\!]_{\mathsf{ov}} \\
&= \forall \alpha\, \beta : \top.\ \textstyle\bigwedge (D_{\mathit{plus}[\alpha]} : \alpha \Rightarrow \alpha \Rightarrow \alpha)\ (D_{\mathit{plus}[\beta]} : \beta \Rightarrow \beta \Rightarrow \beta). \\
&\qquad [\, \mathit{comm}[\alpha]\ D_{\mathit{plus}[\alpha]}\,] \Longrightarrow [\, \mathit{comm}[\beta]\ D_{\mathit{plus}[\beta]}\,] \\
&\qquad\quad \Longrightarrow [\, \mathit{comm}[\alpha \times \beta]\ (\mathit{plus}_2[\alpha, \beta]\ D_{\mathit{plus}[\alpha]}\ D_{\mathit{plus}[\beta]})\,]
\end{aligned}
$$

Overloaded constants appearing in representing sets of type definitions are replaced in the same way using $[\![\cdot]\!]_{\mathsf{ov}}$. However, a fine point should be noted here: Abstracting out unresolvable overloading would give rise to dependent types. This is no problem in the set-theoretic interpretation, but as the overloading elimination is currently implemented as a preprocessing step on the HOL side, it does not handle such overloading. This can be fixed by collapsing the different parts of the translation. However, to remove unresolvable overloading from type definitions while staying in HOL, one has to eliminate the type definition altogether, replacing it by its representing type together with a predicate.

It appears that this subtle issue was overlooked in all proof sketches of conservativity of overloading so far [13, 19, 7]. Practically, this form of overloading seems to be quite rare. It does not occur in the main HOL image, but a few instances exist in the HOLCF development [12].

6 Discussion and Limitations

We briefly discuss some limitations of our approach and current implementation.

6.1 Replacing constants and types

The translations of some concepts defined in HOL are not as one would like to use them in set theory. For example, the translation $[\![\mathit{int}]\!]$ of the HOL integers should be the set $\mathbb{Z}$, which is already defined in Isabelle/ZF but happens to be a different (though isomorphic) object.

This problem is common to all proof translation tools and there is no general solution yet, apart from configuration to match up the concepts with equivalent ones. For example, in the proof translation from HOL4 and HOL Light to Isabelle/HOL [14], concepts can be replaced with others that behave in the same way, possibly with minor modifications such as argument order. In principle, our translation supports such replacements, but currently this requires tedious manual configuration and equivalence proofs.

6.2 Interactions of Pure and HOL

Recall that our translation inserts explicit constants *Lambda* and ‘ for abstraction and application in object-logic statements but leaves the outer proposition structure consisting of $\bigwedge$ and $\Longrightarrow$ intact. The advantage of this approach is that the results adhere to Isabelle's standard rule format. But in a few corner cases, the translation becomes difficult. For example, consider the following substitution rule for Pure equality:

$$\forall \alpha\ \beta : \top.\ \bigwedge (f : \alpha \Rightarrow \beta)\ (x : \alpha)\ (y : \alpha).\ x \equiv y \Longrightarrow f\ x \equiv f\ y$$

This is clearly a rule of the framework, as it contains no connectives of any object-logic. The translation should thus keep the rule as it is. On the other hand, when α and β are instantiated with HOL types, then it should turn the types into sets and translate the application $f\ x$ to $f\,`\,x$. This shows that separating the framework from the object-logic cannot be done in a modular way. We have solved this problem by producing several variants for such rules, for different type instantiations.

An alternative translation that we would like to explore in the future is to abandon the distinction between Pure and HOL connectives, mapping everything to sets. Of course, dependencies on types must still use the framework, as type constructors and polymorphic constants are not sets in our model.

6.3 Performance

In its current implementation, the translation is expensive both in terms of time and memory. We exercised it on the main HOL image and HOL's number theory. Translating Fermat's little theorem and all its dependencies from the basic axioms takes 80 minutes and 1.6 GB of main memory on stock hardware. Measurements indicate that this performance cost is mainly due to extra $\beta\eta$-normalization of terms and type checking proofs becoming explicit in ZF. Obviously, more work is needed here to improve the performance.

7 Conclusion

Our translation maps all Isabelle/HOL primitives to set theory. By translating the proofs along with the theories, we can guarantee soundness of the overall method. The fact that we uncovered a notable omission in all previous proofs of conservativity of overloading (see §5.2) shows that our approach of "implemented semantics" is also useful for better understanding the logical system. Moreover, having an implementation facilitates experiments and modifications and will hopefully stimulate the further development of Isabelle/ZF.

Our elimination of type classes and overloading can also be of help when translating Isabelle/HOL developments to other systems. Previously, these concepts could only be translated by using extra-logical abstraction mechanisms provided by the OCaml language [10].

Acknowledgements. We would like to thank Stefan Berghofer, Florian Haftmann, and Makarius Wenzel, who patiently answered questions about the interior of the Isabelle system. Sascha Böhme and Armin Heller read a draft of this paper on short notice and made helpful comments.

References

1. M. Bortin, E. Broch Johnsen, and C. Lüth. Structured formal development in Isabelle. *Nordic Journal of Computing*, 13:1– 20, 2006.
2. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988.
3. U. Furbach and N. Shankar, editors. *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *LNAI*. Springer, 2006.
4. H. Gaifman. Global and local choice functions. *Israel Journal of Mathematics*, 22(3-4):257–265, 1975.
5. M. J. C. Gordon. Set theory, higher order logic or both? In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs '96)*, volume 1125 of *LNCS*, pages 191–201. Springer, 1996.
6. M. J. C. Gordon. Twenty years of theorem proving for HOLs: Past, present and future. In O. Ait Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 1–5. Springer, 2008.
7. F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *LNCS*. Springer, 2007.
8. P. V. Homeier. The HOL-Omega logic. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 244–259. Springer, 2009.
9. L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, 1999.
10. S. McLaughlin. An interpration of Isabelle/HOL in HOL Light. In Furbach and Shankar [3], pages 192–204.
11. Y. N. Moschovakis. *Notes on Set Theory*. Springer, 1994.
12. O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF=HOL+LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.
13. S. Obua. Checking conservativity of overloaded definitions in higher-order logic. In F. Pfenning, editor, *Term Rewriting and Applications (RTA 2006)*, volume 4098 of *LNCS*, pages 212–226. Springer, 2006.
14. S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In Furbach and Shankar [3], pages 298–302.
15. L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.
16. L. C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11:353–389, 1993.
17. A. Pitts. The HOL logic. In M. Gordon and T. Melham, editors, *Introduction to HOL: A theorem proving environment for Higher Order Logic*, pages 191–232. Cambridge University Press, 1993.
18. M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *LNCS*. Springer, 1989.
19. M. Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, pages 307–322, London, UK, 1997. Springer.
20. F. Wiedijk. The QED manifesto revisited. In R. Matuszewski and A. Zalewska, editors, *From Insight To Proof – Festschrift in Honour of Andrzej Trybulec*, pages 121–133. University of Białystok, 2007.